JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 28, 173–185 (1995)

Specification and Verification of Dynamic Properties in Distributed Computations[1]

ÖZALP BABAOĞLU*,[2] AND MICHEL RAYNAL†,[3]

*Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, 40127 Bologna, Italy; and †IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

The ability to specify and verify dynamic properties of computations is essential for ascertaining the correctness of distributed applications. In this paper, we consider properties that can be encoded as general Boolean predicates over global system states. We introduce two global predicate classes called *simple sequences* and *interval-constrained sequences* for specifying desirable states in some causality-preserving order along with intervening undesired states. Our formalism is simpler than more traditional proposals and permits concise and intuitive expression of many interesting system properties. Algorithms are given for verifying formulas belonging to these predicate classes in an on-line and observer-independent manner during distributed computations. We illustrate the utility of our results by applying them to examples drawn from program testing, debugging, and dynamic reconfiguration in distributed systems. © 1995 Academic Press, Inc.

1. INTRODUCTION

Detecting when the state of a distributed computation satisfies a certain predicate constitutes a fundamental paradigm in the design of distributed algorithms for critical applications. A large class of problems including error reporting, process control, performance monitoring, decentralized coordination, dynamic reconfiguration, and load balancing can be solved by defining an appropriate set of notification or control actions guarded by the appropriate global predicates that encode critical system properties.

Given that the state of a distributed computation consists of disjoint components corresponding to each of the local process states, no single entity internal to the system can have local access to it. Thus, in its most general form, detection of global predicates must be preceded by a phase in which the global state is constructed. Coping with the uncertainty stemming from communication delays and relative process speeds is the principal source of complexity in applying this methodology.

The problem of detecting predicates defined over a single global state has been extensively studied (see [1] for a survey). The case where the predicate is stable leads to particularly simple and efficient solutions based on distributed snapshots [3]. Informally, a distributed snapshot algorithm gathers and pieces together a collection of local states so as to guarantee that the resulting global state is consistent—one that could have been constructed by an idealized external observer. Being stable, the predicate evaluating to true in the snapshot state is sufficient for concluding that it has been "detected."

Detection of non-stable predicates cannot be based on snapshots, even when they are applied repeatedly [15]. No matter how frequently taken, a sequence of snapshots may have gaps that correspond to exactly those global states in which the (nonstable) predicate holds. Any reasonable definition of "detection" for the case of nonstable predicates must be based on *observations* [21, 5, 14, 9, 1]. A further complexity in detecting non-stable predicates is due to the so-called "relativistic effect" which results in multiple observations for the same computation. Modal operators have been proposed so as to make detection of nonstable predicates independent of any particular observation [5, 14, 9].

While predicates (stable or not) over a single global state are able to capture many interesting system properties, they inherently lack notions of logical time or relative order. In order to characterize *dynamic properties* and *behavioral patterns* of distributed computations, simple predicate specifications must be extended to include a temporal component [2, 16, 21, 12]. In other words, specifying and verifying dynamic properties require reasoning about sequences, rather than single instances of global states.

In this paper we consider dynamic property specifications that admit arbitrary predicates over global states as building blocks. We introduce the syntactic classes *SS* (simple sequence) and *ICS* (interval-constrained sequence) of global predicates that define sets of global states related through the notion of causality-preserving sequencing [13]. Our formalism can be viewed as an application of branching time temporal logics [7, 4] to distributed computations. This relationship is further explored in Section 5. The syntax and semantics of class *ICS* is tailored for specifying and verifying properties that are of interest to

[1] A preliminary version of this paper appeared in *Proc. Fourth IFIP Working Conference on Dependable Computing for Critical Applications*, San Diego, California, Jan. 1994 under the title "Specification and Verification of Behavioral Patterns in Distributed Computations."

[2] E-mail: ozalp@cs.unibo.it.

[3] E-mail: raynal@irisa.fr.

0743-7315/95 $12.00

distributed computing. The main contribution of our work lies in the development of procedures for on-line verifying the satisfaction of formulas written in *ICS* by an underlying computation. The verification takes place at a monitor internal to the system and concurrently with the actual computation. As such, our detection algorithms can be viewed as on-line versions of model checkers in various logic formalisms.

2. SYSTEM MODEL

2.1. Asynchronous Distributed Systems

We adopt the terminology and notation of [1]. A distributed system is a finite collection of sequential processes $p_1, p_2, ..., p_n$ that communicate by exchanging messages. Without loss of generality, we assume that communication is reliable and that it incurs finite but arbitrary delays. We make no assumptions about the order in which messages are received with respect to the order in which they are sent.

Processes do not share state and do not have access to a global clock. Furthermore, no bounds exist on the relative speeds of processes. The system thus described corresponds to the well-known *asynchronous* model.

2.2. Distributed Computations

Informally, a distributed computation describes a single execution of a distributed program by a collection of processes. The activity of each sequential process is modeled as a sequence of *events*. An event may be either internal to a process and cause only a local state change, or it may involve communication with another process by sending or receiving a message. A large portion of the internal events at a process may be irrelevant with respect to a given property. Thus, we implicitly consider only those events that are "relevant" in the sense that they can affect the system property under consideration [5]. By definition, communication events are always relevant.

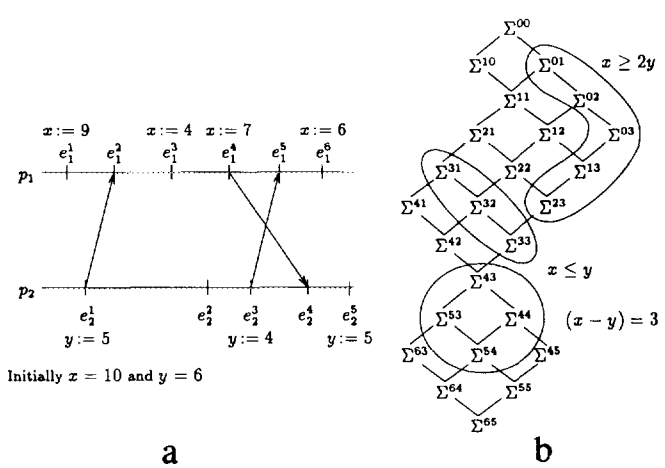

FIG. 1. (a) Space-time diagram representation of a distributed computation. (b) Lattice of consistent global states.

The *local history* of process p_i during the computation is a sequence of events $h_i = e_i^1 e_i^2 ....$ The labeling of the events of process p_i is such that e_i^1 is the first event executed, e_i^2 is the second event executed, etc. Let $h_i^k = e_i^1 e_i^2 ... e_i^k$ denote an initial prefix of local history h_i containing the first k events. We define h_i^0 to be the empty sequence. The *global history* of the computation is the set H containing all events that are executed.

Formally, a *distributed computation* is a partially ordered set γ defined by the pair $(H, \rightarrow)$ where $\rightarrow$ is the binary causal precedence relation defined on events [13]. It is common to depict distributed computations using an equivalent graphical representation called a *space–time diagram*, as shown in Fig. 1a.

2.3. Global States, Observations, and Lattice Structures

Let σ_i^k denote the local state of process p_i immediately after having executed event e_i^k and let σ_i^0 be its initial state before any events are executed. The *global state* of a distributed computation is an n-tuple of local states $\Sigma = (\sigma_1^{c_1}, ..., \sigma_n^{c_n})$, one for each process. Implicitly, global state Σ defines a *cut* of the global history as the subset $C = h_1^{c_1} \cup \cdots \cup h_n^{c_n}$ containing all events whose effects are reflected in Σ.

A global state is *consistent* if the cut associated with it is left closed under the causal precedence relation. In other words, global state Σ of computation γ is consistent if the cut C associated with Σ is such that for all events e and e' of the global history H we have $(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$. Intuitively, consistent global states correspond exactly to those that could be constructed by an idealized observer external to the computation.

Excluding the possibility of simultaneous events, an actual execution of the distributed program results in a total ordering of γ called a *run*. Clearly, only an idealized omniscient observer could know the run associated with the computation. While asynchronous systems do not admit such an idealized observer, they do admit internal perceptions of an execution consistent with its run. Any total order of γ that is consistent with causal precedence constitutes a *sequential observation* (observation for short) of the computation. In other words, observations are linearizations of the partial order defined by causal precedence and can be constructed using mechanisms that are entirely internal to the system (thus subject to all the uncertainties).

