Automatically Acquiring Rules for Event Correlation From Event Logs

Tim Oates, David Jensen, and Paul R. Cohen

Computer Science Technical Report 97-14

Experimental Knowledge Systems Laboratory
Computer Science Department, Box 34610
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003-4610

Abstract

A single fault in a complex network can generate a cascade of events, potentially overloading a manager's console with information. One way to reduce the number of events is event correlation, a process that groups several related events into a single composite event. We have developed Multi-Event Dependency Detection (MEDD), an algorithm that automatically constructs event correlation rules. MEDD is a prototype for a component of NASA's EOSDIS Core System. The algorithm efficiently searches the space of possible dependencies between events and selects the most useful rules. Preliminary results indicate that MEDD identifies useful rules for event correlation, and could reduce the information burden on network managers.

This research was supported by DARPA under contract F30602-91-C-0076 and DoD Fellowship F30602-93-C-0100.

1 Multi-Event Dependency Detection

MEDD finds *dependencies* between patterns of network events recorded in event logs.[1] Dependencies are unexpectedly frequent or infrequent co-occurrences of patterns of events, and can be expressed as rules of the following form: "If an instance of event pattern x is recorded in the log at time t, then an instance of event pattern y will be recorded between $t - \delta/2$ and $t + \delta/2$ with probability p." Dependency rules are denoted $x \Rightarrow y$; x is called the *precursor* pattern, and y is called the *successor* pattern. A dependency is strong if the empirically determined value of p (obtained by counting actual co-occurrences of x and y in historical event logs) is very different from the probability of seeing a co-occurrence of x and y under the assumption that they are independent. Strong dependencies capture structure in event logs because they tell us that there is a relationship between their constituent event patterns, that occurrences of those patterns are not independent.

Performing fault correlation with MEDD is a two-step process. First, MEDD is used off-line to find strong dependencies between patterns of events in existing event logs. Second, the resulting rules are matched against new events as they are generated in real time. Because strong dependencies indicate that occurrences of precursors and successors are not independent, any co-occurrence of the constituent event patterns of a rule can be reported to the network manager as a single meta-event, thereby reducing the volume of events that reach the manager's console. That is, all of the events in the precursor and all of the events in the successor can be collapsed into a single unit.

MEDD finds strong dependencies between patterns of events by performing a general-to-specific, best-first, systematic search over the space of all possible pairs of event patterns. In the remainder of this section, we explain exactly what that means. Section 1.1 defines the space of pairs of event patterns that MEDD searches. Section 1.2 discusses systematic search in general, and Section 1.3 describes the details of systematic search in MEDD. Finally, Section 1.4 describes a post-processing phase that the rules returned by MEDD undergo, and explains in more detail how the rules are used for fault correlation in real time.

1.1 The Space of all Possible Event Patterns

Precursor and successor event patterns contain one or more *partially instantiated events* (PIEs). In general, events recorded in logs comprise multiple fields, and each field takes a value from a set of allowable values specific to that field. For example, the `status` field might take values from the set {`up, down`}. Assuming that events contain f fields, and that field i takes values from the set $\mathcal{V}_i$, then the space of all possible events is given by $\mathcal{E} = \times_{i=\infty}^{\{} \mathcal{V}_i$ (that is, the cross product of all of the $\mathcal{V}_i$ – every possible combination of field values). Any event e that appears in an event log is an element of $\mathcal{E}$. PIEs simply leave the value of one or more fields unspecified, which is denoted by assigned those fields the wildcard value *. Therefore, the space of all possible PIEs is given by $\mathcal{P} = \times_{i=\infty}^{\{} (\mathcal{V}_i \cup \{*\})$. Note that $\mathcal{E} \subset \mathcal{P}$. Consider a simple event structure containing two fields – `status` and `element` – such that

[1] MEDD is based on our earlier work with a similar algorithm named MSDD (Oates & Cohen 1996; Oates, Schmill, & Cohen 1996; Oates *et al.* 1995).

