High Speed and Robust Event Correlation

Shaula Alexander Yemini, Shmuel Kliger, and Eyal Mozes, System Management Arts (SMARTS)
Yechiam Yemini and David Ohsie, Columbia University

The authors describe a network management system and illustrate its application to managing a distributed database application on a complex enterprise network.

Network operations management consists mainly of monitoring, interpreting, and handling *events*, where an event is defined as an exceptional condition in the operation of the network. Events are often the result of underlying problems such as hardware or software failures, performance bottlenecks, configuration inconsistencies, or intrusion attempts. Since a single *problem* event in one resource may cause many *symptom* events in related resources, operations staff must be able to *correlate* the observed events to identify and localize underlying problems.

Figure 1 depicts an event propagation scenario illustrating the range of challenges associated with event management. A client application on local area network LAN 1 wishes to exchange data over a Transmission Control Protocol (TCP) connection with a database server on LAN 2. The TCP connection transports Internet Protocol (IP) packets through routers A, B, C, and D, which connect the LAN domains through a router backbone domain. The router backbone domain uses physical-layer wide area network (WAN) domains; for example, routers C and D use a T3 link provided by WAN 2. Thus, the example of Fig. 1 comprises a number of distinct domains: the database application, the two LANs, the router backbone, and the two WANs. Typically, each of these domains would be administered by a different organization.

Consider the following problem. A clock at an interface in WAN 2 that supports the T3 link intermittently loses synchronization approximately four times a second for a period of 0.25 ms. This results in an intermittent burst of noise causing a loss of 0.1 percent in the T3 link capacity. This relatively minor noise at the physical layer causes bit errors in a large number of packets routed over the C-D link. Bit errors will cause packet losses either at the routers (if the IP header is corrupted) or at the destinations (if the header is intact but the body is corrupted).

Regardless of where packets are lost, the performance of TCP connections over the C-D link is seriously hampered. TCP uses a window adaptation technique that interprets packet loss as an indication of network congestion and reduces the window to size one, allowing the window size to increase gradually until the next packet loss. Because the intermittent noise occurs frequently, the TCP window size will not grow much above one. As a result, database transactions by remote clients will last longer. Since transactions typically lock database records, the periods over which records are locked by remote clients will extend significantly. This will cause both degradation in the database server performance for all transactions, local and remote, and frequent aborts of remote transactions that trigger a timeout of the database record locking mechanism.

This scenario illustrates three important points. First, it illustrates how *problems propagate among related objects*, possibly amplified by various protocol mechanisms. Second, *a single problem can cause numerous observable events in multiple domains*. The database application domain will see performance problems in clients, with transactions requiring longer completion times due to increased aborts. The database server domain will witness server performance slowdowns. The server and clients' systems and the routers C and D will observe packet losses. Third, *some problems are not observable where they originate*. The WAN 2 domain may observe minor error events at the T3 interface, but these events may be indistinguishable from normal operating noise. The WAN 2 domain, where the problem originates, may thus be totally unaware that there is a problem.

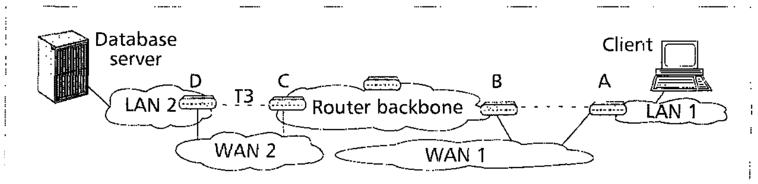
To determine which events to monitor and how to analyze them, operations staff must be intimately familiar with the operational parameters of each managed object and the significance of its events. Furthermore, operations staff for different domains must coordinate analyzing their respective domains, because the WAN 2 interface problem is not observable in the domain where it originates and can only be isolated by correlating information from the enterprise domain layered over it.

Different domains may need to invoke different handlers to fix their respective problems. For example, the database server can be configured to reduce aborts by increasing lock timeout intervals; TCP connections can be configured to use backup network links; the router backbone can use alternate routes to reach LAN 2; and WAN 2 can reconfigure the T3 service to use alternate links until the faulty equipment is fixed.

Shaula Alexander Yemini, Shmuel Kliger, and Eyal Mozes were supported in part by NSF grant DMI-94-61773. David Ohsie's research was supported in part by NSF grant IRI-94-13847. The work was performed while Yechiam Yemini was on sabbatical leave at System Management Arts.

The event management activities described above are currently handled manually. Operations staff monitor and correlate events, and handle identified problems, periodically communicating with one another to analyze problems. This manual processing does not scale to the growing speed, complexity, and size of today's networked systems. The detailed knowledge required to analyze events of an ever growing collection of interacting managed objects exceeds human capacity. Operators cannot keep up with the increasing rates at which events are generated. Mission-critical network-based applications cannot tolerate such an ad hoc, labor-intensive, and error-prone approach.

The design goal of the DECS was therefore to provide a comprehensive software system that *automates* event management for multidomain networks of arbitrary scale and complexity. In what follows, we describe some of the technical challenges of automating event management and the novel approach taken in the DECS.



■ **Figure 1.** *A sample multidomain networked system.*

The Challenges of Automating Event Management

Every automated event management system must be able to accurately model and store knowledge of the underlying networked system and its associated events. This includes the diagnostic knowledge possessed by operations staff. In addition, it must provide algorithms to analyze this knowledge in the context of the current system state in order to detect problems as they occur. Finally, it is desirable that both the knowledge model and runtime diagnostic processing scale well to large and complex networks. In this section, we examine each of these requirements in detail.

Modeling Event Information in a Machine-Processable Form

The first step in automating event analysis is to develop data structures to store knowledge about the managed objects and their associated events. The managed objects include network objects as well as attached systems and applications objects. This event knowledge can be partitioned into two categories:

Static Information — Associated with each class of managed object, independent of the system context in which it appears. Much of this information is available from operations manuals. For example, the events associated with the routers in the example of Fig. 1 could be Simple Network Management Protocol (SNMP) traps and threshold events computed from management information base (MIB) variables. This information would typically be described in router operations manuals.