An observation can be expressed as a sequence of consistent global states $\Sigma^0 \Sigma^1 \Sigma^2 ...$ where Σ^0 denotes the initial global state $(\sigma_1^0, ..., \sigma_n^0)$ and each global state Σ^i is obtained from the previous state Σ^{i-1} by some process executing a single event in γ. For two such global states of observation Ω, we say that Σ^{i-1} *leads to* Σ^i *in* Ω. Let $\rightsquigarrow_\Omega$ denote the transitive closure of the *leads-to* relation in a given observation Ω. We say that Σ' is *reachable from* Σ *in observation* Ω if and only if $\Sigma \rightsquigarrow_\Omega \Sigma'$. We drop the subscript on the relation if there exists *some* observation of the computation in which Σ' is reachable from Σ.

The set of all consistent global states of a computation along with the *leads-to* relation define a *lattice*. The lattice consists of n distinct axes, one for each process in the computation. Let $\Sigma^{k_1 \cdots k_n}$ be a shorthand for the global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$ and let $k_1 + \cdots + k_n$ be its *level*. Thus, the level of a global state represents the total number of events necessary to produce it. Figure 1b illustrates the lattice of global states associated with the distributed computation of Fig. 1a. A path in the lattice is a sequence of global states of increasing level where the level between any two successive elements differs by one. By construction, each such path corresponds to an observation and each observation has a corresponding path in the lattice. Thus, the lattice of global states effectively represents the set of all possible observations of the computation. Algorithms for on-line construction of the lattice can be found in [5, 14, 1]. In [6], Diehl *et al.* present an interesting alternative to the level-by-level construction of the lattice; their approach uses a greedy strategy that allows extending the lattice multiple levels at a time.

3. GLOBAL PREDICATES

Global predicate classes can be established by providing syntax rules that define the set of well-formed formulas and describing the semantics associated with them. In other words, global predicate classes can be viewed as small languages for specifying system properties while detecting a global predicate can be viewed as checking a formula in a model, namely, a distributed computation. As noted earlier, these formulas can be seen as an application of temporal logics to the analysis of distributed programs. They are tailored for the specification of properties that are of interest for distributed computing. Section 4 will show how to detect them in an on-line manner.

3.1. Simple Predicates

Syntactically, the simplest global predicate class we consider is called *SP*, for *simple predicates*, and corresponds to general Boolean expressions defined over a single global state. Formulas of *SP* may reference any state variable of any process, the constants *true* and *false*, and follow the usual syntax rules for Boolean expressions. We let $\varphi(\Sigma)$ denote the value of predicate φ when evaluated in global state Σ. Suppose the annotated Fig. 1a corresponds to a distributed computation where two variables x and y are maintained by two processes p_1 and p_2, respectively. Figure 1b illustrates three simple predicates defined over the global states of this computation, which can be expressed as pairs of (x, y) values, and identifies regions of the lattice satisfying each of them.

Note that the global predicates of [9, 12] belong to a class $SP' \subset SP$ where Boolean expressions are limited to those that can be expressed as conjunctions or disjunctions of local predicates (those naming state variables of a single process only). There are many interesting system properties, including the following, that cannot be expressed in this restricted class SP' but have compact and natural expressions in *SP*:

1. Load in the network is balanced.
2. Resource allocation violates the "at most k-out-of-n" concurrent accesses requirement.
3. Message delay along the route from A to B is less than 5 ms.
4. No more than 100 total users logged in to machines A, B and C.
5. In the computation of Fig. 1, $x \geq 2y$.

While it is not possible to express these properties in their general case as conjunctions or disjunctions of local predicates, limiting the domain of state variables to discrete values may permit solutions in SP' through "brute force" enumeration. As an example, consider the specification for the violation condition of the k-out-of-n resource allocation requirement. Each process maintains a local variable R_i which is set to one while it is holding the resource, and zero at other times. A global configuration of the system is the conjunction of the R_i values, one for each of the n processes. The desired property can then be expressed by forming the disjunction among all configurations that contain $0, 1, \ldots, k$ ones and then complementing it. Clearly, the length of the resulting formula is exponential in k and the approach becomes impractical beyond small values. The same specification in the class *SP* has the obvious compact formulation as the predicate $\varphi = (R_1 + R_2 + \cdots + R_n > k)$. With respect to observations, the semantics of the class *SP* is defined as follows:

DEFINITION 1. Let φ be a predicate in *SP*. Observation Ω *satisfies* φ, denoted $\Omega \vDash \varphi$, if and only if there exists global state $\Sigma \in \Omega$ such that $\varphi(\Sigma)$ is true.

Stable predicates are members of the syntactic class *SP* with the following additional semantic requirement:

DEFINITION 2. Simple predicate φ is *stable* in computation γ if and only if

$$\forall \Sigma, \Sigma' \in \gamma : \varphi(\Sigma) \wedge (\Sigma \rightsquigarrow \Sigma') \Rightarrow \varphi(\Sigma').$$

In other words, if a stable predicate is true in some global state of the computation, then it must be true in all global states reachable from it. This explains why stable predicates can be detected in an observer-independent manner using snapshots: the snapshot is a consistent global state and thus belongs to at least one observation. For all finite computations, its run and the observation including the snapshot must have at least one global state in common. Thus, if the predicate is found to hold in the snapshot, we can declare that it is detected since the run must also satisfy it.

Detection of nonstable predicates poses problems even if we base them on observations rather than snapshots. It could be that the observation satisfies the predicate but

the run does not, or vice versa. The predicate $\varphi = (x \geq 2y)$ of Fig. 1b illustrates the problem. As there is a path in the lattice not including any of the states satisfying φ, a single observation is not sufficient for us to conclude anything regarding the satisfaction or not of φ by the run.

To make detection of non-stable predicates observer independent, Cooper and Marzullo have proposed that the class *SP* be augmented with two modal operators [5]. We adopt their proposal in order to define the semantics of the class *SP* with respect to computations.

DEFINITION 3.

1. Distributed computation γ *satisfies* **Pos** φ, denoted $\gamma \vDash$ **Pos** φ, if and only if there exists an observation Ω of γ such that $\Omega \vDash \varphi$.
2. Distributed computation γ *satisfies* **Def** φ, denoted $\gamma \vDash$ **Def** φ, if and only if for all observations Ω of γ it is the case that $\Omega \vDash \varphi$.

Note that the above definitions of **Pos** and **Def** have been called *weak* and *strong* formulas, respectively, by Garg and Waldecker in [9]. It is easy to show that if φ is a stable predicate of computation γ, then $(\gamma \vDash$ **Pos** $\varphi) \equiv (\gamma \vDash$ **Def** $\varphi)$.

3.2. Simple Sequences

As noted earlier, simple global predicates can effectively capture static properties of systems but are unable to capture dynamic ones. In order to include dynamic aspects of distributed computations in specifications, global predicates must include a temporal component. One such proposal is due to Miller and Choi [16] who introduce the notion of *linked predicates* to add causal sequencing to a set of local predicates.

We extend this idea by considering sequences of global predicates by composing instances of the class *SP*. The first extension we consider is called *SS*, for *simple sequences*, and is defined by the syntax rule

$$SS ::= SP \mid SS; SP$$

where *SP* is as defined in the previous Section. In other words, formulas in *SS* are semicolon-separated sequences of simple predicates, each defined over a single global state. Referring to Fig. 1b, the global predicate $(x \geq 2y); (x \leq y); (x - y = 3)$ is an instance of *SS*. Obviously, the class *SS* includes *SP* as sequences of length one.

The semantics of *SS* with respect to observations is as follows:

DEFINITION 4. Let $\Phi = \varphi_1; \varphi_2; \ldots; \varphi_m$ be a formula in *SS*. Observation Ω *satisfies* Φ, denoted $\Omega \vDash \Phi$, if and only if there exist m distinct global states $\Sigma^{i_1} \ldots \Sigma^{i_m}$ of Ω such that

1. $\Sigma^{i_1} \rightsquigarrow_\Omega \Sigma^{i_2} \rightsquigarrow_\Omega \cdots \rightsquigarrow_\Omega \Sigma^{i_m}$ and
2. $\varphi_1(\Sigma^{i_1}) \wedge \varphi_2(\Sigma^{i_2}) \wedge \cdots \wedge \varphi_m(\Sigma^{i_m})$.

