NORTH-HOLLAND

A Unified Framework for the Specification and Run-time Detection of Dynamic Properties in Distributed Computations*

Özalp Babaoğlu[1]
Department of Computer Science, University of Bologna, Piazza Porta S. Donato 5, 40127 Bologna, Italy

Eddy Fromentin[2] and Michel Raynal[2]
IRISA, Campus de Beaulieu, 35042 Rennes cedesc, France

To a large extent, the dependability of complex distributed programs relies on our ability to effectively test and debug their executions. Such an activity requires that we be able to specify dynamic properties that the distributed computation must (or must not) exhibit and that we be able to construct algorithms to detect these properties at run time. In this article we formulate dynamic property specification and detection as instances of the language recognition problem. Considering boolean predicates on states of the computation as an alphabet, dynamic property specification is akin to defining a language over this alphabet. Detecting a property, on the other hand, is akin to recognizing at run time if the sentence produced by a distributed execution belongs to the language. This formal language-oriented view not only unifies a large body of work on distributed debugging and property detection, it also leads to simple and efficient detection algorithms. We give examples for the case of properties that can be specified as regular grammars through finite automata.

1. INTRODUCTION

Our inability to formally prove the correctness of all but the most trivial distributed programs leaves testing and debugging as the only viable alternatives for arguing about their dependability. A fundamental step in program debugging is specifying which set of executions are considered desirable and which ones are considered erroneous. Informally, a desired (or undesired) temporal evolution of a distributed program's states is called a *dynamic property*. As such, a dynamic property (*property* for short) defines a subset of executions among all those that are possible for the program. Once the properties of interest for a distributed program have been defined, the act of debugging consists of verifying if its executions satisfy these properties.

There are three possible times at which one can prove properties of a distributed program: prior to any execution, during an execution or after an execution. The ability to prove a program property prior to executing it requires reasoning about the program itself as well as the distributed system. In other words, we need to characterize all possible executions of the program given a formal description of its actions (the code) and the environment in which it is to be run. Model checking (Clarke et al., 1986) is one such technique where the program, modeled as a finite-state transition system, is analyzed by traversing the trees representing all possible execu-

Address correspondence to Michel Raynal, IRISA, Campus de Beaulieu, 35042 Rennes cedesc, France.

**This work has been supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Project 6360 (BROADCAST).*

[1]*Further supported by the Italian National Research Council and the Ministry of University, Research and Technology.*

[2]*Further supported by the CNRS under the grant Parallel Traces.*

J. SYSTEMS SOFTWARE 1996; 33:287–298

655 Avenue of the Americas, New York, NY 10010

0164-1212/96/$15.00
PII S0164-1212(96)00027-1

tions, checking to see if they satisfy properties expressed as temporal logic formulas. As an alternative to proving properties a priori, they can be checked concurrently with an actual execution of the program through *run-time property detection*. With this technique, conclusions that are drawn are not about all possible executions of the program, but about all possible *observations* (Babaoğlu and Marzullo, 1993; Schwarz and Mattern, 1994) of an actual execution. The third alternative—reasoning about program properties after an execution—is called *post-mortem* analysis and is similar to run-time property detection. The basic difference is that the analysis has to be based on data collected during the execution (traces) and is performed at program termination, thus making it unsuitable for many applications.

Ideally, one would like to prove as many properties as possible for a program prior to its execution. Unfortunately, techniques such as model checking may be infeasible or have prohibitive costs for complex programs. Furthermore, while these techniques are effective for proving program properties such as "is the program deadlock free?" they cannot address inherently run-time properties such as "has the program terminated?". Much of distributed debugging is an inherently run-time activity in that, short of having proven that the program is error free, we can only hope to detect desired or erroneous sequences of states that result during an actual execution soon after their occurrence. Applications that have a *reactive architecture* (Harel and Pneuli, 1985) (of which distributed debugging is an instance) are yet other examples where the fundamental abstraction is run-time property detection. Clearly, run-time property detection and model checking can be seen as complementary techniques towards distributed debugging—the greater the number of properties that can be verified a priori, the fewer the number of properties that need to be detected at run time.

Early work in run-time property detection has concentrated on the detection of *stable properties* (Chandy and Lamport, 1985) including distributed termination (Dijkstra and Scholten, 1980; Francez, 1980) and deadlocks (Chandy et al., 1983; Bracha and Toueg, 1987). Informally, these properties are stable in the sense that once verified during an execution, they remain true thereafter. Efficient algorithms have been developed that can detect stable properties of distributed computations at run-time (Chandy and Lamport, 1985; Hélary et al., 1987). Unfortunately, stable property detection has limited utility in the context of distributed debugging. Most of the properties that characterize desirable or erroneous executions for debugging purposes are transient whereby they may be verified during some interval of the execution but cease to be satisfied later on. As such, the appropriate formalism for distributed debugging can be seen as *unstable property* detection. If erroneous behaviors are specified as unstable properties, then their detection during an execution reveals a fault in the corresponding program. Existence of many executions during which the unstable properties are not detected increases our confidence in the correctness of the corresponding program.

In this article, we consider the problem of specifying unstable properties and detecting them at run time. We formalize the problem as the design of a decision algorithm that, when superimposed on a distributed execution, will answer "yes" if and only if the distributed execution satisfies a property specified as a formula in some formal language. We develop a general framework for property specification and detection drawing on concepts from formal language theory. The framework, described in Section 3, is based on the labeling of a directed acyclic graph (DAG). Each node of the DAG is associated a set of labels from a given alphabet and each path of the DAG is associated a set of words constructed from the same alphabet. The set of all words corresponding to all paths terminating at a node defines a language. The language recognition problem is then defined in Section 3.4 as the decision procedure for determining if a given word belongs to this language. We instantiate this abstract problem for two possible models of distributed computations as a DAG—in Section 4 as the partially ordered set of local states and in Section 5 as the lattice of consistent global states. When the alphabet used for labeling these DAGs consists of boolean predicates, the formal language associated with the nodes of the graph effectively specifies dynamic properties of the corresponding distributed computation. In addition, run-time property detection reduces to the problem of language recognition. In Sections 4.3 and 5.3, we give examples of simple run-time detection algorithms that result for the case of properties that can be specified as regular grammars through finite automata. This framework unifies a large body of work on distributed debugging and property detection including: linked predicates (Miller and Choi, 1988), conjunction of local predicates (Garg and Waldecker, 1992), atomic sequences of predicates (Hurfin et al., 1993b) relational global predicates (Tomlinson and Garg, 1993), unstable predicates on global states (Cooper and Marzullo, 1991), interval-constrained sequences (Babaoğlu and Raynal, 1995), regular pat-

terns (Fromentin et al., 1994) and regular properties (Jard et al. 1994). The algorithms developed in this work are currently being implemented in the context of the EREBUS distributed debugging facility which is described in Section 6.

