Fast Convex Closure for Efficient Predicate Detection

Paul A.S. Ward and Dwight S. Bedassé*

Shoshin Distributed Systems Group
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
{pasward,dsbedass}@shoshin.uwaterloo.ca

Abstract. The behaviour of parallel and distributed programs can be modeled as the occurrence of events and their interrelationship. Event data collected according to the event model is stored within a partial-order data structure, where it can be reasoned about, enabling debugging, program steering, and autonomic feedback control of the application. Reasoning over event data, a critical requirement for autonomic computing, is typically in the form of predicate detection, a search mechanism able to detect and locate arbitrary predicates within the event data.
To enable hierarchical predicate detection, compound events are formed by computing the convex closure of the matching primitive events. In particular, the Xie and Taylor convex-closure algorithm forms the basis for such an approach to predicate detection. Unfortunately, their algorithm can be quite slow, especially for hierarchical compound events.
In this paper, we study the cause of the problems in the Xie and Taylor algorithm. We then develop an efficient extension to their algorithm, based on a simple caching scheme. We prove our algorithm correct. We also provide experimental results that demonstrate that our approach reduces the execution time of the Xie and Taylor algorithm by up to 98 percent.
Keywords: Autonomic computing, program steering, predicate detection tool.

1 Motivation

The architecture of tools for monitoring and debugging message-passing parallel programs, enabling parallel-program steering, and the autonomic observation and control of enterprise and distributed systems is broadly similar, and can be characterized as shown in Fig.1. A variety of such tools have been built over the years, including ATEMPT [16, 17], Object-Level Trace [13], POET [20], POTA [23], and Log and Trace Analyzer [12]. The managed system is instrumented with monitoring code that captures significant event data. Ideally, the information collected will include the event's process and thread identifiers, number, and type, as well as partner-event identification, if any. This event data is forwarded from each process to a central monitoring entity which, using this information, incrementally builds and maintains a data structure of the partial order of events that form the computation [21]. That data structure may be queried by a variety of systems, the most common being visualization engines for debugging and steering and, more recently, control entities for autonomic computing [15].

* The authors would like to thank IBM for supporting this work.

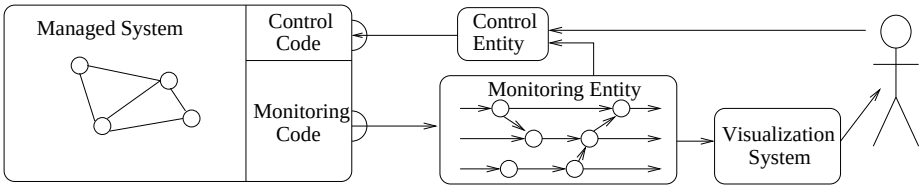

Fig. 1. Monitoring and Control Architecture

The querying of the partial-order data structure for predicate detection has the intent of either displaying predicates of interest to the user, or feeding the information directly into the controller. Rapid analysis of event data is critical for both of these uses. While there have been several approaches to predicate detection (*e.g.*, [4, 14, 22]), this paper focuses on hierarchical predicate detection based on compound events [1]. The current-best algorithm that employs this technique was developed by Xie and Taylor [24] and is implemented within the eclipse system [5].

In using the Xie and Taylor system we discovered it to be very slow in a non-trivial number of cases. Specifically, queries could take several hours to execute. In this paper, we describe a series of experiments that we performed to determine the cause of the slowness in the Xie and Taylor algorithm. As a result of our analysis, we developed a novel incremental closure algorithm that improved the performance of the predicate-detection algorithm by up to 98%.

The remainder of this paper is organized as follows. We first briefly review the operation of the Xie and Taylor algorithm, describing the basics of hierarchical predicate detection based on compound events. We discuss related work. In Sect.3 we detail in three steps the problem with the Xie and Taylor algorithm, the theoretical basis for incremental closure calculations, and finally our algorithm that solves the problem. We then provide both a theoretical and experimental analysis of our approach in Sect.3.1. We discuss related work in Sect.5, contrasting it with our approach. We conclude by observing what we have achieved and what issues remain open.