Dynamic Information — About the specific configuration in which the managed object appears. This information affects the causal propagation of events. For example, if the client application were using a local server there would be no reason to suspect problems in the router backbone. Today's systems change rapidly; it is not unusual for a system to undergo additions, removals or upgrades to software or hardware components every day. Accurate networked system analysis requires up-to-date information about causal propagation across objects. Rule-based systems typically use rules that combine both types of information. As a result, new rules must be developed whenever the system configuration changes, limiting these systems' capability to handle dynamically changing systems.

Automating Monitoring and Analysis of Event Information

Operations staff employ a variety of ad hoc techniques to analyze observable events and identify problems. A major challenge in automating this process is developing the correlation algorithms to replace these ad hoc processes. Because events propagate across related objects, an exceptional behavior of an object does not necessarily indicate a problem with that particular object. It is necessary to correlate events across related objects to isolate the root cause. Since a networked system's configuration evolves dynamically, it is necessary to have correlation algorithms that adapt automatically.

The operational data available for monitoring provides important input to event analysis, but its huge volumes pose a performance problem for real-time processing. Networked systems often include tens of thousands of managed objects. Complex objects, such as routers, can monitor over 5000 operational (MIB) variables. An important challenge is to select the variables that provide the most relevant information.

In a busy networked system, it is not unusual for events to be lost, delayed, or spuriously generated (false alarms). Rule-based and finite-state-machine-based systems are very sensitive to such "noise" in the event stream, limiting their use for real-world networks. Thus, another important challenge is to develop correlation algorithms that are insensitive to noise.

Scaling to Arbitrarily Large and Complex Systems

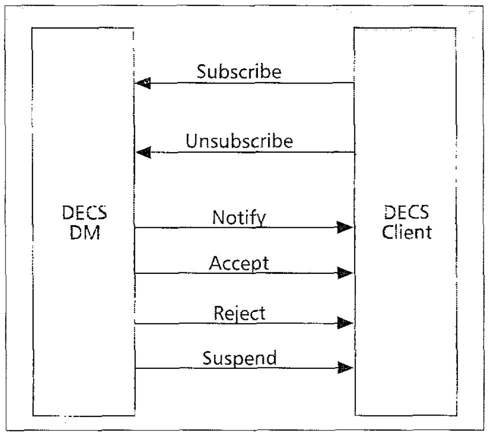
Network devices are increasing in complexity, speed, and features. Networks themselves are becoming larger, more complex, faster, and ever more heterogeneous. In such emerging networks, management applications must reliably access and process data at speeds that cannot be achieved in a centralized system. Effective network management requires a distributed architecture that divides the data and its processing among multiple cooperating event manager nodes. Moreover, many networked systems are already organized as federated multidomain organizations. In the example of Fig. 1, a telephone company may manage the WANs, while local administrators would typically manage each LAN domain. An event management system must be capable of correlating information and coordinating control across distributed management domains.

A Distributed Event Management Architecture

The fundamental building block of DECS is the Domain Manager (DM), a software component that automates event management in a single domain. Each DM monitors observable events in its domain, correlates events to determine problems, and invokes predefined problem handlers. A DECS system can consist of a single DM or a collection of cooperating DMs.

The DECS architecture is *recursive*: any domain, including the global networked system domain, can be partitioned into smaller domains. Partitioning a complex networked system into domains can be motivated by modularity and scalability concerns and/or to account for existing organizational boundaries. In the next section, we present a possible partitioning of the example system in Fig. 1 into enterprise and router backbone domains.

Figure 2 depicts the external interfaces of the overall DECS system and any of its DM partitions. We use the term DECS/DM when referring to either the global DECS or to any of its contained DMs.



■ **Figure 2.** *DECS DM client/server interfaces.*

As the figure shows, clients can subscribe to receive notifications of particular problems; notifications will be sent by the DECS/DM server when these problems are detected.

There are many applications that can be designed as DECS clients. For example, a Network Management System (NMS)-based graphic display application could subscribe to a DECS/DM for problem notification and display results to operators; trouble-ticketing systems could save DECS/DM notifications in the trouble ticket database. Another potential DECS DM client is a policy-based problem-handling application, where users can specify what action to take upon the detection of specific problems.

The clients for a particular DECS/DM can include other DECS/DMs. This is very useful because an event in one domain can be a symptom for a related domain. For example, slow TCP connections in the enterprise domain are a symptom of a problem in the router domain, a problem which cannot be directly observed from within the router domain.

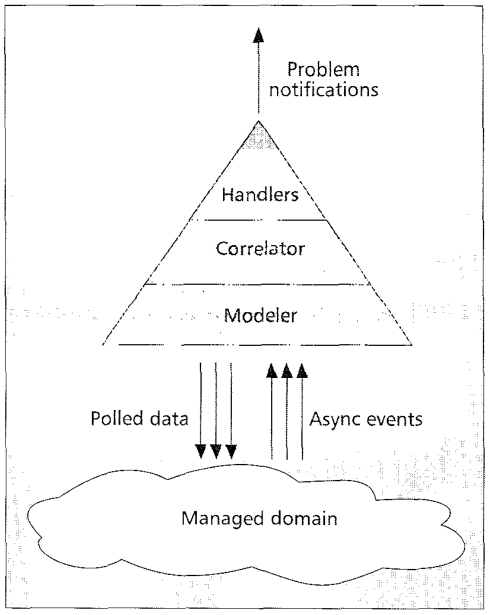
Figure 3 depicts the internal components of a DECS/DM. The Modeler maintains a current model of the domain and its event information by either polling data sources such as network management MIBs or receiving asynchronous event notifications such as SNMP traps. The Correlator correlates domain events and operational data obtained through the Modeler to analyze root cause problems and emits problem notifications to other processes. These components are described in the following sections.

Event Information Modeling

This section describes the modeling framework used by the DECS Modeler as its source of domain management information.