In other words, for the formula to be satisfied by observation Ω, each of the component predicates must hold in distinct global states of Ω, and each such global state must be observed in an order consistent with the syntactic position of its respective component predicate. Note that the semantics of the expression

$$\Omega \vDash \varphi_1 \wedge \Omega \vDash \varphi_2 \wedge \cdots \wedge \Omega \vDash \varphi_m$$

is not equivalent to that of

$$\Omega \vDash \varphi_1; \varphi_2; \ldots; \varphi_m,$$

since the former is unable to capture the relative ordering property.[4]

The semantics of *SS* with respect to computations remain exactly as given in Definition 3 with the introduction of modal operators **Pos** and **Def** and the above semantics with respect to observations. For example, the following formulas are all true for γ corresponding to the computation of Fig. 1a:

$$\gamma \vDash \textbf{Pos}\,(x \geq 2y); (x \leq y); (x - y = 3)$$

$$\gamma \vDash \textbf{Def}\,(x \leq y); (x - y = 3)$$

$$\neg(\gamma \vDash \textbf{Pos}\,(x \leq y); (x \geq 2y); (x - y = 3)).$$

3.3. Negation of Predicates with Respect to Observations

Negation of a simple predicate φ with respect to a global state Σ has the obvious natural interpretation: $\neg\varphi$ holds in Σ if and only if φ does not hold in Σ. This definition, which we call *state negation*, can be extended to negation with respect to observations in one of several ways. We adopt the following interpretation since it captures our intuition for negating predicates when they are interpreted over sequences.

DEFINITION 5. Let Φ be a predicate. The *interval negation* of Φ with respect to observation Ω, denoted $\overline{[\Phi]}$, is a predicate such that $\Omega \vDash \overline{[\Phi]}$ if and only if $\neg(\Omega \vDash \Phi)$.

Note that the negation is applied to the "satisfies" relation and not to the predicate itself. As a consequence, interval negation of a predicate is defined over sequences of global states even when the predicate is defined over a single state. For instance, consider $\Phi = \varphi$ for some φ belonging to the class *SP*. By Definition 1, we have $\Omega \vDash \varphi$ if and only if there exists some global state of Ω in which φ holds. Thus, for $\Omega \vDash \overline{[\varphi]}$ to hold, our interpretation requires $\neg\varphi$ to hold in *all* global states of observation Ω.[5]

Our notation $\overline{[\Phi]}$ for denoting the interval negation of

[4] The two formulas specify without order and with order, respectively, satisfaction of a set of simple predicates by an observation. They differ from yet another formula $\Omega \vDash \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_m$ that is true if the observation contains at least one global state in which all m predicates φ_i hold simultaneously.

[5] Contrast this with the alternative definition $\Omega \vDash \overline{[\varphi]} \equiv \Omega \vDash \neg\varphi$ which would require Ω to contain at least one state where $\neg\varphi$ holds. This, however, would have the unfortunate consequence that $\Omega \vDash \varphi$ and $\Omega \vDash \overline{[\varphi]}$ could both be true for the same observation.

global predicate Φ suggests the fact that it is defined over an interval. Implicitly, the interval in question is the entire observation. Note that for Φ belonging to *SS*, the construct $\neg\Phi$ is meaningless while $\overline{[\Phi]}$ is well defined. For Φ belonging to *SP*, we reserve the notation $\neg\Phi$ to denote state negation as opposed to interval negation.

From Definition 5, it is clear that interval negation is its own inverse. In other words, $\Omega \models [\overline{[\Phi]}] \equiv \Omega \models \Phi$.

3.4. Interval-Constrained Sequences

There are cases where the dynamic property specification must describe not only sequences of desired states, it must also describe undesirable states during intervals [12, 20]. For example, to demonstrate the "low contention" property of a resource management strategy, we might be interested in verifying that during intervals while the resource is requested and acquired by a particular process, the total number of outstanding requests does not exceed some threshold. Note that the intervals of interest with respect to the undesired states are dynamic and cannot be established a priori. An interval is defined dynamically whenever two adjacent predicates are satisfied by two global states in the correct order.

The class *SP* extended with interval negation as discussed in the previous section is sufficient for specifying undesired states over entire observations. What is lacking is the possibility to restrict negation dynamically to subintervals of an observation. We can achieve this by composing predicates from *SP* extended with interval negation to define *interval-constrained negation* as follows. We denote the beginning and end of an interval through simple predicates written to the left and right, respectively, of the negation. Thus, the interval-constrained negation $\varphi_1; \overline{[\theta]}\varphi_2$ is satisfied by an observation if there exists a global state Σ^1 in which φ_1 holds, a later global state Σ^2 in which φ_2 holds, and θ does not hold in the global states in between. Note that Σ^1 and Σ^2 delimit the interval but do not belong to it. Thus, the interval can be empty if Σ^1 directly leads to Σ^2. Moreover, one or both of φ_1 and φ_2 may be missing, in which case the interval is implicitly extended to the initial state or to the final state of the observation, respectively.

We formalize these ideas by defining a new global predicate class called *ICS*, for *interval-constrained sequence*, with the following syntax (for $m \geq 0$):

$$\overline{[\theta_1]}\varphi_1; \ldots; \overline{[\theta_m]}\varphi_m; \overline{[\theta_{m+1}]}.$$

Note that the class *ICS* includes *SS* and (unconstrained) interval negation as special cases. Again referring to Fig. 1, the following global predicates are all instances of *ICS*:

$$(x \geq 2y); (x \leq y); (x - y = 3)$$

$$\overline{[x - y = 3]}$$

$$\overline{[x \geq 2y]}\,(x \leq y); \overline{[true]}\,(x - y = 3)$$

$$(x \geq 2y); \overline{[true]}\,(x \leq y); \overline{[x - y \neq 3]}.$$

In defining the semantics of *ICS* with respect to observations, we consider the most general formula belonging to this class:

DEFINITION 6. Let $\Phi = \overline{[\theta_1]}\varphi_1; \ldots; \overline{[\theta_m]}\varphi_m; \overline{[\theta_{m+1}]}$ be a formula in *ICS*. If any of the θ_i in Φ are missing, we assume they are defined as the constant *false*. Observation Ω *satisfies* Φ, denoted $\Omega \models \Phi$, if and only if

1. If $m = 0$, then $\forall\Sigma \in \Omega$: $\neg\theta_1(\Sigma)$;
2. If $m \neq 0$, then there exist m distinct global states $\Sigma^{i_1} \ldots \Sigma^{i_m}$ of Ω such that
   (a) $\Sigma^{i_1} \rightsquigarrow_\Omega \Sigma^{i_2} \rightsquigarrow_\Omega \cdots \rightsquigarrow_\Omega \Sigma^{i_m}$,
   (b) $\varphi_1(\Sigma^{i_1}) \wedge \cdots \wedge \varphi_m(\Sigma^{i_m})$,
   (c) $\forall i : i < i_1$: $\neg\theta_1(\Sigma^i)$,
   (d) $k = 2, \ldots, m$: $\forall i$: $i_{k-1} < i < i_k$: $\neg\theta_k(\Sigma^i)$ and
   (e) $\forall i : i > i_m$: $\neg\theta_{m+1}(\Sigma^i)$.

With respect to the desired sequence, the semantics of *ICS* is the same as that of *SS*. For the class *ICS*, however, there is the additional requirement that between each pair of global states satisfying predicates φ_{i-1} and φ_i, there can be no intermediate states where θ_i holds. Furthermore, the prefix of the observation up to the state satisfying φ_1 must not include any state satisfying θ_1 and the suffix of the observation beyond the state satisfying φ_m must not include any state satisfying θ_{m+1}.