2 Fundamentals

We now describe the basics of hierarchical predicate detection based on compound events, and the Xie and Taylor approach specifically. We first briefly review the fundamentals of modeling systems as partial orders, which forms the basis of this work.

The event-based approach to modeling multi-threaded, parallel, and distributed systems abstracts computations into sequential processes[1] each of which is a sequence of four types of events: *transmit*, *receive*, *unary*, and *synchronous*. These events are considered to be atomic. Further, they form the primitive events of the computation.

The Lamport "happened before" relation [21] is then defined as the smallest transitive relation satisfying

[1] Throughout this paper we will use the term "process" to indicate any sequential entity. It might be a single-threaded process, a thread, a semaphore, an EJB (in the case of Object-Level Trace), a TCP stream, *etc.*

1. $e^i_{p_1} \preceq e^j_{p_2}$ if $e^i_{p_1}$ occurs before $e^j_{p_2}$ on the same process (*i.e.* $p_1 = p_2$ and $i \leq j$)
2. $e^i_{p_1} \preceq e^j_{p_2}$ if $e^i_{p_1}$ is a send event and $e^j_{p_2}$ is the corresponding receive event

This relation, together with the events, forms the partial order of the computation. Events are concurrent if they are not in the "happened before" relation.

$$e^i_{p_1} \parallel e^j_{p_2} \iff e^i_{p_1} \not\preceq e^j_{p_2} \wedge e^j_{p_2} \not\preceq e^i_{p_1} \tag{1}$$

Given a partial order of computation, there are two types of patterns that are typically sought. First we may seek patterns within the structure of the partial order. For example, we may wish to look for the pattern:

$$e^i_{p_1} \preceq e^j_{p_2} \wedge e^k_{p_3} \preceq e^j_{p_2} \wedge e^i_{p_1} \parallel e^k_{p_3} \; ; \; p_1 \neq p_2 \neq p_3 \tag{2}$$

This particular pattern is a crude form of race detection. We are seeking events in processes p_1 and p_3 that both precede an event in a third process p_2 but that have no synchronization between them. The events thus form a potential race condition.

This form of structural pattern searching is equivalent to directed-subgraph isomorphism. Specifically, it is equivalent to asking if the directed acyclic graph that represents the partial order of the computation contains a subgraph isomorphic to the directed graph that represents the pattern being sought. The directed graphs in this equivalence can be either the transitive reductions or the transitive closures of the respective partial orders. This problem is known to be NP-complete [8].

The second type of pattern that we may seek is a pattern within a consistent global state. There are several varieties that may be sought, such as stable predicates (once the predicate is true, it remains true), definite predicates (the predicate is true on all possible paths in the lattice), possible predicates (the predicate is true on some paths in the lattice), and so forth. From the perspective of a partial-order data structure, the primary concern is the ability to determine what is, or is not, a consistent global state. This in turn means we need the ability to determine structural patterns that are consistent global states. It is, as with the first type, NP-complete in the general case. This paper focuses solely on the problem of determining structural patterns within the partial order.

2.1 Hierarchical Predicate Detection Based on Compound Events

To alleviate the problem of NP-completeness, and to reduce the complexity of patterns, the approach taken is to seek hierarchical predicates based on compound events. In this approach, whenever a sub-pattern is matched, the events that form it are closed (according to a criteria to be described below) into a compound event. The requirements of such a compound event is that it must possess (most of) the properties of a primitive event. Specifically, given a compound event and any other event (primitive or compound), it must be possible to determine the precedence relationship between the two. The most effective way currently known of ensuring these requirements is that the compound event be convex closed [19], defined as follows:

Definition 1 (Convex Event). *An compound event c is convex if and only if*

$$\forall_{e_i, e_j \in c} \exists_{e_l} e_i \preceq e_l \wedge e_l \preceq e_j \Rightarrow e_l \in c$$

and extending the definition of precedence to compound events to:

$$c_i \preceq c_j \iff \exists_{e_i \in c_i; e_j \in c_j} e_i \preceq e_j$$

Note that the definition does not result in cyclic precedence provided the compound events are convex. Note also that a primitive event can be compared with a compound event by considering it to be a compound event with a single constituent element.