2. DISTRIBUTED COMPUTATIONS

A distributed program is one that is executed by a collection of sequential processes, denoted $P_1, \ldots, P_n$ for some $n > 1$, that can communicate by exchanging messages. Processes have access to neither shared memory nor a global clock. Communication incurs finite but arbitrary delays. Without loss of generality, we assume that each process can reliably communicate with every other process.

2.1 Distributed Computations as Partially-Ordered Sets

Execution of process P_i produces a sequence of events h_i called its *local history*. Each event of the local history may be either internal, causing only a local state change, or involve communication with another process through *send* or *receive* events. Let $h_i = e_i^0 e_i^1 \ldots$ be the local history of process P_i. Events of h_i are enumerated according to the total order in which they are executed by P_i and e_i^0 is a fictitious event introduced for initializing the local state of P_i.

Let H be the set of all events and let $\rightarrow$ be the binary relation denoting causal precedence (Lamport, 1978) between events defined as follows

$$e_i^a \rightarrow e_j^b \equiv \begin{cases} (i = j) \wedge (b = a + 1) & \text{or} \\ (e_i^a = send(m)) \wedge \left(e_j^b = receive(m)\right) & \text{or} \\ \exists e_k^c : (e_i^a \rightarrow e_k^c) \wedge \left(e_k^c \rightarrow e_j^b\right). \end{cases}$$

Formally, a distributed comparison can then be modeled as the partially-ordered set (poset) $\mathcal{H} = (H, \rightarrow)$. Figure 1 illustrates a distributed computation consisting of three processes using a graphical representation of the partial order between events known as a space-time diagram.

2.2 Distributed Computations as Directed Acyclic Graphs of Local States

Let σ_i^a be the local state of process P_i immediately after having executed event e_i^a, and let S be the set of all local states. Analogous to the causal precedence relation between pairs of events, we define the binary relation $\prec$ to denote *immediately causal precedence* between local states as follows

$$\sigma_i^a \prec \sigma_j^b \equiv \begin{cases} (i = j) \wedge (b = a + 1) & \text{or} \\ (e_i^{a+1} = send(m)) \wedge \left(e_j^b = receive(m)\right). \end{cases}$$

With respect to the set of local states, a distributed computation can be modeled as yet another DAG, denoted $\mathcal{S} = (S, \prec)$. Figure 2 depicts such a DAG of local states corresponding to the distributed computation of Figure 1.

Two local states of a distributed computation that are related through $\prec$ are said to be *adjacent* in that computation. A *control flow* associated with some local state σ of a computation $\mathcal{S}$ is a sequence of local states that are pairwise adjacent in $\mathcal{S}$ and the sequence begins at some initial state and terminates at σ.

2.3 Distributed Computations as Lattices of Global States

A global state $\Sigma = (\sigma_1, \ldots, \sigma_n)$ of a distributed computation is an n-tuple of local states, one for each

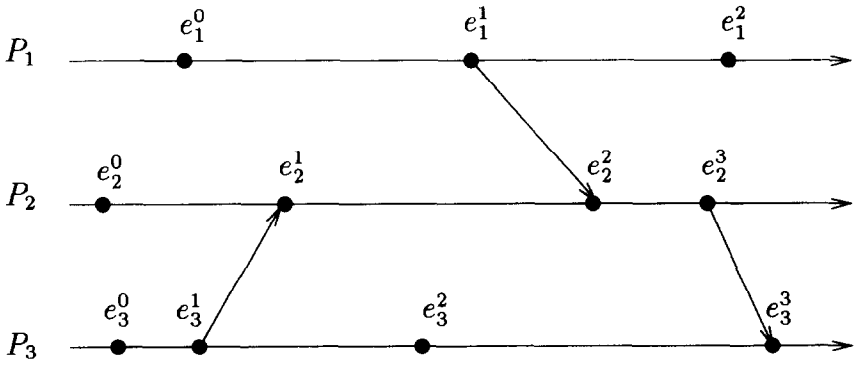

Figure 1. Distributed computation as a partially-ordered set of events.

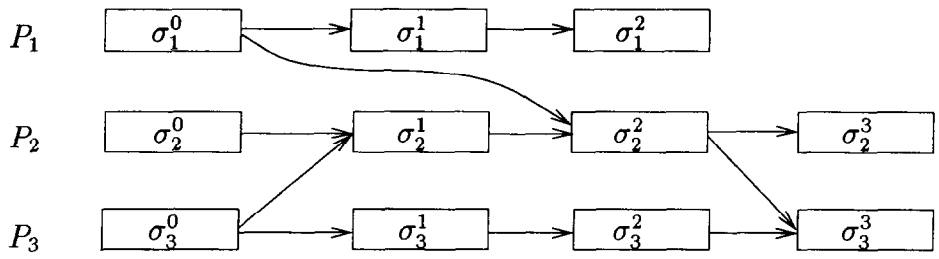

Figure 2. Distributed computation as a DAG of local states.

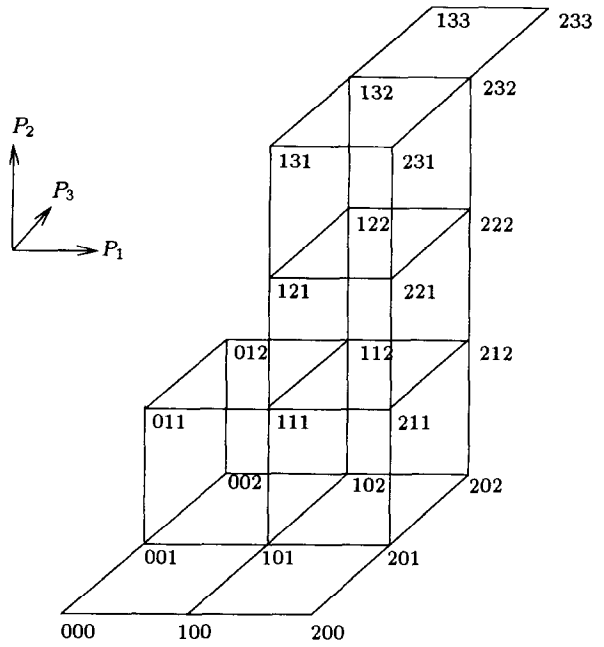

Figure 3. A distributed computation as a lattice of global states.

process. Intuitively, a global state of a computation is said to be consistent if an omniscient external observer could actually observe the computation enter that state. More formally, a global state $\Sigma = (\sigma_1, \ldots, \sigma_n)$ is *consistent* if and only if for all pairs of its local states (σ_i, σ_j), neither $\sigma_i \prec \sigma_j$ nor $\sigma_j \prec \sigma_i$, where $\prec$ denotes the transitive closure of the immediate causal precedence relation between local states.