The class *ICS* happens to be quite expressive and can be used to specify many dynamic properties that are of interest in distributed systems. In particular, we can easily express notions such as atomicity, stability and invariance. For instance, in terms of observations, atomicity of distributed actions can be expressed as transforming one global state to another with no intermediate global states that are observable. In other words, the state in which the atomic action is applied and the resulting state must be adjacent to each other. In terms of *ICS* formulas, states that constrain the interval $\overline{[true]}$ must be adjacent since only the empty sequence satisfies $\overline{[true]}$. By inserting empty interval requirements at selected points in an *ICS* formula, we can express interesting specifications. Here are some *ICS* formulas and their meaning:

- $\varphi_1; \overline{[true]}\varphi_2 \equiv$ Predicates φ_1 and φ_2 hold in adjacent global states, in that order.
- $\overline{[true]}\varphi \equiv$ Predicate φ holds in the initial global state.
- $\varphi; \overline{[true]} \equiv$ Predicate φ holds in the final global state.
- $\overline{[\varphi]} \equiv$ Predicate $\neg\varphi$ is an invariant for the observation.

As before, we define the semantics of *ICS* with respect to computations by augmenting the class with the modal operators **Pos** and **Def**. The resulting semantics remain exactly as given in Definition 3 with the appropriate interpretation of satisfaction with respect to observations. For γ corresponding to the computation of Fig. 1a, the following formulas are all true:

$$\gamma \models \textbf{Pos}\,\overline{[x \geq 2y]}\,(x \leq y); \overline{[true]}\,(x - y = 3)$$

$$\gamma \models \textbf{Pos}\,(x \geq 2y); \overline{[true]}\,(x \leq y); \overline{[true]}\,(x - y = 3)$$

$$\gamma \vDash \textbf{Def}\,(x \le y);\overline{[x \ge 2y]}\,(x - y = 3)$$

$$\gamma \vDash \textbf{Def}\,\overline{[true]}\,(x = 10 \wedge y = 6);(x \le y);(x - y = 3)$$

$$\neg(\gamma \vDash \textbf{Pos}\,(x \ge 2y);\overline{[x \le y]}\,(x - y = 3)).$$

The class *ICS* augmented with modal operators allows us to restate the stability property of simple predicates very concisely.

DEFINITION 7. Simple predicate φ is *stable* in computation γ if and only if $\gamma \vDash \textbf{Def}\,\overline{[\varphi]}\varphi;\overline{[\neg\varphi]}$.

In other words, every observation of the computation consists of two segments—an initial prefix where the predicate is false, followed by a (nonempty) suffix where the predicate is true. Unlike Definition 2, this definition is not trivially satisfied by predicates that never hold during the computation.

Relaxing the requirement that the predicate be false in the prefix up to the first state where it becomes true results in an interesting alternative to the above notion for stability. Consider the following definition.

DEFINITION 8. Simple predicate φ is *eventually stable* in computation γ if and only if $\gamma \vDash \textbf{Def}\,\varphi;\overline{[\neg\varphi]}$.

In this case, all observations must contain a nonempty suffix where the predicate is true, but need not contain a prefix where it is false. Thus, the predicate may be initially nonstable but it eventually "stabilizes."

3.5. Duality of Modal Operators

As noted in [1], modal operators **Pos** and **Def** are not duals of each other under state negation (i.e., with the syntax $\neg\varphi$). In other words, $\neg(\gamma \vDash \textbf{Def}\,\varphi)$ is not equivalent to $\gamma \vDash \textbf{Pos}\,\neg\varphi$ and $\neg(\gamma \vDash \textbf{Pos}\,\varphi)$ is not equivalent to $\gamma \vDash \textbf{Def}\,\neg\varphi$. As an example, consider γ corresponding to the distributed computation of Fig. 1a where $\gamma \vDash \textbf{Def}\,(x \le y)$ and $\gamma \vDash \textbf{Pos}\,(x > y)$ are both true.

Under interval negation, which is on sequences of global states (i.e., with the syntax $\overline{[\Phi]}$), the model operators **Def** and **Pos** are indeed duals. Note that interval negation as given in Definition 5 remains valid also for our new class *ICS*. Thus, the following duality property is valid for all three global predicate classes *SP*, *SS*, and *ICS*.

PROPERTY 1. *If Φ is a formula in SP, SS, or ICS, then for any computation γ, we have*

$$\gamma \vDash \textbf{Def}\,\overline{[\Phi]} \equiv \neg(\gamma \vDash \textbf{Pos}\,\Phi).$$

Proof. By Definition 3, $\gamma \vDash \textbf{Def}\,\overline{[\Phi]} = \forall\Omega \in \gamma\colon \Omega \vDash \overline{[\Phi]}$. By Definition 5, $\Omega \vDash \overline{[\Phi]} = \neg(\Omega \vDash \Phi)$. By predicate calculus, $\forall\Omega \in \gamma\colon \neg(\Omega \vDash \Phi) = (\exists\Omega \in \gamma\colon \Omega \vDash \Phi)$. By Definition 3, we see that this last term is exactly $\neg\,(\gamma \vDash \textbf{Pos}\,\Phi)$. ■

An immediate consequence of this result is that we need to develop detection algorithms only for formulas in *SS* and *ICS*, without negation. Formulas in these classes extended with interval negation can be detected indirectly through the duality property.

4. DETECTION ALGORITHMS

In this section, we give algorithms for verifying if formulas belonging to the classes *SS* and *ICS* are satisfied by a distributed computation. Our algorithms operate on-line in the sense that they construct their results by monitoring the computation as it evolves. They are no more intrusive than the algorithms used for constructing the lattice structure corresponding to the underlying computation. Any of the algorithms in [5, 14, 6] can be used for the on-line construction of the lattice used by our detection algorithms. Our algorithms need to consider only two adjacent levels of the lattice at any given time. Thus, the overhead in space is only marginally greater than that of the algorithms used for the lattice construction.

4.1. Detection of Simple Sequences

The algorithm shown in Fig. 2 detects if **Def** Φ is satisfied during a computation where $\Phi = \varphi_1; \varphi_2; \ldots; \varphi_m$ is a formula of length m belonging to the class *SS*. The algorithm bases its computation on the states of the lattice described in Section 2.3. We do not include the steps necessary for the construction of these states. Techniques based on notification messages of relevant events with vector clock timestamps can be used to construct relevant portions of the lattice on-line with the computation [5, 14, 1].

For notational convenience, we assume that a fictitious global state Σ^{-1} precedes the initial state Σ^0. The function $pred(\Sigma)$ returns a set of global states corresponding to the immediate predecessors of Σ. In other words, each member of $pred(\Sigma)$ leads to Σ. By definition, $pred(\Sigma^0) = \{\Sigma^{-1}\}$. The predecessor of Σ^{-1} is undefined.

With each state Σ of the lattice, we associate an integer variable $prefix(\Sigma)$. Initially, only $prefix(\Sigma^{-1})$ is defined and is zero. In general, $prefix(\Sigma) = u$ indicates that all observations starting with the initial global state Σ^0 and ending at Σ satisfy a maximal prefix of length u of the formula Φ.

The following Lemma formalizes the semantics of variable *prefix*.

LEMMA 1. *In algorithm* Detect_SS(Φ), *the variable prefix*(Σ) *associated with each global state* Σ *is assigned the value* u *if and only if* u *is the largest integer such that all observations starting in global state* Σ^{-1} *and ending in global state* Σ *satisfy the formula* $\varphi_0; \varphi_1; \ldots \varphi_u$ *where* φ_0 *is a (fictitious) predicate that holds only in the global state* Σ^{-1}.

Proof. By induction of ℓ, the levels of the lattice.

Base case ($\ell = -1$): The only global state to be considered at level -1 is the fictitious state Σ^{-1}. Thus, the only observation to be considered consists of the single element Σ^{-1}. In line (2) of the algorithm, the variable $prefix(\Sigma^{-1})$

```
    function Detect_SS(Φ);
    var previous, current, verified: set of global states;
        prefix, u: integer;
    begin
1       previous := {Σ^-1};
2       prefix(Σ^-1) := 0;
        repeat
3           verified := {};
4           current :={global states directly reachable from those in previous};
5           foreach Σ ∈ current do
6               u := min_{Σ'∈pred(Σ)} (prefix(Σ'));
7               if φ_{u+1}(Σ) then prefix(Σ) := u + 1
8               else prefix(Σ) := u;
                fi
9               if (prefix(Σ) = m) then verified := verified ∪ {Σ} fi
            od
10          if ((verified ≠ {}) ∧ (verified = current)) then return (true) fi
11          previous := current − verified;
12      until (previous = {});
13      return (false)
    end
```

FIG. 2. Algorithm for detecting global predicates in the class *SS*.

is set to zero. By definition, $\varphi_0(\Sigma^{-1})$ is true and since an observation of length one cannot satisfy a formula with more than one term, φ_0 is indeed the maximal formula satisfiable at this level.