$\mathcal{V}_{\int\sqcup\dashv\sqcup\sqcap\int} = \{\texttt{up, down}\}$ and $\mathcal{V}_{\rceil\updownarrow\rceil\Updownarrow\rceil\backslash\sqcup} = \{\texttt{host, router}\}$. Then $\mathcal{E}$ and $\mathcal{P}$ are as follows:

$$\mathcal{E} = \left\{ \begin{array}{ll} \texttt{(up host)} & \texttt{(up router)} \\ \texttt{(down host)} & \texttt{(down router)} \end{array} \right\}$$

$$\mathcal{P} = \left\{ \begin{array}{lll} \texttt{(up host)} & \texttt{(up router)} & \texttt{(up *)} \\ \texttt{(down host)} & \texttt{(down router)} & \texttt{(down *)} \\ \texttt{(* host)} & \texttt{(* router)} & \texttt{(* *)} \end{array} \right\}$$

A PIE $p \in \mathcal{P}$ is said to *match* an event $e \in \mathcal{E}$ if every non-wildcard field in p has the same value as the corresponding field in e. For example, the PIE $p =$ (`up *`) matches event $e_1 =$ (`up router`), but it does not match event $e_2 =$ (`down host`) or event $e_3 =$ (`down router`). Event patterns, precursors and successors, are defined to be sets of PIEs; i.e. $x = \{p_1, \ldots, p_n | p_i \in \mathcal{P}\}$ is an event pattern. Precursors and successors are said to match a fragment of an event log if each of their constituent PIEs can be matched on a *different event* in the fragment. Therefore, the event pattern {`(* router), (* host)`} matches the following event log fragment containing three events, whereas the event pattern {`(down *), (* router)`} does not:

```
up   host
up   host
down router
```

1.2 Systematic Search

MEDD's search for dependencies among events is *systematic*, leading to search efficiency. Systematic search non-redundantly enumerates the elements of search spaces for which the value or semantics of any given node are independent of the path from the root to that node. Webb calls such search spaces *unordered* (Webb 1996). Consider the space of disjunctive concepts over the set of literals $\{A, B, C\}$. Given a root node containing the empty disjunct, $false$, and a set of search operators that add a single literal to a node's concept, a *non*-systematic elaboration of the search space is shown in Figure 1. Note that the concept $A \vee B \vee C$ appears six times, with each occurrence being semantically the same as the other five, yet syntactically distinct. In the space of disjunctive concepts, the semantics of any node's concept is unaffected by the path taken from the root to that node. For example, the two paths below yield semantically identical leaf nodes:

$$false \rightarrow A \rightarrow A \vee B \rightarrow A \vee B \vee C$$
$$false \rightarrow C \rightarrow C \vee B \rightarrow C \vee B \vee A$$

The search tree in Figure 1 contains six syntactic variants of the concept $A \vee B \vee C$, and two syntactic variants of the concepts $A \vee B$, $A \vee C$ and $B \vee C$. Clearly, naive expansion of nodes in unordered search spaces leads to redundant generation and wasted computation.

Systematic search of unordered spaces generates no more than one syntactic form of each semantically distinct concept, and is therefore much more efficient than naive search. That is accomplished by imposing an order on the search operators used to generate the children

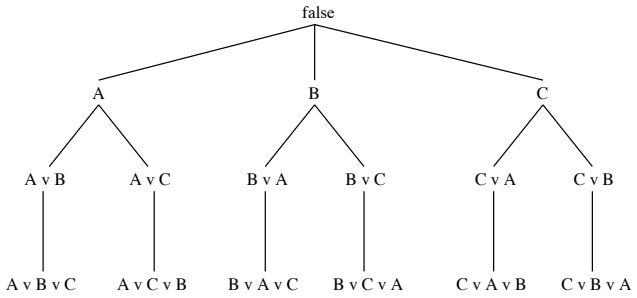

Figure 1: A naive elaboration of the space of disjunctive concepts over the set $\{A, B, C\}$ generated by applying all valid search operators at every node. Naive search generates multiple syntactic variants of individual concepts.

of a node, and applying only those operators at a node that are higher in the ordering than all other operators already applied along the path to the node. Let op_A, op_B and op_C be the operators that add the literals A, B and C respectively to a node's concept. If we order those operators so that $op_A < op_B < op_C$, then the corresponding space of disjunctive concepts can be enumerated systematically as shown in Figure 2. Note that each semantically distinct concept appears exactly once. The concept A is obtained by applying operator op_A to the root node. Because $op_B > op_A$ and $op_C > op_A$, both op_B and op_C can be applied to the concept A, generating the child concepts $A \vee B$ and $A \vee C$. In contrast, the concept C, which is obtained by applying op_C to the root node, has no children. Because all other operators (op_A and op_B) are lower in the ordering than op_C, none will be applied and no children will be generated.