We now motivate this approach with a simple example. Consider seeking four events, e_1, e_2, e_3, and e_4 such that $e_1 \preceq e_2$, $e_3 \preceq e_4$, and yet ensuring that e_1 and e_2 are each concurrent with both e_3 and e_4. Given this requirement, a non-compound-event-based approach would require the pattern sought to be:

$$(e_1 \preceq e_2) \wedge (e_3 \preceq e_4) \wedge (e_1 \parallel e_3) \wedge (e_1 \parallel e_4) \wedge (e_2 \parallel e_3) \wedge (e_2 \parallel e_4)$$

By contrast, the compound-event-based approach seeks the pattern

$$(e_1 \preceq e_2) \parallel (e_3 \preceq e_4)$$

Note that while the compound-event-based approach does require two convex closure operations, it requires only four precedence tests, while the alternate approach requires ten.[2] Further, observe that as predicate complexity increases, the advantage of the compound-event-based approach increases. Finally, note that in this case the matching events will be identical, regardless the method chosen. While this is not always true, we have found that it is not difficult to prune unwanted matches from the system.

2.2 The Xie and Taylor Algorithm

Given the problem of structural predicate detection, Xie and Taylor developed a straightforward naive-backtracking algorithm. A parse tree is created of the pattern sought. This tree is processed in prefix order. Whenever the parse-tree node that is matched is a precedence-relationship node, the convex closure is computed, creating a compound event at that point in the parse tree. This is treated as a matched event. This process continues until either the desired pattern is found, or there is no matching event, in which case the algorithm backtracks, matching a different event.

The key features of their algorithm are their pruning rules, necessary to limit the search space, and their convex-closure algorithm. We do not modify their pruning rules, and thus will not comment on them further other than to note that our approach is orthogonal to their pruning rules. Any revisions to the pruning rules may affect the performance of the algorithm, but will not affect the correctness of the overall system.

The critical aspect of their approach, from the perspective of this paper, is their convex-closure algorithm. This algorithm takes an input event set of primitive events, and returns as output two sets, **front** and **back**, that represent the front and back of the

[2] While it may seem that the precedence test cost is higher for compound events, this is not in fact the case. It is possible to assign a vector timestamp to a convex event in much the same manner as one is assigned to a primitive event, enabling precedence determination between convex events to be as efficient as it is with primitive events [18].

convex event set, respectively. For a given convex event C, $e \in$ **front**(C) if-and-only-if $\nexists_{e'}\ e' \prec_p e$, where $e' \prec_p e$ if events e and e' are in the same process p and event e' precedes event e. Back is defined analogously:

$$e \in \textbf{back}(C) \iff \nexists_{e'}\ e \prec_p e' \tag{3}$$

Thus, in the worse case the convex event covers all processes in the computation, and thus **front** and **back** will have size N, where N is the number of processes. In such a case, the computational complexity of their algorithm is $O(N^3)$. The full technical details of their algorithm are available in their paper [24]. From the perspective of our work, it is a black box. The primary detail specifically required in our work is that in their algorithm the input event set is composed of two (possibly compound) events. The usage of the convex-closure algorithm by their predicate-detection mechanism is such that one of the these input events is held constant, while the other is varied. The significance of this will become apparent in Sect. 3.3.

3 Incremental Predicate Detection Algorithm

As we have already observed, when using the Xie and Taylor algorithm we found it to be slow, to the point that in a non-trivial number of cases its execution time was measured in hours. We therefore set about first determining the cause of the slowness in their algorithm. Having done so, we developed a theoretically-sound solution to the problem, and then created an algorithm based on it. We now describe these three steps in detail.

3.1 Analysis of Existing Approach

To determine the cause of inefficiency in the Xie and Taylor algorithm we performed a series of experiments using a variety of predicates and data sets. In these experiments, we instrumented the Xie and Taylor code to determine how many convex closures were performed, what the input and output sets were for the given closure, and the execution time to perform the closure in question.

In analyzing the data from these experiments we discovered it was very rare for a convex closure to consist of an entirely new set of input events. Rather, in more than 90% of cases, only one of the events changed. We further discovered that in cases where one input event changes, it was typically a near successor of the input event of the prior closure. However, the Xie and Taylor algorithm made no use of this fact. Rather, it would simply recompute the closure from scratch.