The set of all consistent global states for a distributed computation has a lattice structure whose minimal element corresponds to the initial global state $\Sigma^0 = (\sigma_1^0, \ldots, \sigma_n^0)$. Let $\mathscr{L}$ be this lattice. An edge exists from node $\Sigma = (\sigma_1, \ldots, \sigma_i^a, \ldots, \sigma_n)$ to node $\Sigma' = (\sigma_1, \ldots, \sigma_i^{a+1}, \ldots, \sigma_n)$ in $\mathscr{L}$ if and only if there exists an event e that can be executed by P_i in local state σ_i^a. Figure 3 depicts the lattice $\mathscr{L}$ of global states associated with the distributed computation of Figure 1. In the lattice, an n-tuple $(x_1, \ldots, x_n)$ of natural numbers is used as a shorthand to denote the global state $\Sigma = (\sigma_1^{x_1}, \ldots, \sigma_n^{x_n})$.

Informally, a *sequential observation* (*observation* for short) of a distributed computation is the sequence of its global states that could have been constructed by an omniscient external observer. Equivalently, an observation is a sequence of global states that would result if the distributed program were to be executed on a single sequential processor. More formally, a sequence of global states $\Sigma^0\Sigma^1\Sigma^2 \cdots \Sigma^{i-1}\Sigma^i \cdots$ is an observation if there exists a sequence of events $e^1e^2 \cdots$ that is a linear extension of the partial order $\mathscr{H}$ (i.e., all events appear in an order consistent with the relation $\rightarrow$) such that Σ^i is the global state that results after executing event e^i in Σ^{i-1}.

By construction, each path of the lattice starting at the minimal element and proceeding upwards corresponds to an observation of the computation and each observation corresponds to a path in the lattice (Babaoğlu and Marzullo, 1993; Schwarz and Mattern, 1994). In other words, the lattice of consistent global states represents all possible observations for the computation. Note that, internal to the computation, the actual sequence of global states that is produced cannot be known, and this lattice represents the best information that is available.

3. PROPERTIES AS LANGUAGES OVER LABELED DAGs

In the previous section, we showed that a distributed computation $\mathscr{H}$ can be represented either as $\mathscr{S}$, the DAG of local states, or as $\mathscr{L}$, the DAG corresponding to the lattice of global states. In this section, we will develop a general framework for specifying dynamic properties based on labeling a generic DAG. In the following sections, we will instantiate this framework with the two specific DAGs for distributed computations $\mathscr{S}$ and $\mathscr{L}$, depending on

whether we are interested in sequences of local or global properties, respectively. Development of this framework helps us understand and unify a large number of proposals for detection of properties on local and global states as instances of a single problem.

3.1 Graph Labeling and Languages

Let $G = (V, E)$ be a DAG, and let A be a finite alphabet of symbols. We define a *labeling function* λ that maps nodes of G to nonempty sets of symbols drawn from A. For each node $v \in V$, the set $\lambda(v)$ is called the *label* of v.[1] Figure 4 illustrates a labeling for the DAG of ten nodes with the alphabet $A = \{a, b, c\}$.

For each node v, let G_v be the subgraph obtained from G by retaining only node v and all of its predecessors in G. Clearly if G is a DAG, then so is G_v. A *direct path* of G_v is a sequence of nodes starting at a source node (i.e., one with no predecessors) and ending at node v. Let Π_v be the set of all such paths in G_v. We extend the notion of node labeling to paths by associating with them words constructed from the same alphabet used to label nodes. Let A be a finite alphabet, λ be a labeling function, and $\pi_v = u_0 u_1 \ldots u_k$ be a directed path of G_v. The label of path π_v is the set $\tilde{\lambda}(\pi_v)$ of all words $\omega = \omega_0 \omega_1 \ldots \omega_k$ such that $\omega_i \in \lambda(u_i)$ for each node u_i in path π_v.

Because each path label is a set of words, sets of path labels can be seen as defining a language. The language associated with node v of G under the labeling function λ, denoted $L^\lambda(v)$, is defined as the set of words that are the labels of all directed paths of G_v. In other words,

$$L^\lambda(v) = \bigcup_{\pi_v \in \Pi_v} \tilde{\lambda}(\pi_v).$$

As an example, consider node v_{10} of the labeled DAG of Figure 4. The set of all directed paths for subgraph $G_{v_{10}}$ is $\Pi_{v_{10}} = \{v_1 v_2 v_6 v_9 v_{10}, v_1 v_{10}, v_7 v_5 v_6 v_9 v_{10}, v_7 v_8 v_9 v_{10}\}$. Under the labeling function λ that is illustrated, the language associated with node v_{10} is

$$L^\lambda(v_{10}) = \{abaac, abcac, ac, accac, bcaac, acaac, bccac, aaac, baac\}.$$

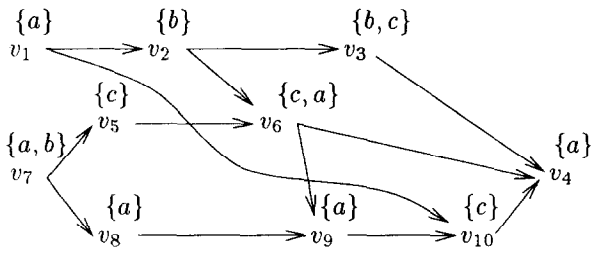

Figure 4. A directed acyclic graph and its labeling.

3.2 Dynamic Properties

Informally, we would like dynamic properties to characterize the time evolution of states that arise during program execution. In our framework, a *dynamic property* (*property* for short) defines a set of words over some finite alphabet. We find it convenient to distinguish the name of the property from the language that it defines. Let $L(\Phi)$ be the set of words associated with property Φ.

Informally, we think of each word in language $L(\Phi)$ as satisfying property Φ. In defining properties for DAGs, we can extend this notion of satisfaction to the entire graph in two possible ways.[2]

Definition 1. *Given an alphabet A, a directed acyclic graph G, a labeling function λ, a node v of G and a property Φ over A, we say that node v satisfies* **SOME** Φ, *denoted* $v \models$ **SOME** Φ, *if and only if there exists some labeling of some directed path in G_v that defines a word in the language of Φ. In other words,* $v \models$ **SOME** $\Phi \equiv L^\lambda(v) \cap L(\Phi) \neq \emptyset$.

Definition 2. *Given an alphabet A, a directed acyclic graph G, a labeling function λ, a node v of G, and a property Φ over A, we say that node v satisfies* **ALL** Φ, *denoted* $v \models$ **ALL** Φ, *if and only if all labelings of all directed paths in G_v define words in the language of Φ. In other words,* $v \models$ **ALL** $\Phi \equiv L^\lambda(v) \subseteq L(\Phi)$.

As an example, consider the two properties Φ_1 and Φ_2 defined over the alphabet $A = \{a, b, c\}$ associated with the languages $L(\Phi_1) = aba^*c(ac)^*$ and

[1] We assume that the empty symbol ϵ is implicitly included in every alphabet and constitutes the label of a node in case the labeling function defines no other symbols.