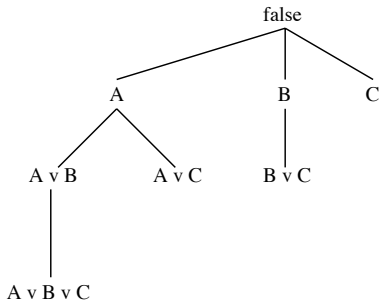

Figure 2: Systematic elaboration of the space of disjunctive concepts over the set $\{A, B, C\}$. Only one syntactic form of each semantically distinct concept is expanded.

The trees in Figure 1 and Figure 2 represent general-to-specific elaborations of the space of disjunctive concepts. That is, the root node is the most general concept *false*, and children of a node are generated by making that node's concept more specific through the addition of a single literal. All of the concepts at depth d contain exactly d literals.

1.3 Search in MEDD

MEDD accepts as input a set of historical event logs, searches for dependencies between patterns of events in those logs until a user defined limit on the number of nodes to expand

is reached, and returns a list of the nodes explored. This section describes the search in detail, and the next explains how the returned nodes are processed and used for event correlation.

MEDD's traversal of the space of dependencies between event patterns in both general-to-specific and systematic. Each node in the search space corresponds to a dependency rule, and the root of that space is the completely general rule in which both the precursor and successor contain only wildcards. For the simple two-field event structure introduced in Section 1.1, the root node contains the rule {(* *)} ⇒ {(* *)}. The children of a node are generated by modifying either the precursor or successor of that node in one of two ways: by filling in the value of a field that contains a wildcard in an existing PIE, or by adding a new PIE containing a single non-wildcard field. In either case, the descendants of a node are always more specific – they specify more non-wildcard values for fields – than the original node.

Consider the node {(up *)} ⇒ {(* *)}. Several of the children of that node are shown below:

```
{(up host)} => {(* *)}          {(up *), (down *)}   => {(* *)}
{(up *)}    => {(down *)}       {(up *), (* host)}   => {(* *)}
{(up *)}    => {(* router)}     {(up *), (* router)} => {(* *)}
```

The three children in the left column were generated by specifying a value for a single field that was wildcarded in the parent. The three children in the right column were generated by adding a new PIE containing a single non-wildcard to the precursor.

The search is made systematic, thereby obtaining the efficiency gains described in Section 1.2, by only adding non-wildcards and PIEs to the right of the right-most non-wildcard in a node when generating that node's children. Consider the node {(up *)} ⇒ {(down *)}. The right-most non-wildcard in this rule is down in the successor. Therefore, {(up *)} ⇒ {(down router)} is a valid child, but {(up router)} ⇒ {(down *)} would not be generated because it requires adding a non-wildcard to the left of down. (The interested reader is referred to (Oates & Cohen 1996) for a detailed discussion of the use of this type of operator ordering to achieve systematicity.)

MEDD's search through the space of dependencies is guided by a best-first heuristic. Each time a node is generated, MEDD scans its historical event logs, counting the number of times the precursor and successor of that node co-occurred within a temporal window of size δ (specified by the user). Frequency of co-occurrence becomes the node's heuristic value, biasing the search to prefer rules with frequently occurring precursors and precursor/successor pairs that frequently co-occur. The search proceeds by iteratively selecting the node with the highest value, generating that node's children, and adding them to the list of nodes under consideration.

Counting co-occurrences of precursors and successors is not as straightforward as it might appear. Recall that an event pattern is said to match a fragment of a log (i.e. occur in that fragment) if each of the PIEs in the pattern can be matched on a different event. That is, we are only concerned with whether a set of PIEs co-occur within some pre-specified temporal window, not whether they occur in any specific order. Log fragments are obtained for matching by finding all sets of events within the logs that span no more than δ time steps. For a log fragment with n events and a pattern with k pies, there are n choose k ways that the pattern might match. If the precursor matches a fragment, then successor must also

be checked for a match. That process is complicated by the fact that the successor may not match because certain events are "taken" by the precursor, but the precursor could match on a different set of events allowing the successor to match. In the worst case, all possible matches of the precursor must be tried to find a match for the successor. We avoid this combinatorial problem by simply trying to match the elements of the precursor sequentially and, if a complete match is found, doing the same with the successor. This scheme may miss some matches, but it is computationally efficient. We are currently investigating ways to efficiently implement more accurate counting algorithms.