This problem is best illustrated by example. Consider the set of events shown in Fig.2. The pattern $q \preceq (a \preceq d)$ is being sought, and events q and a have already been matched. All that remains is to match d, compute a convex closure between that and the matched a, and confirm that this is a successor to the matched q. If d is matched to d_1 then the convex closure $C1$ of a and d_1 is computed. Unfortunately, $C1$ is concurrent to q. As a result, the search backtracks and matches d to d_2. The compound event $C2$ is then computed as the convex closure of events a and d_2. This is found to be a successor

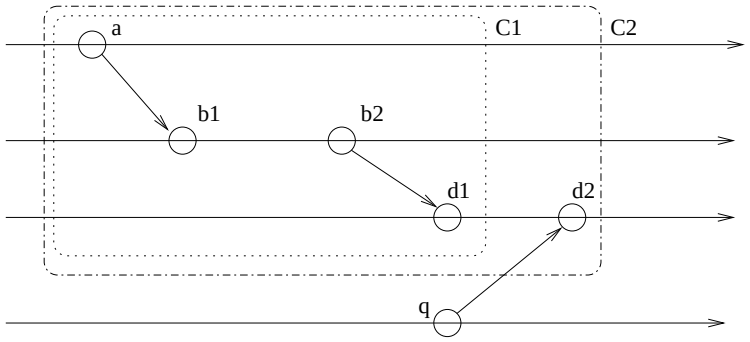

Fig. 2. Incremental Closure Computation

to q, and thus the desired predicate is found. Note that $C2$ is computed without regard to the original computation of $C1$. By experimental analysis, this lack of incremental computation was found to be the major cause of inefficiency in the Xie and Taylor algorithm.

3.2 Theoretical Basis for Improvement

Having found the problem, it was necessary to determine if recomputing the closure from scratch was an inherent requirement of convex events, or if it was possible to incrementally compute such closures. Thus, considering the example of Fig.2, we wished to compute $C2$ given $C1$.

In this regard, we discovered the following theorems. To understand these theorems, we first define the following functions.

Definition 2 (Convex Closure). *$CC(E)$ is the convex closure of event set E*

Definition 3 (Location Set). *$\mathbf{l}E$ is the set of processes in which the various events of event set E occur.*

Given these definitions, we were able to prove the following theorem.

Theorem 1 (Incrementality Theorem).

$$(\mathbf{l}CC(E \cup \{e\}) = \mathbf{l}CC(E)) \wedge (CC(E) \preceq e) \Longrightarrow$$
$$CC(E \cup \{e\}) = CC(E) \cup CC(\mathbf{back}(CC(E)) \cup \{e\})$$

Proof: See [2] □

This theorem states that, as long as the location set does not change, the convex closure of an event set E together with a succeeding event e will be the union of the convex closure of the original set, together with that of the closure of e and the back of the convex closure of the original set. What this means in practice is that if the convex closure of the event set E has already been computed, then only a small addition closure needs to be computed. It is fairly trivial to show that the **front** set will remain the same as that of the closure of E, while the **back** set will be that of the closure of $\mathbf{back}(CC(E))$ together with e.

3.3 Algorithm

Given Theorem 1, we devised the following algorithm for incremental closure. First, we assume we have a small cache of closures that have already been computed. This cache will contain the two input events, together with the convex closure that was computed. Our incremental closure algorithm is then as follows:

```
CC(E1,E2) {
  if (in_cache(E1,?E3,?CCcached) and E3 precedes E2) {
    compute CC(back(CCcached), E3);
    forall e (back(CCcached) precedes e precedes E2) {
      verify e is an acceptable event;
    }
    if (no unacceptable event is found) {
      update cache as appropriate;
      return convex closure;
  }
  if (exists a non-acceptable event OR
      no matching cached closure) {
    apply Xie and Taylor;
  }
}
```