The Modeling Framework

The following information must be captured to enable automated event analysis in a networked system: the classes and instances of managed objects in the managed domain, the problems that can originate in each object, the relationship of each object to other objects, and propagation pattern of events from one object to another along relationships. This information must be provided for all managed objects: network components, attached systems, and the services and applications that run on them. Information must also be provided about resources that do not directly participate in communication, such as power supplies or PC hardware.



■ **Figure 3.** *A DECS domain manager.*

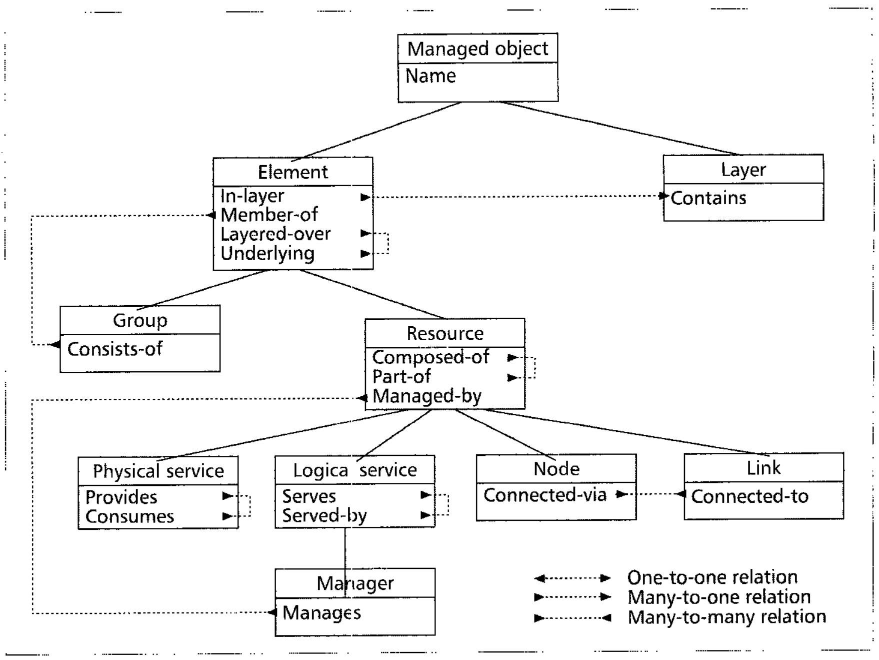
SMARTS' Modeler is a runtime repository of management information used by the DECS correlator component. The Modeler's semantic framework [1] is based on the Netmate model described in [2 and 3]. It uses an object-oriented paradigm to represent networked system *classes*, their attributes and relationships, and their event information. A class is a template for a set of object instances, describing common structural and behavioral properties. All objects that are instances of a class share the structural and behavioral properties of the class.

Modeler classes are organized along an *inheritance hierarchy*. Inheritance is particularly valuable for networked systems modeling because typical networks contain numerous types of objects, such as computers, internetworking devices, and databases; in addition, each generic type is represented by many different subtypes. Inheritance allows management applications to treat objects generically, ignoring their specific details when they are not relevant to the problem at hand.

The Modeler emphasizes *relationship properties* which capture the dependencies between managed objects. Relationship properties represent information such as a node being connected to a particular link, a TCP connection being layered over a particular IP link, a client using the services of a particular server, an application being executed on a particular computer, and so on. Knowledge of such relationships is essential for automating problem management. Relationships are class properties, so all objects instantiated from a given class have similar types of relationships.

The "MODEL Language" section presents a formal language for specifying arbitrary managed object classes for the Modeler. However, we have found a particular set of abstract classes to be extremely powerful. We call this set the *Netmate hierarchy* because it extends Netmate's [3] network classes with classes for system and application objects. The Netmate hierarchy is depicted in Fig. 4. Note that relationships can be one-to-one, many-to-one, or many-to-many, and that each relationship has an inverse. That is, for each relationship from A to B, there exists a corresponding relationship from B to A. For example, *Man-*

ages is a relationship between objects of the *Manager* class and objects of the *Resource* class. If *Manager* **M** *Manages* a *Resource* **O**, there is a relationship from *Resource* to *Managers*, *Managed-by*, such that **O** is in the *Managed-by* relationship with **M**.



■ **Figure 4.** *The Netmate class hierarchy.*

Event Knowledge

We use the following terminology for event information. A *problem* is an event that can be handled directly; in the example of Fig. 1, the faulty interface is a problem. Some problems are directly observable, while others can be observed only indirectly, by observing their symptoms. *Symptoms* are defined as events that are observable; for example, degraded application performance is a symptom of the faulty interface problem. Symptoms cannot be handled; to make a symptom go away, it is necessary to handle its root cause problem. Some events are neither problems nor symptoms, while other events can be both problems and symptoms.

Relationships are an essential component of correlation, because problems and symptoms propagate from one object to another along relationships. For example, in the example of Fig. 1, noise in the WAN physical layer causes bit errors in the C-D link layered over it, loss of packets at routers connected to this link, poor performance of TCP connections layered over this link, and ultimately frequent transaction aborts in the database serving the client communicating with the database over this TCP connection.

The Modeler's event knowledge contains the following information for each class of managed objects:

- The data attributes of objects of this class (e.g., MIB variables).
- The relationships in which an instance of the class can be involved.
- The set of events that are observable within instances of this class. These are typically defined by expressions over data attributes (e.g., a particular MIB variable is above threshold), or by asynchronous event notifications.
- The problems that can originate within instances of this class.
- The set of events caused by each problem. This set can include events within the object, as well as events in other objects to which the object is related. For example, the problem *SyncLoss* in a object of class *WANInterface* can propagate to the event *BitErrorRate* > *threshold* in links related to the *WANInterface* object via the *Connected_to* relationship.
- The events and/or problems that are *exported* by instances of the class.

The MODEL Language

We have developed a formal object/event specification language called the Managed Object Definition Language (MODEL) [4]. MODEL can specify all the event information described above. The MODEL language is an extension of CORBA IDL [5]. It uses IDL syntax wherever possible; for example, it uses IDL syntax to define classes, data attributes, and methods. It adds new syntactic constructs to specify semantics that cannot be specified in CORBA IDL: relationships, events, problems, and causal propagation.