[2] Satisfaction rules similar to these have been proposed in other contexts (Cooper and Marzullo, 1991; Garg and Waldecker, 1992; Fromentin et al., 1994; Jard et al., 1994, Fromentin et al., 1995). In particular, our definitions are in the same spirit as those of modal operators *POS* and *DEF* of (Cooper and Marzullo, 1991), *strong* and *weak* of (Garg and Waldecker, 1992) and *POT*, *INEV*, *SOME*, and *ALL* of more general transition systems (Clarke et al., 1986; Queille and Sifakis, 1983).

$L(\Phi_2) = (b + c)^*a(a + b)^*c(a + b + c)^*$, respectively.[3] With respect to the DAG of Figure 4, the following assertions hold

$v_{10} \models$ **SOME** Φ_1[4]

$v_{10} \models$ **ALL** Φ_2.

3.3 Duality of Modal Operators

As defined, the two modal operators **SOME** and **ALL** are duals of each other. This can be seen easily as follows. Let β be a language defined over some alphabet A and let $\overline{\beta}$ be the complement language of β over the same alphabet. In other words, $\overline{\beta}$ includes all words that are not included in β. Then for any other language γ over A, the following holds

$$(\gamma \subseteq \beta) \equiv (\gamma \cap \overline{\beta} = \emptyset).$$

Now let Φ be a property with its associated language $L(\Phi)$ and let $\overline{\Phi}$ be the complementary property with its associated language $L(\overline{\Phi}) = \overline{L(\Phi)}$. Thus, Definitions 1 and 2 can be rewritten as

$$(v \models \textbf{SOME } \Phi) \equiv (L^\lambda(v) \cap L(\Phi) \neq \emptyset)$$
$$\equiv \neg(L^\lambda(v) \subseteq \overline{L(\Phi)})$$

$$(v \models \textbf{ALL } \Phi) \equiv (L^\lambda(v) \subseteq L(\Phi)) \equiv (L^\lambda(v) \cap \overline{L(\Phi)} = \emptyset).$$

Finally, we obtain the desired duality relationships by substitution

$$v \models \textbf{SOME } \Phi \equiv \neg(v \models \textbf{ALL } \overline{\Phi})$$

$$v \models \textbf{ALL } \Phi \equiv \neg(v \models \textbf{SOME } \overline{\Phi}).$$

3.4 Detection of Dynamic Properties

For the sake of concreteness, in what follows we consider only properties that correspond to regular languages. As we shall see, most of the existing proposals, including behavior patterns on local states (Miller and Choi, 1988; Fromentin et al., 1994) and global states (Cooper and Marzullo, 1991; Babaoğlu and Raynal, 1995), happen to be special cases of such properties. Moreover, these properties admit simple and efficient detection algorithms.

It is well known that grammars that specify regular languages are equivalent to deterministic finite-state automata. Formally, an *automaton* is a 5-tuple (A, Q, q_0, Q_F, δ) where

A is a finite alphabet
Q is a finite set of states
q_0 is an initial state
Q_F is a set of accepting states
δ is a deterministic transition function.

Let Φ be a property such that $L(\Phi)$ is a regular language. The finite state automaton that recognizes $L(\Phi)$ is given the same name as the property itself. Figure 5 illustrates the two automata recognizing the languages associated with the properties Φ_1 and Φ_2 of the example of Section 3.2. In the figure, accepting states are shown as triangles.

Given a property Φ and DAG G, let $R^\Phi(v)$ denote the set of states of the automaton recognizing $L(\Phi)$ that are reached after processing all of the words in $L^\lambda(v)$. Recall that $L^\lambda(v)$ is the language associated with node v of G defined as the set of all words that are labels of all directed paths of G_v. For example, given the automata of Figure 5 recognizing $L(\Phi_1)$ and $L(\Phi_2)$, and the DAG of Figure 4, we have $R^{\Phi_1}(v_{10}) = \{q_3, q_5\}$ and $R^{\Phi_2}(v_{10}) = \{q_2\}$. Note that $R^{\Phi_1}(v_{10}) \cap Q_F \neq \emptyset$, while $R^{\Phi_2}(v_{10}) \subseteq Q_F$. From the previous section, we know that for this example $v_{10} \models$ **SOME** Φ_1 and $v_{10} \models$ **ALL** Φ_2. In fact, this relationship can be shown to hold in general by rewriting the satisfaction rules of Section 3.2 as follows after applying the definitions

$$v \models \textbf{SOME } \Phi \equiv (R^\Phi(v) \cap Q_F \neq \emptyset)$$

$$v \models \textbf{ALL } \Phi \equiv (R^\Phi(v) \subseteq Q_F).$$

Expressing the satisfaction rules in terms of relationships between the accepting states and the set of reachable states of the automaton gives us effective decision procedures for computing them.

Given a property Φ, an automaton accepting $L(\Phi)$ and a labeled DAG G, the problem of detecting the propety can be reduced to computing the sets $R^\Phi(v)$ for each node v of G. We proceed inductively in defining $R^\Phi(v)$.

Base case. First, we augment G by adding a fictitious node v_0 and new edges such that v_0 becomes

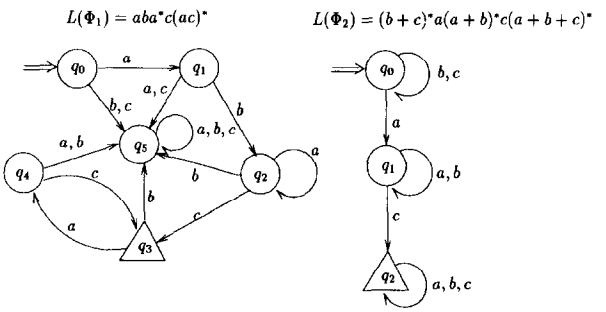

Figure 5. Two properties and their respective languages as finite automata.

[3] In specifying these languages, we use regular expressions with the usual syntax rules.

[4] Words *abaac* and *abcac* of $L^\lambda(v_{10})$ are also in $L(\Phi_1)$.

```
R^Φ(v_0) := {q_0}
done := {v_0}
while (∃v ∈ V : (v ∉ done) ∧ (∃u ∈ done : (u, v) ∈ E)) do
    reached := {}
    foreach (v ∉ done : pred(v) ⊆ done) do
        R^Φ(v) := {}
        foreach α ∈ λ(v) do
            foreach u ∈ pred(v) do
                foreach q ∈ R^Φ(u) do
                    R^Φ(v) := R^Φ(v) ∪ {δ(q, α)}
                od
            od
        od
        reached := reached ∪ {v}
    od
    done := done ∪ reached
od
```

Figure 6. A generic algorithm for detecting property Φ in directed acyclic graph $G(V, E)$ with labeling function $\lambda()$.

an immediate predecessor of all source nodes in the initial graph G. The node labeling function is extended such that $\lambda(v_0) = \{\epsilon\}$. Then, by definition $R^{\Phi}(v_0) = \{q_0\}$.