We now describe the algorithm in detail. First, we check the cache to see if there is a matching input event. In this matching, we will only check against the first of the two input events, `E1`, in the closure computation. This is because that first event is stable, while the second is varied in the backtracking search process. On finding a match in the cache, we verify that the corresponding input event `E3` that is cached precedes the second input event, `E2` to this convex-closure computation. If this condition is true, we compute an incremental closure between **back** of the cached closure and the second input event. This, however, is insufficient. Per the theorem, the locations sets must be identical. To satisfy this condition, we must check all events between the cached convex closure and the new input event, `E2`, to determine if any are receive or synchronous events with a partner outside of the location set of the cached convex closure. If any such event exists, and that event is a successor to the cached closure, then the event is unacceptable. Specifically, such an event means that the location set of the closure will exceed that of the location set of the cached closure. Note that only the events that are part of the incremental closure need to be checked, and not those of the cached closure. This is typically a small number of events.

If no unacceptable event is found, then the cache should be updated as appropriate, and the closure returned. We have found that a suitable cache replacement policy is to replace the closure that was just used. Specifically, this means that a small cache may be used, while still rendering most closure operations into incremental operations. The closure returned will be the **front** set of the cached closure and the **back** set of the incremental closure computation.

If no matching cache element is found, or an unacceptable event is found (that is, the location sets do not match), then we simply revert to the Xie and Taylor algorithm.

4 Analysis

We have implemented our algorithm as an eclipse plug-in, within the basic predicate-detection system implemented by Xie and Taylor. This allows us to evaluate our algorithm both experimentally and analytically.

From an analytical perspective, we can do no better than Xie and Taylor, since we degenerate to their algorithm whenever we do not have a suitable basis for an incremental-closure computation. Further, we can do worse than Xie and Taylor when we consider the worst-case scenario. In this case, we will compute an incremental closure over all but a finite number of events in the computation. We then verify this, to determine if their exist unacceptable events. In the worst case, the last event checked fails the acceptability requirement, and thus we must compute the desired closure using the Xie and Taylor algorithm. The acceptability check is thus executed $O(n)$, where n is the number of events in the computation. The cost of the acceptability check is $O(N)$, since all events in **front** must be verified for non-precedence against . In such an instance, our algorithm would be $O(nN + N^3)$, while Xie and Taylor remains at $O(N^3)$.

While analytically we are no better, and in the worst case, worse than Xie and Taylor, in practice, our algorithm is substantially superior. We have evaluated our algorithm over more than 50 different parallel and distributed computations covering a variety of different environments, including Java [10], PVM [9], DCE [6], and μC++ [3] (a language used for teaching concurrency). The PVM programs tended to be SPMD style parallel computations. As such, they frequently exhibited close neighbour communication and scatter-gather patterns. The Java programs were web-like applications, including various web-server executions. The DCE programs were sample business-application code. The μC++ were sample concurrency problems used in an educational environemnt, such as Dining Philosophers.

For each experiment we used a variety of predicates, appropriate to the computation at hand. In the experiments we computed the number of convex closures, the number of unique **front** sets, the number of successful incremental closures, and the total execution time using our algorithm and the Xie and Taylor algorithm. The cache size employed was one, while the hardware used was a Pentium III 2 GHz, with 512 MB of memory, together with eclipse version 2.1.3.

For long-running queries, defined as those whose runtime exceeded 30 minutes when using the Xie and Taylor algorithm, we have found that our algorithm reduced the runtime by more than 90%. In one instance the runtime was reduced from over four hours to less than one minute. The cause of the substantial improvement is easily comprehended when we observe that, for such queries, the cache-hit rate always exceeded 90%. Further, we observed that the number of closures per unique **fronts** averaged 15. This means that, for a given **front** set, 15 closures were computed. In the Xie and Taylor algorithm, each such closure would be recomputed from scratch. In our approach, even with a cache size of one, we effectively only incur the cost of computing the largest such closure.

While space limitations prevent the publication of the code used, it is available on request from the first author. Further details of the algorithm, it's analysis, and raw result data is available in [2] and/or from the first author.

5 Related Work

Before concluding, we first briefly review related approaches. Existing work can be broken down into two main categories, corresponding to the two main types of pattern sought, and a third, smaller, but more recent, strand. There exists a significant body of work on seeking predicates in consistent global states (*e.g.*, [4, 22]), as we have alluded to in Sect.2. While such work is clearly critical in debugging, monitoring, and controlling parallel and distributed systems, it is fundamentally different from that of seeking patterns within the partial order itself.