Induction step: Assume that the Lemma holds for each of the global states at some level $\ell \geq -1$. Consider global state Σ at level $\ell + 1$ of the lattice. Let Σ' be a global state of level ℓ that leads to Σ and let $prefix(\Sigma') = u'$. By the hypothesis, each observation ending in state Σ' satisfies the formula $\varphi_0; \varphi_1; \ldots \varphi_{u'}$. In line (6) of the algorithm, u is defined to be the minimum among all such u'. Thus, every observation Ω' ending in a global state of level ℓ that leads to Σ satisfies the formula $\varphi_0; \varphi_1; \ldots \varphi_u$. For each observation Ω', let Ω be its extension obtained by adding global state Σ. By construction, every such Ω will satisfy the formula $\varphi_0; \varphi_1; \ldots \varphi_{u+1}$ if $\varphi_{u+1}(\Sigma)$ holds and satisfy the formula $\varphi_0; \varphi_1; \ldots \varphi_u$ otherwise. These two cases correspond exactly to the $prefix(\Sigma)$ values determined by lines (7) and (8), respectively, of the algorithm. ■

PROPERTY 2. *Given a formula* $\Phi = \varphi_1; \varphi_2; \ldots \varphi_m$ *belonging to the class SS, function* Detect_SS(Φ) *returns true for computation* γ *if and only if* $\gamma \vDash$ **def** Φ.

Proof. The algorithm uses three variables *previous*, *current*, and *verified*, each a set of global states. The test for satisfaction of the formula proceeds by levels of the lattice structure. At each step, the set *current* is defined as the states directly reachable from those in set *previous*. In other words, *previous* and *current* define sets of adjacent states in the lattice. The set *previous* is initialized to contain the single (fictitious) state Σ^{-1}. For each state Σ in *current*, the algorithm next computes the value u for $prefix(\Sigma)$. By Lemma 1, we know that all observations starting in state Σ^{-1} and ending in state Σ satisfy the formula $\varphi_0; \varphi_1; \ldots \varphi_u$ where φ_0 is the fictitious predicate that holds only in state Σ^{-1}. Clearly, all observations starting in state Σ^0 and ending in state Σ must satisfy the original formula $\varphi_1; \ldots \varphi_u$. If the entire formula is satisfied up to Σ (i.e., $prefix(\Sigma) = m$), then Σ is added to the set *verified* in line (7). Note that formulas in *SS* are monotonic with respect to length of observations in the sense that if a formula is satisfied by an observation of length ℓ, it can never be invalidated by extending the observation beyond length ℓ. This fact allows us to exclude from all future consideration that portion of the lattice reachable from a state $\Sigma \in$ *verified*. This, in turn, is accomplished by deleting Σ from the set *current* when the level of the lattice is advanced in line (11). If the entire formula is satisfied by all remaining states of the current level, then in line (10) we claim that **Def** Φ is satisfied by the computation. If the prefix verified is partial, we continue with the next level of the lattice, unless there are none, in which case we claim that **Def** Φ is not satisfied by the computation. ■

Verifying that a computation satisfies **Pos** Φ rather than **Def** Φ requires two simple syntactic modifications to algorithm *Detect_SS*(Φ).

PROPERTY 3. *Given a formula* $\Phi = \varphi_1; \varphi_2; \ldots \varphi_m$ *belonging to the class SS, algorithm* Detect_SS(Φ), *with the following modifications, returns true for computation* γ *if and only if* $\gamma \vDash$ **Pos** Φ:

1. *The* min *operator of line (6) is changed to* max,
2. *The test in line (10) is changed to* "**if** ((*verified* ≠ {}) **then return** *(true)* **fi**."

The proof is essentially the same as that of Property 2 with the following lemma defining the new meaning for variable *prefix* replacing Lemma 1.

LEMMA 2. *In algorithm* Detect_SS(Φ), *with the modifications of Property 3, the variable prefix*(Σ) *associated with each global state* Σ *is assigned the value* u *if and only if* u *is the largest integer such that at least one observation starting in global state* Σ^{-1} *and ending in global state* Σ *satisfies the formula* $\varphi_0; \varphi_1; \ldots \varphi_u$ *where* φ_0 *is a (fictitious) predicate that holds only in the global state* Σ^{-1}.

4.2. Detection of Interval-Constrained Sequences

The algorithm shown in Fig. 3 detects **Pos** Φ for formulas belonging to the class *ICS*. Unlike the class *SS*, however, formulas in *ICS* are not monotonic with respect to the length of observations (levels of the lattice). In other words, a prefix of the formula terminated with $\overline{[\theta_u]}$ that is satisfied by an observation of length ℓ may cease to be satisfied when the observation is extended to length $\ell + 1$ since it may be impossible to avoid future undesired states (those in which θ_u holds). For this reason, we introduce an array *path*(Σ) of $m + 1$ Boolean values associated with each state Σ. Informally, this array records those prefixes of formula Φ that can be satisfied by at least one observation starting with the initial state Σ^0 and ending at state Σ. The array associated with (fictitious) state Σ^{-1} is initialized to true for the prefix of length zero and false for all others.

The following lemma formalizes the semantics of array *path*.

LEMMA 3. *In algorithm* Detect_ICS(Φ), *the array path*(Σ) *associated with each global state* Σ *is assigned the value true at index* u *if and only if at least one observation starting in global state* Σ^{-1} *and ending in global state* Σ *satisfies the formula* $\varphi_0; \overline{[\theta_1]}\varphi_1; \ldots \overline{[\theta_u]}\varphi_u; \overline{[\theta_{u+1}]}$ *where* φ_0 *is a (fictitious) predicate that holds only in the global state* Σ^{-1}.

Proof. By induction on ℓ, the levels of the lattice.

Base case ($\ell = -1$): The only global state to be considered at level -1 is the fictitious state Σ^{-1}. Thus, the only observation to be considered consists of the single element Σ^{-1}. In lines (2) and (3) of the algorithm, the array *path* (Σ^{-1}) is set to *true* only at index zero. By definition, φ_0 holds in state Σ^{-1} and $\overline{[\theta_1]}$ is true when applied to an empty interval. Since φ_i terms of the formula need to hold in distinct global states, an observation of length one cannot satisfy a formula with more than one such term. Thus $\varphi_0; \overline{[\theta_1]}$ is indeed the maximal formula satisfiable at this level.

Induction step: Assume that the Lemma holds up to some level $\ell \geq -1$. Consider global state Σ at level $\ell + 1$ of the lattice. Let Σ' be a global state of level ℓ that leads

```
    function Detect_ICS(Φ);
    var previous, current, verified: set of global states;
        path, p_path: array[0..m] of boolean;
        detected: boolean;
        u: integer;
    begin
1       previous := {Σ^-1};
2       path(Σ^-1)[0] := true;
3       for u := 1 to m do path(Σ^-1)[u] := false od
4       verified := {};
5       current := {Σ^0};
6       while (current ≠ {}) do
7           foreach Σ ∈ current do
8               for u := 0 to m do p_path[u] := ⋁_{Σ'∈pred(Σ)} path(Σ')[u] od
9               path(Σ)[0] := p_path[0] ∧ ¬θ_1(Σ);
10              for u := 1 to m do
11                  path(Σ)[u] := (p_path[u] ∧ ¬θ_{u+1}(Σ)) ∨ (p_path[u − 1] ∧ φ_u(Σ))
                od
12              if path(Σ)[m] then verified := verified ∪ {Σ} fi
            od
13          if verified ≠ {} then detected := true else detected := false fi
14          previous := current;
15          current := {global states directly reachable from those in previous};
16          verified := {};
        od
17      return (detected)
    end
```

FIG. 3. Algorithm for detecting global predicates in the class *ICS*.