Inductive step. Let $pred(v)$ be the set of nodes that are immediate predecessors of node v in G. Let $R^{\Phi}_{pred}(v)$ be the set of reachable states of the automaton recognizing $L(\Phi)$ after processing all words associated with nodes in $pred(v)$. In other words,

$$R^{\Phi}_{pred(v)} = \bigcup_{u \in pred(v)} R^{\Phi}(u).$$

Thus, by induction we have

$$R^{\Phi}(v) = \bigcup_{q \in R^{\Phi}_{pred(v)},\ \alpha \in \lambda(v)} \delta(q, \alpha).$$

The above inductive definition can be easily transformed into a computation by doing a breadth-first traversal of G starting at node v_0 as shown in Figure 6.

4. PROPERTIES ON CONTROL FLOWS

In this section, we will instantiate the generic dynamic property detection algorithm of the previous section in order to detect properties on control flows. In other words, the directed acyclic graph of interest is the DAG of local states $\mathcal{S} = (S, \prec)$ defined in Section 2.2 and the alphabet of interest is a set of local predicates.

4.1 Local Predicates

A *local predicate* is a formula in propositional logic (boolean expression) naming only variables that are local to a single process. Let ϕ be such a local predicate. If the predicate holds in some local state σ of $\mathcal{S}$, we say that σ *satisfies* ϕ and write $\sigma \models \phi$. Let A be an alphabet consisting of a finite set of local predicates. We define a labeling function for the distributed computation $\mathcal{S} = (S, \prec)$ such that the labels of a local state are the local predicates it satisfies

$$\forall \sigma \in S : \lambda(\sigma) = \{\phi \in A : \sigma \models \phi\}.$$

As an example, consider the distributed computation of Figure 7 where x, y and z are three variables local to processes P_1, P_2, and P_3, respectively. Each local state σ_i^k is characterized by the value of the corresponding local variable. In Figure 7, values of the variables are shown next to each of the local states (e.g., in local state σ_2^1 the local variable y has value 2). Let us consider the following four local predicates $\phi_1 \equiv (x < 3)$, $\phi_2 \equiv (x \text{ is prime})$, $\phi_3 \equiv (y \neq 0)$ and $\phi_4 \equiv (z < 4)$. For the computation of Figure 7, we have $\sigma_1^0 \models \phi_1$, $\sigma_1^0 \models \phi_2$, $\sigma_1^2 \models \phi_1$, $\sigma_1^2 \models \phi_2$, $\sigma_2^0 \models \phi_3$, $\sigma_2^1 \models \phi_3$, $\sigma_2^2 \models \phi_3$, $\sigma_2^3 \models \phi_3$, $\sigma_3^0 \models \phi_4$, $\sigma_3^1 \models \phi_4$ and $\sigma_3^3 \models \phi_4$. Thus, $\mathcal{S}$ labeled with the alphabet $A = \{\phi_1, \phi_2, \phi_3, \phi_4\}$ is as shown in Figure 7.

4.2 Behavior Patterns on Local States

Consider the class of properties that can be specified as a set of predicates that need to be satisfied in a particular order during a computation. Each such sequence is called a *behavior pattern*. Recall that in Section 2.2 we defined a control flow associated with

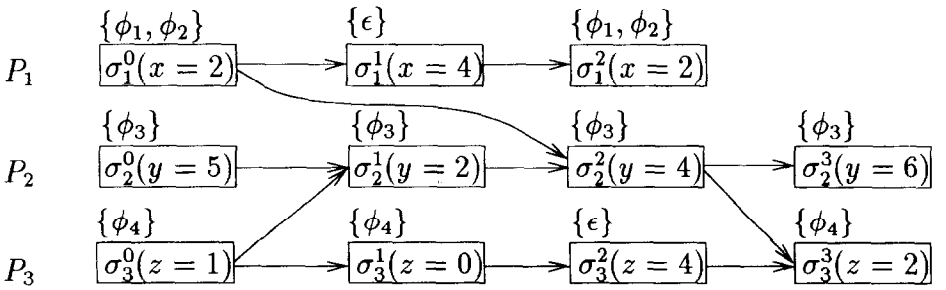

Figure 7. DAG of local states annotated with values of three variables and node labels.

some local state σ as a sequence of local states that are pairwise adjacent in $\mathcal{S}$ and the sequence begins at some initial state and terminates at σ. In other words, control flows of a computation correspond to directed paths of $\mathcal{S}$. Given an alphabet of local predicates, the language $L(\Phi)$ associated with the control flows of a local state is able to express behavior patterns that are admissible by the property Φ.

The satisfaction rules introduced in Section 3.2 have the following meaning when interpreted in the context of distributed computations as a DAG of local states.

Definition 3. *Given a computation $\mathcal{S}$ and property Φ on control flows, a local state σ satisfies* **SOME** Φ *if and only if there exists a control flow π_σ terminating at local state σ such that $\tilde{\lambda}(\pi_\sigma)$ includes at least one word of $L(\Phi)$. That is,*

$$\sigma \models \textbf{SOME } \Phi \equiv (L^{\lambda}(\sigma) \cap L(\Phi) \neq \emptyset).$$

Definition 4. *Given a computation $\mathcal{S}$ and property Φ on control flows, a local state σ satisfies* **ALL** Φ *if and only if for each control flows π_σ terminating at local state σ, every word of $\tilde{\lambda}(\pi_\sigma)$ is included in $L(\Phi)$. That is,*

$$\sigma \models \textbf{ALL } \Phi \equiv (L^{\lambda}(\sigma) \subseteq L(\Phi)).$$

For example, consider the properties $\Phi_1 = \phi_1\phi_3^+$, $\Phi_2 = \phi_1\phi_3^+\phi_4$, $\Phi_3 = \phi_1\phi_4$ and $\Phi_4 = (\phi_1 + \phi_4)^*\phi_3^+$ on control flows specified as regular expressions over the alphabet $A = \{\phi_1, \phi_2, \phi_3, \phi_4\}$ as defined before. For the computation of Figure 7, the following relations hold

$$\sigma_2^2 \models \textbf{SOME } \Phi_1$$

$$\sigma_3^3 \models \textbf{SOME } \Phi_2$$

$$\neg(\sigma_3^3 \models \textbf{SOME } \Phi_3)$$

$$\sigma_2^3 \models \textbf{ALL } \Phi_4$$

$$\neg(\sigma_2^3 \models \textbf{ALL } \Phi_1).$$

Recalling the duality results of Section 3.3, we can immediately conclude that $\sigma_3^3 \models \textbf{ALL } \overline{\Phi}_3$ and $\sigma_2^3 \models \textbf{SOME } \overline{\Phi}_1$.

4.3 Run-Time Detection of Properties on Control Flows

As discussed in Section 3.4, regular languages can be recognized through deterministic finite state automata. In the case of properties on control flows, we can instantiate the generic algorithm of Figure 6 such that detection can be done at run time without introducing any delays and without adding any control messages to the distributed computation. All that is necessary is to piggyback control information onto the existing messages of the computation as described below.