The MODEL language compiler generates C++ code to implement classes defined in MODEL. The MODEL language is extremely concise. In a complex network scenario, 40 classes comprising a total of 1170 lines of MODEL code compiled to 91,000 lines of C++ code. The MODEL specification described a system with 4000 managed object instances representing a total of 9500 problem and 6000 symptom instances. If the generated code were developed manually, the development effort would be huge.

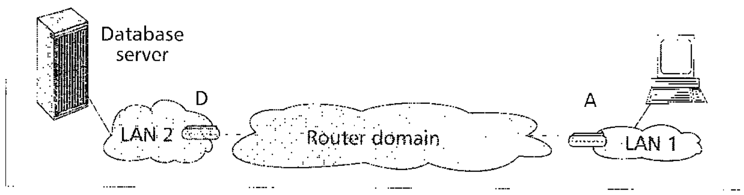
Using the Modeler to Augment Standard MIBs

The Modeler adds a layer of semantics to the data in standard MIBs. It provides a much higher level of abstraction than SNMP or Common Management Information Protocol (CMIP) MIBs and includes event management information which cannot be captured by SNMP or CMIP MIBs. Modeler objects typically correspond to larger-grain objects than MIB objects (e.g., a typical Modeler object represents an SNMP group or table). To support use of the Modeler in conjunction with standard MIBs, the MODEL language contains constructs for inclusion of standard MIB subtrees as instrumentation for Modeler objects.

Modeling the Sample Scenario

Having described the modeling framework, we will now illustrate its application to the example of Fig. 1. Note that there are several ways to partition the sample system into management domains. The partitioning typically depends on the number of managed objects in the system, the level of granularity modeled for each managed object class, and the complexity of dependencies between managed objects.

For simplicity, we'll assume here that the network is partitioned into two management domains: the enterprise domain, which includes everything outside the router backbone domain, and the router backbone domain connecting the two enterprise sites. The enterprise domain consists of the client and server applications, the systems they run on, and the



■ **Figure 5.** *The enterprise view.*

LANs to which they are attached. The router backbone domain consists of the router backbone and the two WANs. Each domain has a dedicated DECS/Domain Manager to monitor, correlate, and handle its events. Each DM may subscribe to problem notifications from the other DMs.

From the point of view of the enterprise domain, the entire router domain is a single entity that connects the two LAN domains (Fig. 5). Any problem in the connection between the two LANs is viewed as a router domain problem. Within the router domain, finer-grained correlation can determine the particular problem using symptoms from the enterprise domain.

In the scenario of Fig. 1, the problem is that a clock at a WAN hardware interface supporting the T3 link loses synchronization and ultimately causes degradation in performance of the applications. However, this problem is unlikely to be detected by the router DM. The reason is that the router domain can detect only IP header corruption. Intermittent noise is unlikely to cause many header corruptions, because of the small size of the header relative to the data.

Data corruption and packet loss are observable only by TCP connections layered over the IP link; thus, these symptoms are observed only in the enterprise domain. The enterprise DECS/DM will correlate symptoms of TCP connection problems to a router domain problem in the A-D link. Having subscribed to receive notification of this problem, the router backbone DECS/DM will be notified of the detection of a router backbone problem by the enterprise DECS/DM. The router DECS/DM can then use the problem event as a symptom and correlate it with other observable symptoms in its domain to isolate the clock problem.

The enterprise domain Modeler will contain instances of classes modeling the client and server applications, TCP and IP nodes and links underlying the application, the physical equipment in the LAN domains (workstations, hub, etc.), and an instance of a class representing the router backbone object. If we map these entities to the modeling framework, the client and server are special cases of *Logical Services*, the TCP and IP nodes are special cases of *Nodes*, the TCP and IP links are special cases of *Links*, the workstations are special cases of *Physical Services*, and the router backbone is modeled as a special case of *Link*. To illustrate the MODEL language, the specification (see box, this page) provides a schematic of some of the classes in the enterprise domain.

How would this situation be handled without using the DECS? An enterprise operator would notice a large number of problems with TCP connections. After ruling out other possible causes, the operator would probably run `traceroute` to check the A-D link. Since the problem we described is intermittent, it may not occur during `traceroute`. If it does, the operator will detect a problem in the A-D link and call the wide-area service provider to notify them. Service provider staff will then perform additional troubleshooting to isolate the cause of the problem. This process could take a long time, during which the database would not be accessible remotely. Automating this process via DECS is vastly superior to this slow and unreliable manual approach.

The Coding Approach to Event Correlation

In the previous sections, we introduced the Modeler and explained the information it provides for event correlation. In this section, we explain the event correlation algorithms.

Our approach to correlation is based on coding techniques [6]. The underlying idea of the coding approach is simple. Each problem causes many symptom events. Symptom events can include local events in the object where the problem originated and events propagated to other related objects. We treat the complete set of events caused by a problem as a "code" that identifies the problem. Correlation is then simply the process of "decoding" the set of observed symptoms by determining which problem has the observed symptoms as its code.

Event codes typically contain an enormous amount of redundancy, as typical systems tend to be overinstrumented. Since developers have no way to anticipate the troubleshooting processes operators may apply, they make almost every device parameter queryable. Thus, there exist routers with over 5000 MIB variables. Typically, the information content of

```
interface Client : LogicalService {
        event AbortedTransactions  #Aborted > AbortedThreshold;
        attribute long #Aborted;
        attribute long AbortedThreshold;
};
interface DBServer : LogicalService {
        event SlowResponse - AvgResponse > ResponseThreshold;
        attribute long AvgResponse;
        attribute long ResponseThreshold;
};
interface TCP_Node : Node {
        problem TCPPacketLoss - AbortedTransactions, SlowResponse;
        propagate AbortedTransactions = Client, Underlying;
        propagate SlowResponse = DBServer, Underlying;
};
interface IP_Node : Node {
        problem PacketsLoss - IPDiscardedPackets, TCPPacketsLoss;
        propagate TCPPacketsLoss   TCP_Node, Underlying;
        event IPDiscardedPackets = DiscardedPackets > DiscardedThreshold;
        attribute long DiscardedPackets;
        attribute long DiscardedThreshold;
};
interface Router Backbone : Link {
        problem Failure = PacketsLoss;
        propagate PacketsLoss = IP_Node, ConnectedTo;
        export Failure;
};
```

most variables is minimal, as many events are symptoms of multiple problems; they do not suffice to distinguish one problem from another.