Pattern seeking within the partial order has historically focused on a non-compound-event-based approach. Such work includes the offline algorithm of Jaekel [14] and its online version by Fox [7]. Neither method uses the compound-event-based approach of Xie and Taylor. A variant of the pattern-seeking approach to predicate detection is Han's technique for comparing two execution histories [11]. It is unclear if our work would be of relevance to her problem. The most recent work in this area is that of Xie and Taylor, and has already been described.

A third strand of work, which is quite recent, is typified by the IBM Log and Trace Analyzer [12]. This work takes the approach of using what event data is available, rather than adding monitoring code to an application. This approach is based on the observation that most enterprise applications already possess substantial log data which represent events of significance. Further, such applications are unlikely to be instrumented according to the desires of a third-party autonomic controller. The basic approach is that the logs are gleaned for event data, which the analyzer then attempts to correlate. The value of this approach is that it requires no change to existing systems. The success of the approach is dependent on the degree to which the existing sources possess sufficient information to provide correct correlation.

6 Conclusions

In this paper we have shown how to efficiently perform hierarchical predicate detection based on compound events. Our algorithm performs incremental closure computations, effectively reusing work already done. We have both proven our algorithm correct, and have demonstrated its efficacy via experiment. While our approach applies only to structural predicate detection, we expect to study its applicability to the problem of seeking patterns in consistent global states in the near future.

References

1. A. A. Basten. Hierarchical event-based behavioural abstraction in interactive distributed debugging: A theoretical approach. Master's thesis, Eindhoven University of Technology, Eindhoven, 1993.
2. Dwight S. Bedassé. An efficient computation of convex closure on abstract events. Master's thesis, University of Waterloo, Waterloo, Ontario, 2005. Available at: http://etheses.-uwaterloo.ca/display.cfm?ethesis_id=498.

3. P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software — Practice and Experience*, 22(2):137–172, February 1992.
4. Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11:191–201, 1998.
5. Eclipse Foundation. The eclipse platform. Online documentation available at: http://www.-eclipse.org/.
6. Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
7. Mark Fox. Event-predicate detection in the monitoring of distributed applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1998.
8. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.
9. Al Geist, Adam Begulin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at http://java.sun.com/docs/books/jls/.
11. Jessica Zhi Han. Automatic comparison of execution histories in the debugging of distributed applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1998.
12. IBM Corporation. Log and trace analyzer for autonomic computing. Online documentation available at: http://www.alphaworks.ibm.com/tech/logandtrace.
13. IBM Corporation. Object level trace. Online documentation available at: http://www-106.-ibm.com/developerworks/websphere/WASInfoCenter/infocenter/olt_content/olt/index.htm.
14. Christian E. Jaekl. Event-predicate detection in the debugging of distributed applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1997.
15. Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41– 50, 2003.
16. Deiter Kranzlmüller, Siegfried Grabner, R. Schall, and Jens Volkert. ATEMPT — A Tool for Event ManiPulaTion. Technical report, Institute for Computer Science, Johannes Kepler University Linz, May 1995.
17. Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, Linz, Austria, 2000.
18. Thomas Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization*. PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, 1994.
19. Thomas Kunz. Automatic support for understanding complex behaviour. In *Proceedings of the International Workshop on Network and Systems Management*, pages 125–132, August 1995.
20. Thomas Kunz, James P. Black, David J. Taylor, and Twan Basten. POET: Target-system independent visualisations of complex distributed-application executions. *The Computer Journal*, 40(8):499–512, 1997.
21. Leslie Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, 1978.
22. Alper Sen and Vijay K. Garg. On checking whether a predicate definitely holds. In *3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.
23. Alper Sen and Vijay K. Garg. Partial order trace analyzer (POTA) for distributed programs. In *Proc. Workshop on Runtime Verification*, 2003.
24. Ping Xie and David Taylor. Specifying and locating hierarchical patterns in event data. In *Proceedings of the 2004 CAS Conference*, pages 66–80, October 2004.