Let Φ be a property such that $L(\Phi)$ can be recognized by the finite state automaton $\Phi = (Q, A, q_0, Q_F, \delta)$. A *controller* is superimposed on each process P_i of the computation that maintains an array $B_i[Q]$ of boolean values with the following semantics: For each state $q \in Q$, the element $B_i[q]$ is set to true if and only if there exists a control flow π_σ terminating at the current local state σ of P_i such that at least one word in $\tilde{\lambda}(\pi_\sigma)$ places automaton Φ in state q. Initially, only $B_i[q_0]$ is defined to be true. The algorithm executed by each controller can be easily derived from the generic algorithm and is shown in Figure 8. Note that the detection algorithms presented in (Miller and Choi, 1988; Hurfin et al., 1993; Fromentin et al., 1994) are particular cases of this generic algorithm.

Let B_σ denote the value of array B at local state σ as maintained by the algorithm of Figure 8. Then, the satisfaction rules for **SOME** and **ALL** applied to properties on control flows can be easily computed through the following relations that are obtained from the definition of array B

$$\sigma \models \textbf{SOME } \Phi \equiv \exists q \in Q : ((B_\sigma[q] = \mathit{true}) \wedge q \in Q_F)$$

$$\sigma \models \textbf{ALL } \Phi \equiv \forall q \in Q : ((B_\sigma[q] = \mathit{true}) \Rightarrow q \in Q_F).$$

The duality relations can be interpreted in terms of these definitions as follows. Let $\Phi = (A, Q, q_0, Q_F, \delta)$ be the finite state automaton accepting the language $L(\Phi)$. Then, the automaton $\overline{\Phi} = (A, Q, q_0, Q - Q_F, \delta)$ accepts the language $L(\overline{\Phi})$ which, by definition, is $\overline{L(\Phi)}$. Thus, the duality relations take on the following form when inter-

```
when P_i enters a new local state σ_i:
    foreach α ∈ λ(σ_i) do
        B^α[Q] := (false, ···, false)
        foreach (q ∈ Q : B_i[q] = true) do
            foreach r ∈ δ(q, α) do B^α[r] := true od
        od
    od
    foreach q ∈ Q do B_i[q] := ⋁_{α∈λ(σ_i)} B^α[q] od

when P_i sends a message m:
    piggyback B_i[Q] on m

when P_i receives m containing B_m:
    foreach q ∈ Q do B_i[q] := B_i[q] ∨ B_m[q] od
```

Figure 8. Algorithm executed by the controller of process P_i in order to detect properties on control flows.

preted with respect to properties on control flows

$\sigma \models$ **SOME** Φ
$\equiv \neg(\forall q \in Q : ((B_\sigma[q] = true) \Rightarrow q \in Q - Q_F))$

$\sigma \models$ **ALL** Φ
$\equiv \neg(\exists q \in Q : ((B_\sigma[q] = true) \wedge q \in Q - Q_F))$.

5. PROPERTIES ON OBSERVATIONS

There are many interesting properties of distributed computations that cannot be specified in terms of control flows, which only consider local predicates. Properties such as deadlock, termination, and mutual exclusion require us to reason about global states of distributed computations. Thus, we will instantiate the generic dynamic property detection algorithm of Section 3.4 in order to detect properties on observations, which are in terms of global states. The directed acyclic graph of interest is the lattice of global states $\mathscr{L}$ defined in Section 2.3 and the alphabet of interest is a set of global predicates.

5.1 Global Predicates

A *global predicate* is a formula in propositional logic that can name any variable of any process. It is meaningful to evaluate global predicates only in global states that are consistent. As such, the appropriate model for interpreting global predicates is the lattice $\mathscr{L}$. If a global predicate φ holds in some global state Σ of $\mathscr{L}$, we say that Σ satisfies φ and write $\Sigma \models \varphi$.

Given a finite alphabet A of global predicates, a labeling function λ is defined on global states in a way analogous to that for local states

$\forall \Sigma : \lambda(\Sigma) = \{\varphi \in A : \Sigma \models \varphi\}$.

Consider for example the lattice of Figure 9 corresponding to the distributed computation of Figure 7 and the two global predicates

$\varphi_1 \equiv ((x > y) \wedge (y > z))$

$\varphi_2 \equiv ((x \neq y) \Rightarrow (x > z))$.

We can see that global state $\Sigma_1 = (\sigma_1^1, \sigma_2^1, \sigma_3^1)$ (indicated as 111 in Figure 9) satisfies both φ_1 and φ_2, thus obtaining the label $\{\varphi_1, \varphi_2\}$. The labeling of the entire lattice with the alphabet $A = \{\varphi_1, \varphi_2\}$ is shown in Figure 9.

5.2 Behavior Patterns on Observations

In Section 4.2 we defined behavior pattern as a sequence of predicates that a computation satisfies and considered properties that could be specified in terms of behavior patterns on local states. We now consider properties on global states specified as sets of behavior patterns on observations. In this case, a sequence of global predicates defining a behavior pattern must be satisfied by a sequence of global states in order for the corresponding property be recognized. In other words, a global property Φ, through its associated language $L(\Phi)$, defines a set of admissible behavior patterns on observations.

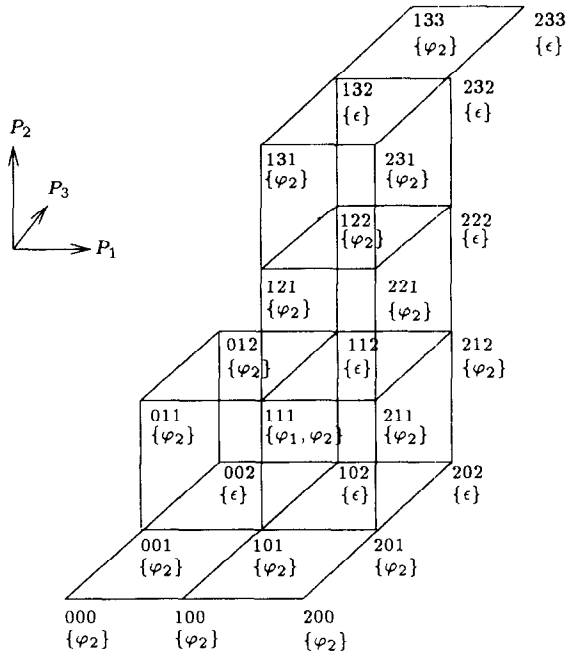

Figure 9. Lattice $\mathscr{L}$ of global states labeled with predicates.

With respect to the lattice of global states, an observation is the analog of a control flow for local states. As such, we can define the modal operators **SOME** and **ALL** in terms of global properties and observations in a manner analogous to those for local properties.