The coding technique proceeds in two phases. In the *codebook selection* phase, a subset of events is selected for monitoring; the result of this process is called the *codebook*. The codebook is an optimal subset of events that must be monitored to distinguish the problems of interest from one another while ensuring the desired level of *noise tolerance*. In a networked system, events can be lost or delayed, and spurious alarms can be generated. It is essential that the correlation algorithms can correctly identify a problem, even in the case of "noise" in the event stream. In the *decoding* phase, the events in the codebook are monitored and analyzed in real time by decoding; that is, by finding the problems whose symptoms or "code" match the set of observed symptoms most closely.

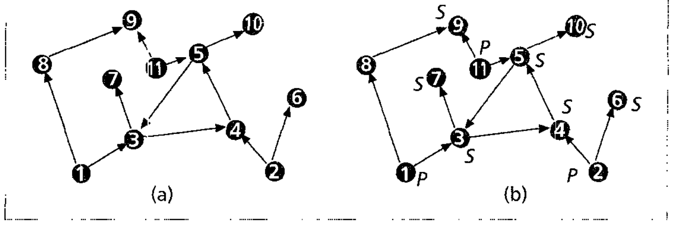
The coding approach reduces the complexity of real-time correlation analysis through preprocessing of the event knowledge model to optimize the number of events that must be monitored and analyzed, and by reducing correlation to a simple problem of minimal distance decoding.

The Mathematics of the Codebook Approach

Causality Graph Models — Correlation is concerned with analysis of causal relations among events. We use the notation $e \rightarrow f$ to denote causality of event f by event e. Causality is a partial order relation between events. The relation $\rightarrow$ can be described by a *causality graph* whose nodes represent events and whose directed edges represent causality. Figure 6a depicts a causality graph on a set of 11 events.

Nodes of a causality graph may be marked as problems (P) or symptoms (S), as in Fig. 6b. Note that some events may be neither problems nor symptoms (e.g., event 8), while other events may be both symptoms and problems.

The causality graph may include information that does not contribute to correlation analysis. For example, a cycle (such as events 3, 4, 5) represents causal equivalence. Therefore, a cycle of events can be aggregated into a single event. Similarly, certain symptoms are not directly caused by any problem (e.g., symptoms 7, 10) but only by other symptoms. They do not contribute any information about problems that is not already provided by these other symptoms. These *indirect symptoms* may be eliminated without loss of information. Henceforth, we will assume that a causality graph has been appropriately pruned.



■ Figure 6. *a) A causality graph; b) its labeling.*

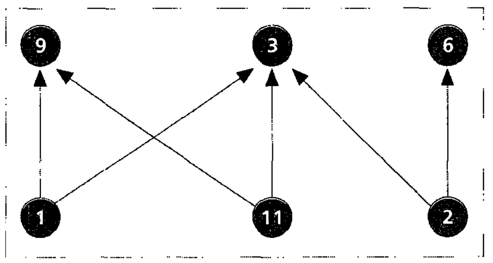

■ Figure 7. *A correlation graph.*

Figure 7 depicts the correlation graph corresponding to the causality graph of Figure 6 after pruning indirect symptoms and aggregating cycles.

Problems, Codes, and Correlation — In order to utilize coding techniques for correlation, the information contained in the correlation graph must be converted into a set of codes, one for each problem in the correlation graph. A code is simply a vector of 0s and 1s. We use the notation $\underline{p}$ to represent the *code* of the problem p. This conversion from correlation graph to a set of codes is best illustrated by example.

Consider the correlation graph of Fig. 7. The code for each problem contains a single bit corresponding to each symptom in the graph; since the graph contains three symptoms, each code will have a length of three. We arbitrarily order the symptoms so that the first bit of each code represents symptom 3, the second symptom 6, and the third symptom 9. Given this ordering, the code for problem 1 (denoted $\underline{1}$) is (1,0,1). A bit value of 1 in the code for problem 1 indicates that problem 1 causes the corresponding symptom; a bit value of 0 indicates that it does not cause the symptom. Thus, the meaning of the code (1, 0, 1) for problem one is that problem one causes symptom 3, does *not* cause symptom 6, and does cause symptom 9. It is easily seen that $\underline{2}$ is (1, 1, 0) and $\underline{11}$ is (1, 0, 1).

The event correlation problem is that of finding problems whose codes optimally match an observed symptom vector. Suppose that symptoms 3 and 9 have been observed. This may be described by an *event vector* $\underline{a}$ = (1, 0, 1). Since both $\underline{1}$ and $\underline{11}$ match the observation $\underline{a}$, one would infer that the two events are correlated with either problem 1 or 11. Note that these two problems have identical codes and are indistinguishable. Similarly, an event vector $\underline{a}$ = (1, 1 , 0) would match the code of problem 2. How should an event vector $\underline{a}$ = (0, 1, 0) be interpreted? One possibility is that this is just a spurious false alarm. Another possibility is that problem 2 occurred but symptom 3 was lost. The choice of interpretations depends on whether loss is more likely than spurious generation of events. There are, of course, other more remote possibilities.

Now, suppose that spurious or lost symptoms are unlikely. Then the information provided by symptom 9 is redundant. If only symptoms 3 and 6 are considered, the resulting codes $\underline{1}$ = $\underline{11}$ = (1, 0) and $\underline{2}$ = (1, 1) are sufficient to distinguish and correlate event vectors. Since real event correlation problems typically involve significant redundancy, the number of symptoms associated with a single problem may be very large. A much smaller set of symptoms can be selected to provide the desired level of distinction between problems. We call such a subset of symptoms a *codebook*. The complexity of correlation is a function of the number of symptoms in the codebook. An optimal codebook can thus reduce the complexity of correlation substantially.