to Σ and let $path(\Sigma')[u] = true$. By the hypothesis, some observation ending in state Σ' satisfies the formula φ_0; $\overline{[\theta_1]}\varphi_1$; ... $\overline{[\theta_u]}\varphi_u$; $\overline{[\theta_{u+1}]}$. In line (8) of the algorithm, *p-path*$[u]$ is defined to be true if $path(\Sigma')[u]$ is *true* for a state Σ' of level ℓ leading to Σ. Thus, for each u such that *p-path* $[u] = true$, it is the case that some observation Ω^ℓ ending in a global state of level ℓ leading to Σ satisfies the formula φ_0; $\overline{[\theta_1]}\varphi_1$; ... $\overline{[\theta_u]}\varphi_u$; $\overline{[\theta_{u+1}]}$. For this observation Ω^ℓ, let Ω be its extension obtained by adding global state Σ. For $u = 0$, such Ω will satisfy the formula φ_0;$\overline{[\theta_1]}$ (thus $path(\Sigma)[0] = true$) if θ_1 continues not to hold in Σ (line (9)).For $u \geq 1$, observation Ω satisfies the formula φ_0; $\overline{[\theta_1]}\varphi_1$; ... $\overline{[\theta_u]}\varphi_u$; $\overline{[\theta_{u+1}]}$, and consequently $path(\Sigma)[u]$ is true, if either some predecessor of Σ satisfies this prefix and Σ is not an undesired state (case *p-path*$[u] \wedge \neg\, \theta_{u+1}(\Sigma)$ of line 11), or some predecessor of Σ satisfies the prefix $\overline{[\theta_1]}\varphi_1$; ... $\overline{[\theta_{u-1}]}\varphi_{u-1}$; $\overline{[\theta_u]}$ and Σ is the next desired state (case *p-path*$[u - 1] \wedge \varphi_u(\Sigma)$ of line 11). ■

PROPERTY 4. *Given a formula* $\Phi = \overline{[\theta_1]}\varphi_1; \overline{[\theta_2]}\varphi_2; ...; \overline{[\theta_m]}\varphi_m; \overline{[\theta_{m+1}]}$ *belonging to the class ICS, algorithm* Detect_ICS(Φ) *returns true for computation* γ *if and only if* $\gamma \vDash$ **Pos** Φ.

Proof. As before, the algorithm proceeds by trying to extend the prefix of the formula that is satisfied by increasing levels of the lattice. The algorithm uses three variables *previous*, *current*, and *verified*, each a set of global states, with the same meaning as in algorithm *Detect_SS* and a boolean variable *detected* whose value reflects if some observation ending at a state of the current level satisfies Φ. At each step, the set *current* is defined as the states directly reachable from those in set *previous*. In other words, *previous* and *current* define sets of adjacent states in the lattice. The set *previous* is initialized to contain the single (fictitious) state Σ^{-1}. For each state Σ in *current*, the algorithm next computes the Boolean array $path(\Sigma)$. By Lemma 3, we know for each u whether some observation starting in state Σ^{-1} and ending in state Σ satisfies the formula φ_0; $\overline{[\theta_1]}\varphi_1$; ... $\overline{[\theta_u]}\varphi_u$; $\overline{[\theta_{u+1}]}$ where φ_0 is the fictitious predicate that holds only in state Σ^{-1}. If the formula Φ is satisfied up to Σ (i.e., $path(\Sigma)[m]$ is true), then Σ is added to the set *verified* (line (12)). Since formulas in *ICS* are not monotonic with respect to length of observations, we cannot claim detection of Φ if there are more levels; so the algorithm only records in variable *detected* whether some state of the current level satisfies Φ (line (13)). Lines (14) and (15) advance the algorithm to the next level of the lattice. If the current level contains no states, then the entire lattice has been visited and the value of *detected* corresponding to the final level is returned as the result of the algorithm (line (17)). Given the semantics of *detected*, the returned value is true if and only if some observation covering the entire computation satisfies Φ. ■

If the formula has the particular following form $\Phi = \overline{[\theta_1]}\varphi_1; \overline{[\theta_2]}\varphi_2; ... ;\overline{[\theta_m]}\varphi_m$ (i.e., the final $\overline{[\theta_{m+1}]}$ is missing), then the detection algorithm can be simplified. Without the final invalidating predicate $\overline{[\theta_{m+1}]}$, the entire formula once again becomes monotonic with respect to the length of observations. Consequently, the detection can be terminated as soon as some state Σ of the current level is such that $path(\Sigma)[m]$ is true. The algorithm can be modified by returning true immediately rather than setting the variable *detected* in line (13).

Verifying that a computation satisfies **Def** Φ rather than **Pos** Φ can be based on an algorithm similar to *Detect_ICS*(Φ). Such an algorithm, however, will require more history information to be associated with each global state of the lattice.

4.3. Cost of Property Detection Algorithms

Our approach to dynamic property detection consists of two concurrent activities: that of constructing the lattice of consistent global states by monitoring the computation and that of evaluating formulas by traversing this lattice. Similarly, the cost analysis of property detection can be decomposed into the costs associated with these two activities. The algorithms *Detect_SS*( ) and *Detect_ICS*( ) that were presented are concerned with the traversal of the lattice and not its construction, which can be accomplished using any one of the approaches described in [5, 1, 6]. For concreteness, we quantify these costs by following the approach described in [5, 1].

The lattice of consistent global states can be constructed on the fly by augmenting the underlying computation so as to notify a (logical) monitor whenever a process executes a relevant event. These notification messages contain the process local state along with a vector clock timestamp. While the amount of information necessary to encode the local state is application specific, the vector clock timestamp has fixed cost proportional to n, the number of processes. Thus, the total number of notification messages generated is intrinsic to the computation and is bounded by $O(nk)$, where k is the maximum number of relevant events executed by any process. The monitor stores the local states being notified by each process in a queue and performs a simple computation based on their vector timestamps in order to verify if they can belong to any consistent global states. This verification of consistency can be achieved in time $O(n^2)$. In the worst case, the monitor cannot throw away any local state and thus has to store up to $O(nk)$ local states. The number of potential global states that need to be tested for consistency can be as large as $O(k^n)$. Both the storage and computational worst case costs arise when the computation consists of n independent processes that do not communicate. We note that any realistic computation for which one would be interested in detecting dynamic properties is likely to be much more communication intensive such that the costs for building the lattice will be much smaller than the worst-case bounds indicated above.

As for the cost of traversing the lattice in order to establish the truth of a predicate, the major costs are functions

of the size of the lattice and $m + 1$, the number of terms in the predicate. For computations without any communication, every possible global state is consistent and the lattice has dimension $O(k^n)$. In the worst case, the predicate is such that every state in the lattice needs to be examined. The cost of examining a given state is proportional to the product of m and the number of other states at the preceding level of the lattice, which is bounded by $O(k^{n-1})$. Note that the traversal algorithm does not need access to the entire lattice but only to two adjacent levels at a time. Thus, the storage cost for the traversal phase is bounded by $O(k^{n-1})$, which is the maximum number of states that can exist at any single level. For each of these states, we need to associate the vector *path* of $m + 1$ elements.

The above cost analysis establishes upper bounds for dynamic property detection that result from unrealistic computations. As noted, realistic computations will have associated lattice structures that are much smaller, and thus, smaller detection costs as well. Section 6 indicates how the cost of lattice traversal may be further reduced through heuristics that are dependent on the property being detected [8].

5. RELATION TO TEMPORAL LOGICS

The work reported in this paper has had distributed debugging as its practical motivation [11]. Starting with the work of Cooper and Marzullo [5], we set out to explore ways in which their approach could be extended to capture larger classes of properties. The desire to include dynamic aspects of distributed computations has led us to consider sequences of states. Thus, it comes as no surprise that our proposal has strong parallels with other formalisms for reasoning about sequences. In particular, temporal logics which have emerged as particularly appropriate formal techniques for specifying and analyzing concurrent systems [4], parallel programs [17], protocols [10], and distributed systems [19, 20] are closely related to our approach.

Temporal logics extend ordinary predicate logic with modal operators. Unlike predicate logic formulas, which have single states as models, temporal logic formulas have sets of possibly infinite sequences of states as their models. In terms of expressive power, our proposal adds nothing to a traditional temporal logic having the operators *NEXT* and *UNTIL*. Every formula in *ICS* can be written as an equivalent, albeit very cumbersome, formula in this temporal logic.[6] In this sense, *ICS* could be viewed as a stylized syntactic shorthand for temporal logic formulas appropriate for distributed computations. The two idioms of *ICS*—causality-preserving sequencing and interval negation—are designed so as to admit concise and intuitive formulas for specifying properties of distributed computations where these notions play a key role [20].