1.4 Rule Post-Processing and Application

MEDD returns all of the nodes that it explores in the space of dependencies. To find the strongest dependencies among those explored, a 2x2 contingency table that describes the frequency of co-occurrence of each rule's precursor and successor is built. (Actually, the complete table is built during the search as each node is expanded. The first cell of the table is used as the node's heuristic value to guide the search.) Then, the G statistic, a statistical measure of non-independence, is computed for each rule, and the rules are sorted in non-decreasing order of G. We then remove generalizations of the strongest rules that were generated as the search descended through the tree to find those rules. Finally, the top k rules are retained. Currently, the choice of k is ad hoc. We are investigating automated methods for choosing "good" values for k.

To perform fault correlation in real time, the remaining rules are matched against incoming events in sorted order. That is, the rule representing the strongest dependency that matches a new set of events is used to cluster those events.

2 Experiments

This section describe an experiment designed to test both MEDD's ability to find strong dependencies between patterns of events and the utility of those dependencies with respect to event correlation. Event logs were generated by a modified version of the Netsim network simulator which is publicly available from MIT (Heybey & Robertson 1994). Netsim is a discrete event simulator that models a network of components that communicate via the TCP/IP protocol. The simulator implements hosts, ethernet links, switches, point-to-point links (e.g. modems), TCP interfaces (both with and without the slow start protocol), poisson traffic sources, and uniform traffic sources.

The simulator was modified to allow randomly selected components to fail. The interval between failures and the duration of failures are drawn from distributions with user specified means and standard deviations. When a component fails, all network packets that it receives, or attempts to send, are dropped. Some failures may be reported accurately, such as a host reporting that a local application has failed. Other failures, such as switches going down, may only be detectable indirectly through connection timeouts. In addition, a monitor component was added to the simulator to act as an SNMP proxy agent. That component periodically polls the other network components and reports on their reachability.

Netsim generated two separate event logs for a simulated network containing 17 components. Each event contained six fields: the ID of the component reporting the event, that component's type, the time at which the event was reported, the event type, and two additional fields whose semantics and contents depend on the event type. The first event log, which covered 261 network errors and 237 reported events, was used by MEDD to search for dependency rules. Not all errors generated events, and some errors generated multiple events. The latter errors are the ones that MEDD attempts to correlate. The following is a sample fragment of the event log:

```
monitor MONITOR 27200000 PROXY tcp1b TCP-INTERFACE
h3 HOST 27663290 ERROR app1b APPLICATION
app2a APPLICATION 28571516 BAD_CONN btcp2b BTCP-INTERFACE
psource POISSON-SOURCE 28571860 BAD_CONN psink SINK
monitor MONITOR 28600000 PROXY et ETHERNET
app2b APPLICATION 28671598 BAD_CONN btcp2a BTCP-INTERFACE
psource POISSON-SOURCE 28672653 BAD_CONN psink SINK
```

The second event log, which covered 248 network errors and 243 reported events, was used as a source of new events to test the rules generated from the previous log. The rules were used to find and report correlated events.

MEDD generated 30,000 search nodes, and the total CPU time required by the search and post-processing was 344 seconds.

References

Heybey, A., and Robertson, N. 1994. The network simulator version 3.1.

Oates, T., and Cohen, P. R. 1996. Searching for structure in multiple streams of data. In *Proceedings of the Thirteenth International Conference on Machine Learning*, 346 – 354.

Oates, T.; Schmill, M. D.; Gregory, D. E.; and Cohen, P. R. 1995. Detecting complex dependencies in categorical data. In Fisher, D., and Lenz, H., eds., *Finding Structure in Data: Artificial Intelligence and Statistics V*. Springer Verlag. 185 – 195. Includes work on an incremental algorithm not contained in workshop version.

Oates, T.; Schmill, M. D.; and Cohen, P. R. 1996. Parallel and distributed search for structure in multivariate time series. Technical Report 96-23, University of Massachusetts at Amherst, Computer Science Department. Long version of conference paper with same title.

Webb, G. I. 1996. OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research* 3:45–83.