Definition 5. *Given an alphabet A of global predicates, lattice $\mathscr{L}$, labeling function λ, node Σ of $\mathscr{L}$ and a property Φ over A, $\Sigma \models$ **SOME** Φ, if and only if there exists some labeling of at least one observation terminating in Σ that defines a word in $L(\Phi)$.*

Definition 6. *Given an alphabet A of global predicates, lattice $\mathscr{L}$, labeling function λ, node Σ of $\mathscr{L}$ and a property Φ over A, $\Sigma \models$ **ALL** Φ, if and only if all labelings of all observations terminating in Σ define words in $L(\Phi)$.*

For the example of Figure 9 and the global predicates φ_1 and φ_2 as defined above, we have

$(\sigma_1^2, \sigma_2^3, \sigma_3^3) \models$ **SOME** $(\varphi_2^+ \varphi_1 \varphi_2^+)$ with alphabet $A = \{\varphi_1, \varphi_2\}$ while $(\sigma_1^2, \sigma_2^3, \sigma_3^3) \models$ **ALL** φ_2^+ with alphabet $A = \{\varphi_2\}$.

5.3 Run-Time Detection of Properties on Observations

A fundamental requirement for detecting properties on observations is the construction of consistent global states so that global predicates may be evaluated over them. In the most general case, the entire lattice of global states, representing all observations of the computation in question has to be constructed and traversed. To achieve this, each process P_i of the computation is augmented with a controller that sends local states produced by P_i to a monitor. The monitor pieces together local states received from the processes in order to construct consistent global states and to incrementally build the lattice. Several algorithms are known for doing such constructions based on a mechanism of vector clocks to ensure consistency of the global states (Cooper and Marzullo, 1991; Diehl et al., 1993).

Construction of the lattice and checking of properties can be performed at run time (i.e., concurrently with the distributed computation) by the monitor through the algorithm shown in Figure 10. Note that this is simply the generic algorithm of Figure 6 instantiated with the lattice $\mathscr{L}$ as the DAG.

As indicated in Section 3.4, we consider only the detection of properties corresponding to regular languages (and thus each property Φ can be recognized by a finite state automaton $\Phi = (A, Q, q_0, Q_F, \delta)$. With each global state Σ of the lattice, we associate a boolean array $B_\Sigma[Q]$. For each state $q \in Q$ of the automaton, element $B_\Sigma[q]$ is true if and only if there exits an observation terminating at Σ whose labeling puts the automaton in state q. A fictitious global state Σ^{-1} is added to the lattice as the unique predecessor of the initial global state Σ^0 and $B_{\Sigma^{-1}}[q_0]$ is the only element of $B_{\Sigma^{-1}}$ that is initially true. Function $pred(\Sigma)$ returns the set of global states that immediately precede Σ in the lattice. The algorithm needs to consider only the global states of two adjacent levels of the lattice maintained in sets *previous* and *current* (the immediate predecessors of a global state in *current* belongs to the set previous). For each state Σ of *current*, the algorithm computes $B_\Sigma[Q]$ from the values associated with $pred(\Sigma)$ and the labeling of Σ.

```
previous := {Σ^-1}
current := {Σ^0}
while (current ≠ ∅) do
    foreach Σ ∈ current do
        foreach q ∈ Q do B_pred[q] := ⋁_{Σ'∈pred(Σ)} B_Σ'[q] od
        foreach α ∈ λ(Σ) do
            B^α[Q] := (false, ···, false)
            foreach (q ∈ Q : B_pred[q] = true) do
                foreach r ∈ δ(q, α) do B^α[r] := true od
            od
        od
        foreach q ∈ Q do B_Σ[q] := ⋁_{α∈λ(Σ)} B^α[q]
    od
    previous := current;
    current := {global states directly reachable from those in previous}
od
```

Figure 10. Algorithm executed by the monitor in order to detect properties on observations.

As with properties on control flows, the satisfaction rules for **SOME** and **ALL** can be easily computed from B_Σ through the following relations

$$\Sigma \models \textbf{SOME}\ \Phi \equiv \exists q \in Q : ((B_\Sigma[q] = true) \wedge q \in Q_F)$$

$$\Sigma \models \textbf{ALL}\ \Phi \equiv \forall q \in Q : ((B_\Sigma[q] = true) \Rightarrow q \in Q_F).$$

The duality relations of Section 4.3 can be applied to global properties in a completely analogous manner.

Note that this general detection algorithm for behavior patterns on observations can be simplified in case the properties of interest are not as expressive as regular languages. For example, Cooper and Marzullo (1991), consider properties on single global states rather than behavior patterns. Babaoğlu and Raynal (1995) consider properties specified through simple sequencing and interval negation. For these special cases, our modal operators **SOME** and **ALL** correspond to **Pos** and **Def** of (Cooper and Marzullo, 1991; Babaoğlu and Raynal, 1995).

6. THE EREBUS DISTRIBUTED DEBUGGER

In the previous sections, we have presented a model of distributed computation, a unified framework to specify dynamic properties of these computations and algorithms to detect properties either on control flows or on observations. This work has been carried out in the context of a project to design and implement a distributed debugging facility called EREBUS (Hurfin et al., 1993a). We now briefly describe this system.

EREBUS is a debugger for distributed programs running on a 32-processor Intel hypercube written in Estelle (Estelle, 1989) an ISO-standard language particularly well suited for protocol descriptions. Estelle is a cross between Pascal and communicating automata: a program is composed of processes that interact through message passing on FIFO channels.

Each Estelle program is an automaton described as a set of transitions from one local state to another local state. Each transition has a guard stating a condition on the local states or on the type and field value of the first message on some input port, and an action block formed by a Pascal-like instructions.

EREBUS consists of three main components:

- An observation tool that, during the first execution of an Estelle program, logs control information in order to be able to replay it later (i.e., to reproduce the same partial order of events).
- The replay tool, which takes the same program and the previously logged control information so as to reproduce an execution equivalent to the first one. Actually this re-execution can be reproduced on the same machine (Intel hypercube), or on a local-area network of workstations or even on a single workstation. This component is similar to the one described in Lebanc and Mellor-Crummey (1987); we consider the message passing model whereas Lebanc and Mellor-Crummey (1987) considers the shared-variable model.
- The final component of EREBUS consists of the run-time detection (either during the initial execution or during a replay) of dynamic properties as described in this article.

The first two components are fully implemented and functional. Some experimental results are described in Hurfin et al. (1993a) and Gerstel et al. (1994). The last component is partially implemented in that we can detect properties on control flows. The main problem with properties on observations is the building of the lattice. Because the size of the lattice can be exponential in k (where k is the maximum number of events produced by a single process), it appears that detection of properties on observations may be infeasible. With this motivation, we have investigated detection of restricted properties on global states that do not require the construction of the lattice (Fromentin and Raynal, 1995).

The lessons learned from the implementation effort of our run-time detection algorithms can be summarized as follows: detection of properties on control flows is feasible while detection of properties on observations appears not to be. Thus, from a practical point of view, we are lead to two possible strategies:

- First, properties should be expressed on control flows, whenever this is possible.
- Second, to be feasible, detection of properties on observations has to be augmented with certain heuristics such that construction of the full lattice can be avoided. In this case, however, the notion of "detection" has to interpreted as "approximate".