Figure 8a depicts an example of 6 problems and 20 symptoms. The set of all problems and their respective codes, as represented in the figure, is termed the *correlation matrix*. Figure 8b depicts a codebook consisting of three symptoms {1, 2, 4}. This codebook distinguishes among all six problems. However, it can only guarantee distinction by a single symptom. For example, problems p_2 and p_3 are distinguished by symptom 4. A loss or spurious generation of this symptom will result in a potential decoding error. Distinction between problems is measured by the Hamming distance between their codes. The *radius* of a codebook is half the minimal Hamming distance among codes. When the radius is 0.5, the code provides distinction among problems but is not resilient to noise. To illustrate resiliency to noise, consider the codebook of Fig. 8c, where six symptoms are used to produce a codebook of

	P_1	P_2	P_3	P_4	P_5	P_6
1	1	0	0	1	0	1
2	1	1	1	1	0	0
3	1	1	0	1	0	0
4	1	0	1	0	1	0
5	1	0	1	1	1	0
6	1	1	1	0	0	1
7	1	0	1	0	0	0
8	1	0	0	1	1	1
9	0	1	0	0	1	1
10	0	1	1	1	0	0
11	0	0	0	1	1	0
12	0	1	0	1	0	0
13	0	1	0	1	1	1
14	0	0	0	0	0	1
15	0	0	1	0	1	1
16	0	1	1	0	0	1
17	0	1	0	1	1	0
18	0	1	1	1	0	0
19	0	1	1	0	1	0
20	0	0	0	0	1	1

	P_1	P_2	P_3	P_4	P_5	P_6
1	1	0	0	1	0	1
2	1	1	1	1	0	0
4	1	0	1	0	1	0

	P_1	P_2	P_3	P_4	P_5	P_6
1	1	0	0	1	0	1
3	1	1	0	1	0	0
4	1	0	1	0	1	0
6	1	1	1	0	0	1
9	0	1	0	0	1	1
18	0	1	1	1	0	0

■ **Figure 8.** *a) Correlation matrix; b) codebook of radius 0.5; c) codebook of radius 1.5.*

radius 1.5. This means that a loss or a spurious generation of any two symptoms can be detected, and any single-symptom error can be corrected.

We illustrate the error correction capabilities of the codebook of Fig. 8c. We will use a minimal-distance decoder, which means that we decode to the problem whose code has minimal Hamming distance from the event vector. Such a decoder will decode as p_1 all event vectors that contain a single-symptom perturbation of p_1. The event vectors {011100, 101100, 110100, 111000} will be decoded as p_1 with a single symptom loss, while {111110, 111101} will be interpreted as the occurrence of p_1 along with a single spurious symptom. The total number of events that can be generated due to a single symptom perturbation (loss or spurious one) in the 6 problem codes + the null problem $p_0 = 000000$ is 42. Therefore, a total of 48 event vectors (out of the possible 63) will be correctly decoded despite single-symptom observation errors. When two symptom errors occur, a minimal distance decoder can detect that errors have occurred but may not decode the event vector uniquely. It is easily seen that, in general, we can correct observation errors in $k - 1$ symptoms and detect k errors as long as k is less than or equal to the radius of the codebook.

Consider now the problem of model errors. For example, suppose problem p_4 in Fig. 8 can actually cause symptom 6 even though the model fails to reflect this. This will cause a single-symptom error with respect to the code of p_4. Symptom 6 will appear as a spurious symptom whenever p_4 occurs. In other words, *an error in the correlation model is equivalent to an observation error.* In contrast to random observation errors, model errors will appear as persistent observation noise. This persistence may be automatically detected by analyzing correlation logs and then used to correct the correlation model.

Correlation Performance Benchmarks

A model of a satellite-based communications network was used to benchmark the performance of DECS correlation. The modeled domain includes close to 4000 managed objects involving some 9500 problems and 6000 symptoms. Random scenarios were created by selecting random subsets of the model.

The benchmark model makes two conservative assumptions *unfavorable* to codebook correlation. It assumes an *underinstrumented* system where the number of observed symptoms is much smaller than the number of problems; typical systems are overinstrumented. It assumes a *sparse propagation model* where only a small number of symptoms are caused by a typical problem; in real-world systems with complex dependencies, problems tend to propagate very widely. Thus, typical real-world systems involve many more symptoms, which would yield smaller codebooks, a larger reduction in the number of symptoms to monitor, and faster correlation.

Figure 9a shows the effective event correlation rate measured in symptoms per second of actual elapsed time. The effective event correlation rate includes symptoms generated by a problem but not processed by the correlator because codebook reduction removed them from the codebook. In domains with fewer than 4000 problems, symptom processing was measured in thousands of symptoms per second. This is 2–4 orders of magnitude faster than the published figures of 0.25 events/s for ECXPERT [7] and 15 symptoms/s for IMPACT [8].

Another important aspect of the coding approach is its resilience to symptom loss. Figure 9b shows the correlation error rates when the probability of symptom loss ranges up to 20 percent. Even a substantial number of lost or spurious symptoms causes only minimal error probability, falling under 5 percent when the codebook radius exceeds 1.5.

Practical Advantages of the Codebook Approach

In addition to the speed and robustness advantages inherent to codebook technology, there are many practical advantages to using the codebook approach in DECS.

Domain-wide cross-object correlation rules are computed automatically, rather than having to be developed manually for each system. The codebook generation process takes as input the class libraries compiled from MODEL specifications, a specification of the domain's managed object instances and their topology, and the current set of problem notification subscriptions. It then computes the correlation matrix. Manually generating these correlation rules would be an enormous development effort.