One distinction between our approach and that traditionally pursued by logicians who study temporal logic is how we determine the models in which a formula is to be evaluated. In our case, the model needs to be inferred from circumstantial evidence accumulated by monitoring the underlying computation. Neither the computation nor the program responsible for producing it is known. In contrast, traditional work in temporal logic assumes that a set of models is given or that a formal description (e.g., a state transition graph) is given to produce the set of models.

Interpreting our formulas over observations corresponds exactly to model checking in a *linear time* temporal logic. Interpreting our formulas over computations, on the other hand, corresponds [18] to model checking in *branching time* temporal logic [7]. There are strong parallels between our proposal and a particular branching time temporal logic known as *computation tree logic* (CTL) [4]. Our computations are the analogs of the state transition graphs of CTL and the lattice structure representing consistent global states is the analog of the tree structure in CTL representing possible program states. The CTL operators *inevitable* and *potential* [4, 10] are the analogs of the modal operators **Def** and **Pos**, respectively, of Cooper and Marzullo [5].

If we compare the motivations for traditional model checking with those of dynamic property detection, we can note several important differences. First, model checking is concerned with verifying if a finite concurrent system meets its specification while our approach is concerned with detecting if an actual execution of the system exhibits a given behavior. As such, model checking has to derive all possible executions of the system from a finite representation before evaluating formulas. Our algorithms, on the other hand, monitor a single, but real, execution of the system so as to derive all possible observations and evaluate formulas on the fly. The need for representing all possible executions limits model checking to finite-state concurrent systems. In contrast, our algorithms can cope with infinite-state systems as long as any single execution is finite. In fact, if the property being detected is monotonic, even the execution may be infinite. In summary, model checking and dynamic property detection must be seen not as competitive but as complementary techniques in reasoning about concurrent systems.

6. DISCUSSION

Any proposal for increased expressive power has to be evaluated with respect to the increased cost of interpreting it. By allowing arbitrary Boolean expressions over global states as the building blocks of our formulas, our detection algorithms are forced to consider the lattice of consistent global states of computations. We conjecture that many

[6] For example, the formula $\overline{[\theta_1]}\varphi_1$; $\overline{[\theta_2]}\varphi_2$; ... is equivalent to $(\neg\theta_1)$ $UNTIL(\varphi_1 \wedge NEXT((\neg\theta_2)\ UNTIL(\varphi_2 \wedge NEXT(...))))$.

interesting system properties have an inherent total cost that is independent of the formalism used to detect them. For example, if we adopt a formalism where global predicates are restricted to be conjunctions or disjunctions of local predicates, detection of each term becomes inexpensive but the formula necessary to express the property can have an exponential number of terms so that the overall cost of detection remains high.

The cost of our detection algorithms may be reduced in several ways. One possibility is to restrict the class *SP*, from which classes *SS* and *ICS* are built, so as to admit efficient detection as is done in [23]. In this work, Tomlinson and Garg devise efficient mechanisms for detecting global predicates of the form $x + y < c$ where c is a constant and x and y are local variables at two processes. Another possibility for reducing the cost of detecting general global predicates is to limit the size of the lattice being examined. It is well known that certain communication patterns (e.g., those resulting from remote procedure calls) in a computation will lead to very "lean" lattice structures avoiding exponential number of states. Or, we can avoid examining the entire lattice by employing certain heuristics in the detection algorithms for constructing the set *current* from the set *previous*. These heuristics could be based on the notion of "greedy" linear extensions associated with partial orders [22]. With incomplete explorations of the lattice structure, detection of $\gamma \vDash \mathbf{Def}\ \Phi$ and $\neg(\gamma \vDash \mathbf{Pos}\ \Phi)$ will take on probabilistic interpretations.

7. CONCLUSIONS

Two new classes of global predicates have been proposed to express dynamic properties in distributed computations. These classes admit Boolean expressions over global states as building blocks and include temporal specifications through causality-preserving sequencing and interval negation. The possibility to specify undesired states between desired ones adds significantly to the expressiveness of the proposal. Our formalism borrows ideas from temporal logics [7, 4, 20]. The originality of our work lies, on the one hand, in the syntactic simplicity of formulas expressing even complex properties, and on the other hand, in the design of the associated algorithms for on-line verification of such formulas.

Our detection algorithms operate on-line with respect to the structure responsible for the model (the underlying computation). In that sense, they could be viewed as "on-the-fly" model checkers. Unlike traditional temporal logic applications where model checking is performed off-line, our approach constructs the relevant portions of the model (lattice of global states) as the underlying computation evolves. At no time is the model constructed or stored in its entirety. Only a current window of adjacents states is considered with dynamic programming principles being exploited to keep track of past states in a storage-efficient manner.

APPENDIX: A FURTHER EXTENSION

In this Appendix we consider an extension to the class *ICS* whereby not only undesired properties, but also desired ones can be specified to hold in sequences of adjacent global states. In other words, this extension will allow us to write specifications consisting of alternating *positive intervals* (where desired predicates continuously hold) and *negative intervals* (where undesired predicates do not hold). While the negative intervals may be empty, positive intervals must contain at least one global state for the specifications to be meaningful.

In defining the syntax of this new class, called *AIS* for *alternating interval sequence*, we borrow the notation φ^+ from regular expressions to denote "one or more instances of φ." Thus, formulas in *AIS* follow the same syntax rules as those of *ICS* with the exception that predicates designating desired properties may contain a plus.

The semantics of *AIS* with respect to observations is a generalization of that for *ICS*. A formula in *ICS* is satisfied by an observation if the negative intervals are separated by positive intervals of length exactly one. In the case of *AIS*, negative intervals may be delimited by positive intervals of arbitrary non-zero length.

DEFINITION 9. Let $\Phi = \overline{[\theta_1]}\,\varphi_1^+; \ldots; \overline{[\theta_m]}\,\varphi_m^+; \overline{[\theta_{m+1}]}$ be a formula in AIS. If any of the θ_i in Φ are missing, we assume they are defined as the constant *false*. Observation Ω satisfies Φ, denoted $\Omega \vDash \Phi$, if and only if

1. If $m = 0$, then $\forall \Sigma \in \Omega$: $\neg\, \theta_1(\Sigma)$,
2. If $m \neq 0$, then there exist m global states $\Sigma^{i_1} \ldots \Sigma^{i_m} \in \Omega$ and m nonnegative integers $a_1, \ldots, a_m$ such that
   (a) $\Sigma^{i_1} \rightsquigarrow_\Omega \Sigma^{i_2} \rightsquigarrow_\Omega \cdots \rightsquigarrow_\Omega \Sigma^{i_m}$,
   (b) $k = 1, \ldots, m : \forall i : i_k \leq i \leq i_k + a_k : \varphi_k(\Sigma^i)$,
   (c) $\forall i : i < i_1 : \neg\, \theta_1(\Sigma^i)$,
   (d) $k = 2, \ldots, m : \forall i : i_{k-1} + a_{k-1} < i < i_k : \neg\, \theta_k(\Sigma^i)$, and
   (e) $\forall i : i > i_m + a_m : \neg\, \theta_{m+1}(\Sigma^i)$.

Note that having some $a_k = 0$ results in a positive interval of length one containing the single state Σ^{i_k} in which φ_k holds. Thus, the class *ICS* is a special case of *AIS*, obtained by requiring all of the a_k to be zero. The semantics of *AIS* with respect to computations is defined in terms of the usual modal operators.