EREBUS has been used to detect control flow properties of protocols for implementing RPC-like mechanisms and causally-ordered communication.

7. CONCLUSIONS

We have presented a general framework that allows us to express a large class of properties for distributed computations as languages over alphabets of predicates. We have shown that if the predicates are on local states, the behavior patterns are on control flows of the computation; if the predicates are on global states, the behavior patterns are on observations of the computation. A control flow satisfies a behavior pattern if the sequence of local states defining it satisfies the sequence of local predicates of the pattern. Similarly, an observation satisfies a behavior pattern if the sequence of global states defining this observation satisfies the sequence of global predicates of the pattern. This language-oriented approach to dynamic property detection allows us to understand a large number of existing algorithms as special cases of a generic one.

We have seen that the detection of behavior patterns on control flows can be done at run time and without requiring additional messages; only piggybacking of an array of bits (one for each of the finite state automaton accepting the language associated with the property) is necessary. Thus, detection of these properties can be rather efficient. Detection of behavior patterns on observations, on the other hand, can also be done at run time; however, in its full generality it requires an additional monitor process to construct all possible observations of the computation. As such, detection of such properties can be rather expensive. This is the cost to be paid for being able to specify and detect a larger class of properties. The detection algorithms we have proposed can be seen as on-the-fly model checkers in the sense that they have no a priori knowledge of the model against which to check a property but have a infer it on the fly.

From a practical point of view, run-time detection of dynamic properties is a fundamental problem when one is interested in analyzing or debugging distributed executions. The work presented in this article originated from the EREBUS distributed debugger project (Hurfin et al., 1993a). The algorithms that have been described in this work are currently being implemented within the debugging facility.

8. ACKNOWLEDGEMENTS

We are grateful to M. Hurfin, C. Jard, S. Lorcy, and N. Plouzeau for interesting discussions related to the debugging of distributed programs.

REFERENCES

Babaoğlu, Ö., and Marzullo, K., Consistent global states of distributed systems: fundamental concepts and mechanisms, in *Distributed Systems*, chapter 4 (S. J. Mullender, ed.), ACM Press, Frontier Series, 1993, pp. 55–93.

Babaoğlu, Ö., and Raynal, M., Specification and Detection of Behavioral Patterns in Distributed Computations, *Journal of Parallel and Distributed Computing* 28 (2), (1995).

Bracha, G., and Toueg, S., Distributed Deadlock Detection, *Distributed Computing* 2(3):127–138 (1987).

Clarke, E. M., Emerson, E. A., and Sistla, A. P., Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications, *ACM Toplas*, 8(2):244–263 (1986).

Chandy, K. M., Haas, L. M., and Misra, Jayadev, Distributed Deadlock Detection, *ACM Transactions on Computer Systems*, 1(2):144–156 (May 1983).

Chandy, K. M., and Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems* 3(1):63–75 (February 1985).

Cooper, R., and Marzullo, K., Consistent Detection of Global Predicates, in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 1991, pp. 167–174.

Diehl, C., Jard, C., and Rampon, J. X., Reachability analysis on distributed executions, in *Theory and Practice of Software Development*, (Gaudel and Jouannaud, eds.), TAPSOFT, Springer Verlag, LNCS 668, April 1993, pp. 629–643.

Dijkstra, E. W. D., and Scholten, C. S., Termination Detection for Diffusing Computation, *Information Processing Letters*, 11:217–219 (August 1980).

Estelle: a formal description technique based on an extended state transition model. ISO/TC97/SC21/WG6.1, 1989. ISO 9074.

Francez, N., Distributed Termination, *ACM TOPLAS*, 2(1):42–55 (January 1980).

Fromentin, E., Jard, C., Jourdan, G. V., and Raynal, M., On the Fly Analysis of Distributed Computations, *Information Processing Letters*, 54:267–274 (1995).

Fromentin, E., and Raynal, M., Characterizing and Detecting the Global States Seen by all the Observers of a Distributed Computation, in *Proc. 15th IEEE Int. Conf. on Distributed Computing Systems*, Vancouver, Canada, June 1995, pp. 431–438.

Fromentin, E., Raynal, M., Garg, V. K., and Tomlinson, A. I., On the Fly Testing of Regular Patterns in Distributed Computations, in *Proc. of the 23rd International Conference on Parallel Processing*, St. Charles, IL, August 1994, pp. 73–76.

Garg, V. K., and Waldecker, B., Detection of Unstable Predicates in Distributed Programs, in *Twelfth International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag, LNCS 625, New Delhi, India, December 1992, pp. 253–264.

Gerstel, O., Hurfin, M., Plouzeau, N., Raynal, M., and Zaks, S., On-the-Fly Replay: A Practical Paradigm and its Implementation for Distributed Debugging, in *Proc. 6th IEEE Symposium on Parallel and Distributed Debugging*, Dallas, Texas, October 1994, pp. 266–272.

Harel, David and Pnueli, Amir, On the development of reactive systems, in *Logics and Models of Concurrent Systems* (K. R. Apt, ed.), Springer-Verlag, 1985, pp. 477–498.

Hélary, J.-M., Jard, C., Plouzeau, N., and Raynal, M., Detection of Stable Properties in Distributed Applications, *6th ACM SIGACT-SIGOPS, Symp. Principles of Distributed Computing*, Vancouver, Canada, 1987, pp. 125–136.

Hurfin, M., Plouzeau, N., and Raynal, M., A Debugging Tool for Distributed Estelle Programs, *Journal of Computer Communications*, 16(5):328–333 (May 1993).

Hurfin, M., Plouzeau, N., and Raynal, M., Detecting Atomic Sequences of Predicates in Distributed Computations, in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA, May 1993, pp. 32–42. (Reprinted in SIGPLAN Notices, Dec. 1993).

Jard, C., Jeron, T., Jourdan, G. V., and Rampon, J. X., A General Approach to Trace-Checking in Distributed Computing Systems, in *Proc. 14th IEEE International Conference on DCS*, Poznan, Poland, June 1994, pp. 396–403.

Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558–565 (July 1978).

Lebanc, T., and Mellor-Crummey, J., Debugging Parallel Programs with Instant Replay, *IEEE Transactions on Computers*, C-36(4):471–482 (April 1987).

Miller, B. P., and Choi, J., Breakpoints and Halting in Distributed Programs, in *Proc. 8th IEEE Int. Conf. on Distributed Computing Systems*, San Jose, July 1988, pp. 316–323.

Queille, J.-P., and Sifakis, J., Fairness and Related Properties in Transition Systems—a Temporal Logic to Deal with Fairness, *Acta Informatica*, 19:195–220 (1983).

Schwarz, R., and Mattern, F., Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail, *Distributed Computing*, 7(3):149–174 (1994).

Tomlinson, A. I., and Garg, V. K., Detecting Relational Global Predicates in Distributed Systems, in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA, May 1993, pp. 21–31. (Reprinted in SIGPLAN Notices, Dec. 1993).