Codebook reduction automatically determines whether there are sufficient symptoms to distinguish between the problems of interest. This knowledge is essential for a practical system; otherwise, processing may proceed indefinitely as the system attempts to narrow diagnosis to a single problem.

The codebook automatically adapts to topology changes affecting correlation rules. The codebook is updated whenever the set of relationships used to compute it changes, or the set of problem subscriptions changes. The ability to adapt dynamically to changes in system topology and configuration is a major advantage of the codebook technology. In low earth orbit (LEO) satellite networks, for example, link topology changes very rapidly. Even conventional terrestrial networks typically undergo several changes a day as equipment is added, moved, or changed, or software is added or upgraded.

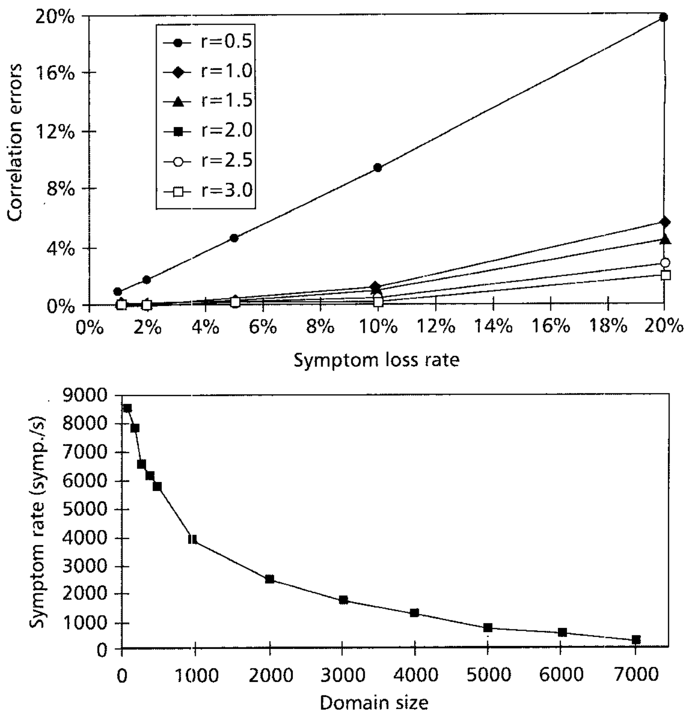
Putting It All Together

Previously, we showed briefly how to apply the DECS modeling technology to the example of Fig. 1. Developing models and their instrumentation is the only development step required in applying DECS to a domain. Also, once a class model has been developed, the same class specification can be reused in any system context where objects of that class appear. Thus, MODEL libraries can be developed that, over time, will eliminate most of the modeling efforts for networked systems using standard system, network, and application components. The Modeler also includes utilities to convert standard MIB specifications into the MODEL language to eliminate redundant modeling. The only other input needed by the system is up-to-date knowledge of the system topology.

The typical development life cycle of a DECS-based event management applications is as follows:

At Development Time — Event models of managed objects for which no models exist are written in the MODEL language. Libraries for collections of managed object classes are built by compiling MODEL specifications into C++. New class definitions can be added at any time and incorporated into a running DECS using Delegation technology [7].

DECS client application programs can also be developed. Examples of DECS clients include a display of detected problems, a trouble-ticketing application, or a policy-based problem handler that invokes predefined handlers associated with each problem. A simple DECS client is available as part of the basic system and can be specialized as needed.



■ **Figure 9.** *a) Symptom processing rate; b) correlation error rate.*

At Runtime — Codebooks are automatically generated by the system. The codebook generation process requires as input
- The relevant class libraries
- Specification of the current domain topology, including specification of all the managed object instances in the current system, the class to which each object belongs, and the values of its relationship attributes, used to build the Modeler's management information repository
- The current set of problem notifications to which clients of the DECS have subscribed

The correlation matrix is computed and then reduced in the codebook reduction process.

Once a codebook has been computed, the set of symptoms in the codebook is monitored in real time and matched against the current codebook using a minimal-distance decoder. When a code is matched, the DECS sends a notification to all clients that have subscribed to notification of this problem. Whenever the set of relationships used to compute the current codebook changes, or the set of problem subscriptions changes, the codebook is updated to reflect the current configuration.

Summary and Conclusions

Several other event correlation systems have been developed in both research and commercial contexts (see [8] for a comprehensive survey). The Distributed Event Correlation System takes a different approach from other systems by combining
- A high-level specification language extending CORBA IDL to specify event information *per managed object class*
- A runtime modeler of managed objects' data, relationships, and event information compiled from MODEL specifications
- A codebook correlation engine using observable events as problem codes to yield super-fast and noise-insensitive correlation
- A distributed architecture of cooperating domain managers for scalability to arbitrarily complex systems

An important advantage of our system is its ability to correlate events *across different managed objects*. Finite state machine (FSM)-based correlators are limited to correlating events in a single object. As such, they do not have the power to pinpoint the source of problems that occur in related objects.

Rule-based systems can perform cross-object correlation, but their performance is orders of magnitude slower than codebook decoding. DECS gains its performance advantage through its novel preprocessing of event information to produce the codebook, while rule-based systems do all of their searching during the correlation process. Rule-based systems are also limited in their scalability.

Both FSM-based and rule-based correlators are very sensitive to noise in the event stream. A lost or spuriously generated event will cause wrong state transitions or guide searches along incorrect paths. Neither approach is robust enough to address network environments where lost and false alarms are common.

The DECS coding approach is super-fast because the number of symptoms to monitor and analyze is substantially reduced, because searching is removed from the real-time correlation path, and because minimal-distance decoding is a very fast process. In addition, a side effect of codebook generation is the automatic determination of whether the system provides a sufficient set of observable symptoms to be able to isolate the problems of interest. Currently, determining which information to make accessible via MIB variables is completely ad hoc, resulting in enormous amounts of redundant information, with no guarantee that problems are distinguishable from one another.

The DECS use of programming language and compiler technology adds several advantages. Developing information models in a high-level specification language, and having a compiler generate the code to support them, is much faster and more reliable than manual development. Defining event information in an object-oriented topology-independent fashion provides enormous reuse advantages. Each model, once developed, can be reused in every context in which an object of this class appears.