Figure 4 illustrates an algorithm for verifying if a computation satisfies **Pos** Φ for a formula in the class *AIS*. The algorithm is based on that for detecting formulas in *ICS*. The only notable addition is that of boolean array *ni* of size m used for denoting noninterrupted sequences of desirable states. Variable $ni(\Sigma)[u]$ is true if some observation ending in global state Σ can be decomposed into a prefix $\Sigma^{-1} \ldots \Sigma^i$ satisfying the formula $\overline{[\theta_1]}\,\varphi_1^+; \ldots \overline{[\theta_{u-1}]}\,\varphi_{u-1}^+; \overline{[\theta_u]}$ and the nonempty suffix $\Sigma^{i+1} \ldots \Sigma$ consisting of only states where φ_u holds. Note that now variable $path(\Sigma)[u]$ is true if and only if there exists an observation terminating at Σ and satisfying the prefix formula $\overline{[\theta_1]}\,\varphi_1^+; \ldots \overline{[\theta_u]}\,\varphi_u^+; \overline{[\theta_{u+1}]}$.

```
    function Detect_AIS(Φ);
    var previous, current, verified: set of global states;
        path, p_path: array[0..m] of boolean;
        ni: array[1..m] of boolean;
        detected: boolean;
        u: integer;
    begin
1       previous := {Σ^-1};
2       path(Σ^-1)[0] := true;
3       for u := 1 to m do path(Σ^-1)[u] := false; ni(Σ^-1)[u] := false od
4       verified := {};
5       current :={Σ^0};
6       while (current ≠ {}) do
7           foreach Σ ∈ current do
8               for u := 0 to m do p_path[u] := ⋁_{Σ'∈pred(Σ)} path(Σ')[u] od
9               for u := 1 to m do ni[u] := ⋁_{Σ'∈pred(Σ)} ni(Σ')[u] od
10              path(Σ)[0] := p_path[0] ∧ ¬θ_1(Σ);
11              for u := 1 to m do
12                  path(Σ)[u] := (p_path[u] ∧ ¬θ_{u+1}(Σ)) ∨ (p_path[u − 1] ∧ φ_u(Σ))
                                  ∨(ni[u] ∧ φ_u(Σ));
13                  ni(Σ)[u] := (ni[u] ∧ φ_u(Σ)) ∨ (p_path[u − 1] ∧ φ_u(Σ))
                od
14              if path(Σ)[m] then verified := verified ∪ {Σ} fi
            od
15          if verified ≠ {} then detected := true else detected := false fi
16          previous := current;
17          current :={global states directly reachable from those in previous};
18          verified := {};
        od
19      return (detected)
    end
```

FIG. 4. Algorithm for detecting global predicates in the class *AIS*.

ACKNOWLEDGMENTS

We are grateful to F. Schneider for his help in relating our work to temporal logics. The presentation has benefited from valuable comments by V. Garg, T. Jeron, K. Marzullo, M. Melliar-Smith, and the anonymous referees. We thank M. Hurfin and M. Mizuno for pointing out an error in a previous version of the paper. This work has been supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Project 6360 (BROADCAST). Babaoğlu was further supported by the Italian National Research Council and the Ministry of University, Research and Technology. Raynal was further supported by the French CNRS under the grant Parallel Traces.

REFERENCES

1. Ö. Babaoğlu and K. Marzullo, Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. J. Mullender (ed.). *Distributed Systems,* Chap. 4, pp. 55–96. ACM Press, New York, 1993.
2. P. C. Bates and J. C. Wileden, High-level debugging of distributed systems: the behavioral abstraction approach. *J. Systems and Software* **4**(3), 255–264: (Dec. 1983).
3. K. M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Systems* **3**(1), 63–75, (Feb. 1985).
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems* **8**(2), 244–263, (1986).
5. R. Cooper and K. Marzullo, Consistent detection of global predicates. *ACM/ONR Workshop on Parallel and Distributed Debugging,* Santa Cruz, California, May 1991, pp. 163–173.
6. C. Diehl, C. Jard, and J. X. Rampon, Reachability analysis on distributed executions. In M.-C. Gaudel and J.-P. Jouannaud (Ed.). *Proceedings of TAPSOFT,* Orsay, Paris, France, Apr. 1993. Lecture Notes in Computer Science, 161.668, pp. 629–643. Springer-Verlag, Berlin/New York, 1993.
7. E. A. Emerson and J. Srinivasan, Branching time temporal logic. In G. Rozenberg J. W. de Bakker, and W.-P. de Roever (Eds.). *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency,* Noordwijkerhout, The Netherlands, June 1988. Lecture Notes in Computer Science, Vol. 354, pp. 123–172. Springer-Verlag, Berlin/New York, 1988.
8. E. Fromentin and M. Raynal, Inevitable global states: A concept to detect unstable properties of distributed computations in an observer independent way. *Proceedings of 6th IEEE Symposium on Parallel and Distributed Processing,* Dallas, Texas, Oct. 1994, pp. 242–248.
9. V. K. Garg and B. Waldecker, Detection of unstable predicates in distributed programs. *Proceedings of the 12th International Conference on Foundations of Software Technology and Theoretical Com-*

puter Science, New Delhi, India, Dec. 1992. Lecture Notes in Computer Science, Vol. 652, pp. 253–264. Springer-Verlag, Berlin/New York, 1993.

10. R. Gotzhein, Temporal logic and applications: A tutorial. *Computer Networks and ISDN Systems,* pp. 203–218, 1992.
11. M. Hurfin, N. Plouzeau, and M. Raynal, Debugging tool for distributed Estelle programs. *J. Comput. Comm.,* 328–333 (May 1993).
12. M. Hurfin, N. Plouzeau, and M. Raynal, Detecting atomic sequence of predicates in distributed computations. *Proceedings of the ACM/ ONR Workshop on Parallel and Distributed Debugging,* San Diego, California, May 1993, pp. 32–41. Reprinted in *ACM SIGPLAN Notices,* Dec. 1993.
13. L. Lamport, Time, clocks, and the ordering of events in a distributed system. *Comm. ACM* **21**(7), 558–565, (July 1978).
14. K. Marzullo and G. Neiger, Detection of global state predicates. In S. Toueg and P. Spirakis (Eds.) *Proceedings of the Fifth International Workshop on Distributed Algorithms (WDAG-91),* Delphi, Greece, Oct. 1991. Lecture Notes in Computer Science, Vol. 579, pp. 254–272, Springer-Verlag, Berlin/New York, 1992.
15. F. Mattern, Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.* **18**(4), 423–433 (Aug. 1993).
16. B. P. Miller and J. Choi, Breakpoints and halting in distributed programs. *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems,* San Jose, California, July 1988, pp. 316–323.
17. A. Pnueli, Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, (Eds.). *Current Trends in Concurrency.* Lecture Notes in Computer Science, Vol. 224, pp. 510–584. Springer-Verlag, Berlin/New York, 1986.
18. F. B. Schneider, Personal communication, May 1993.
19. R. L. Schwartz and P. M. Melliar-Smith, Temporal logic specifications for distributed systems. *Proceedings of the 2nd IEEE International Conference on Distributed Computing Systems,* 1981, pp. 446–454.

Received December 6, 1993; revised November 11, 1994; accepted November 12, 1994

20. R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt, An interval logic for high-level temporal reasoning. *Proceedings of the ACM Symposium on Principles of Distributed Computing,* Montreal, Canada, Aug. 1983, pp. 173–186.
21. R. Schwarz and F. Mattern, Detecting causal relationships in distributed computations: In search of the Holy Grail. *Distrib. Comput.* **7**(3), 149–174, (1994).
22. M. M. Syslo, Minimizing the jump number for partially-ordered sets: A graph-theoretic approach, II. *Discrete Math.* **63,** 279–295 (1987).
23. A. I. Tomlinson and V. K. Garg, Detecting relational global predicates in distributed systems. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging,* San Diego, California, May 1993, pp. 21–31. Reprinted in *ACM SIGPLAN Notices,* Dec. 1993.

ÖZALP BABAOĞLU is professor of computer science at the University of Bologna, Italy. He received a Ph.D. in 1981 from the University of California at Berkeley, where he was one of the principal designers of BSD Unix. Before moving to Bologna in 1988, Dr. Babaoğlu was an associate professor in the Department of Computer Science at Cornell University. He is active in several European research projects exploring issues related to fault tolerance and large scale in distributed systems. Dr. Babaoğlu serves on the editorial boards for *ACM Transactions on Computer Systems* and Springer-Verlag *Distributed Computing.*

MICHEL RAYNAL is professor of computer science at the University of Rennes, France. He received a Doctorat d'état en Informatique in 1981 from Rennes University. His research interests are distributed algorithms, operating systems, protocols, and parallelism. He wrote several books devoted to distributed systems, among them: *Distributed Computations and Networks* (1988), published by the MIT Press, and *Synchronization and Control of Distributed Programs* (1990), published by Wiley. He served as a PC member for many international conferences. He is currently involved in the Esprit Project BROADCAST devoted to the study of large-scale fault-tolerant distributed systems.