The ability to *compute* the correlation rules enables our system to dynamically adapt to topology and configuration changes. This adaptability is important because it is not unusual for an enterprise network to undergo changes that affect its potential problems and their symptoms several times a day.

The distributed system architecture of the DECS enables it to scale to arbitrarily large and complex networks, and provides a natural organization for federated multi-domain networks.

Finally, the distributed system architecture of the DECS enables it to scale to arbitrarily large and complex networks, and provides a natural organization for federated multidomain networks.

The Distributed Event Correlation System is being applied in Motorola's IRIDIUM® system, a worldwide satellite-based communications system designed to permit voice, data, fax, and paging services. DECS is currently implemented on Sun workstations running Solaris 2.4. A port to processors running Wind River Systems VxWorks™ embedded real-time operating system is under development.

References

[1] Y. Yemini *et. al.*, "Semantic Modeling of Managed Information" *Proc. 2nd IEEE Workshop on Network Mngmt. and Control*, Tarrytown, NY, 1993.

[2] A. Dupuy *et al.*, "A Network Management Environment," *IEEE Network*, 1991.

[3] A. Dupuy *et al.*, "Design of the Netmate Network Management System," *Proc. 2nd Int'l. Symp. on Integrated Network Mngmt.*, K. Krishnan and W. Zimmer, eds., Apr. 1991.

[4] *MODEL Language Reference Manual*, System Management ARTS, Nov. 1994.

[5] "The Common Object Request Broker: Architecture and Specification," Object Management Group and Xopen 1992.

[6] S. Kliger *et al.*, "A Coding Approach to Event Correlation," *Proc. 4th Int'l. Symp. on Integrated Network Mngmt.*, Santa Barbara, CA, May 1995.

[7] Y. Nygate and L. Sterling, "ASPEN — Designing Complex Knowledge Based Systems," *Proc. 10th Israeli Symp. on Artificial Intelligence, Comp. Vision, and Neural Networks*, 1993, pp. 51–60.

[8] G. Jakobson, and H. Weissman, "Alarm Correlation," *IEEE Network*, vol. 7, no. 6 1993.

[9] Y. Yemini, G. Goldszmidt, and S. Yemini, "Network Management by Delegation," *Proc. 2nd Int'l. Symp. on Integrated Network Mngmt.*, K. Krishnan and W. Zimmer, Apr. 1991.

[10] D.Ohsie and S. Kliger, "Network Event Management Survey," SMARTS Tech. Rep., 1993.

Additional Reading

[1] A. Dupuy *et al.*, "Network Fault Management: A User's View," *Proc. IFIP Symp. on Integrated Network Mngmt.*, Amsterdam: North Holland, 1989.

[2] L. Feldkuhn, and J. Erickson, "Event Management as a Common Functional Area of Open Systems Management," *Proc. IFIP Symp. on Integrated Network Mngmt.*, Amsterdam: North Holland, 1989.

[3] G. Jacobson, R. Weihmayer, and M. Weissman, "A Dedicated Expert Shell for Telecommunication Network Alarm Correlation," *Proc. 2nd IEEE Network Mngmt. anc Control Workshop*, 1993, pp. 277–88.

[4] S. Kliger, Y. Yemini, and S. Yemini, "Apparatus and Method for Event Correlation and Problem Reporting," Patent Application, 1994.

[5] S. Kliger *et al.*, "Decs Performance Benchmarks Summary," SMARTS Tech. Rep. 10-31-94, 1994b.

[6] A. Langsford, and J. Moffett, *Distributed Systems Management*, Reading, MA: Addison Wesley, 1993.

[7] A. Leinwand and K. Fang, *Network Management : A Practical Perspective*, Reading, MA: Addison Wesley, 1993.

[8] L. Lewis, "A Case Base Reasoning Approach to the Resolution of Faults in Communications Networks," *Proc. 3rd Int'l. Symp. on Integrated Network Mngmt.*, 1993.

[9] S. Roman, *Coding and Information Theory*, New York: Springer Verlag, 1992.

[10] M. Rose, *The Simple Book: An Introduction to Internet Management*, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1994.

[11] Fault Isolation Information Standards Group, "Report of the X3 Standards Planning and Requirements Committee (SPARC) in Computer Business Equipment Manufacturers Association," FIISG-91-020-RD, 1992.

[12] W. Stallings, *SNMP, SNMPv2, and CMIP The Practical Guide to Network- Management Standards*, Reading, MA: Addison Wesley, 1993.

[13] M. T. Sutter and P. E. Zeldin, "Designing Expert Systems for Real-Time Diagnosis of Self-Correcting Networks," *IEEE Network*, Sept. 1988, pp. 43–51.

Biography

SHAULA ALEXANDER YEMINI is president and co-founder of SMARTS. Prior to founding SMARTS, Dr. Yemini was the senior manager for distributed systems software technology at IBM Research in Yorktown Heights, New York. Her work includes the co-invention of Optimistic Recovery, a technique for transparent fault tolerance in distributed systems which won an IBM Outstanding Innovation Award. Dr. Yemini has a Ph.D. in computer science from UCLA.

SHMUEL KLIGER is vice president, Advanced Technology at SMARTS, and leads the Distributed Event Correlation System project there. His research experience includes designing and implementing distributed concurrent logic programming languages and environments. Dr. Kliger has a Ph.D. in computer science from the Weizmann Institute of Science.

EYAL MOZES is the major developer of the MODEL language and its compiler. Dr. Mozes has a Ph.D. in computer science from Stanford University.

YECHIAM YEMINI is the director of the Distributed Computing and Communications Laboratory at Columbia University and a co-founder of SMARTS. His interests include broad areas of distributed and networked systems technologies; he has published over 100 articles and edited three books in these areas. Prof. Yemini has a Ph.D. in computer science from UCLA.

DAVID OHSIE is a Ph.D. candidate at Columbia University, where he is currently pursuing his thesis research in causal analysis.