Sequence-based global predicates for distributed computations : definitions and detection algorithms

Ozalp Babaoglu, Michel Raynal

▶ To cite this version:

Ozalp Babaoglu, Michel Raynal. Sequence-based global predicates for distributed computations : definitions and detection algorithms. [Research Report] RR-1961, INRIA. 1993. inria-00074712

HAL Id: inria-00074712
https://inria.hal.science/inria-00074712

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Sequence-Based Global Predicates for Distributed Computations: Definitions and Detection Algorithms

Ozalp BABAOGLU
Michel RAYNAL

N° 1961
Juillet 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

Rapport de recherche

1993

Sequence-Based Global Predicates for Distributed Computations: Definitions and Detection Algorithms

Ozalp Babaoglu*, Michel Raynal**

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet ADP

Rapport de recherche n° 1961 — Mai 1993 — 15 pages

Abstract: We consider the problem of detecting sequences of predicates defined over global states of distributed computation. We introduce two new global predicate classes called *simple sequences* and *interval-constrained sequences* that define causally-ordered sets of desirable states along with intervening forbidden states. Our formalism is more general than former 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 programs testing, debugging and dynamic reconfiguration in distributed systems.

(Résumé : tsvp)

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

*Department of mathematics, University of Bologna, Piazza Porta S. Donato 5, 40127 Bologna (Italy) ozalp@dm.unibo.it
**Michel.Raynal@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Définition et algorithmes de détection de séquences de prédicats globaux pour les calculs répartis

Résumé : Cet article introduit deux nouvelles classes de prédicats globaux portant sur les exécutions réparties : les séquences simples et les séquences avec contraintes d'intervalles. Ces types de prédicats sont formellement définis et des algorithmes qui les détectent sont présentés. L'utilité de telles séquences de prédicats est illustrée par des exemples empruntés au test de programmes, à la mise au point et la reconfiguration de charge dans les systèmes répartis.

Sequence-Based Global Predicates for Distributed Computations: Definitions and Detection Algorithms*

Özalp Babaoğlu[†] Michel Raynal[‡]

Abstract

We consider the problem of detecting sequences of predicates defined over global states of a distributed computation. We introduce two new global predicate classes called *simple sequences* and *interval-constrained sequences* that define causally-ordered sets of desirable states along with intervening forbidden states. Our formalism is more general than former 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.

Keywords: Non-stable predicates, behavioral pattern specification, distributed debugging, program verification.

1 Introduction

The ability to detect when the state of a distributed computation satisfies a certain property is a fundamental problem in distributed algorithm design. A large class of problems including distributed debugging, 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 system properties of interest.

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 instantaneous access to it. Thus, 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.

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

[†]Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, 40127 Bologna (Italy), Tel: +39 51 354430, FAX: +39 51 354490, E-mail: ozalp@dm.unibo.it. Supported in part by Hewlett-Packard of Italy and the Italian Ministry of University, Research and Technology.

[‡]IRISA, Campus de Beaulieu, 35042 Rennes Cedex (France), E-mail: raynal@irisa.fr. Supported in part by the CNRS under the grant Parallel Traces.

The problem of detecting simple predicates 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 [4]. 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 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 repeated [3, 17,14]. No matter how frequently taken, a sequence of snapshots may have gaps that correspond to exactly those global states in which the (non-stable) predicate holds. Any reasonable definition of "detection" for the case of non-stable predicates must be based on *observations* [5,13,8,1]. 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 non-stable predicates independent of any particular observation [5,13,8].

While simple predicates (stable or not) on a single global state are able to capture many interesting system properties, they inherently lack notions of time or relative order. In a sense, detecting simple predicates is akin to verifying *safety* properties (e.g., deadlock, termination, overload, program error, etc.) of a distributed computation. In order to characterize dynamic *behavioral patterns* of distributed computations, simple predicate specifications must be extended to include a temporal component [2,15,16]. By detecting a set of desired (or undesired) system states in a particular order, we are able to verify certain *liveness* properties of distributed computations.

The numerous proposals for extending simple predicates so as to include behavioral pattern specifications can be divided into those based on states [15,8,11] and those based on events [18, 9,7,10]. In all of these proposals, the ordering specification is based on the notion of causal precedence [12] and global properties are specified as compositions of local properties — those that can be expressed naming variables of a single process only. In the most general case, global predicates are constrained to be conjunctions [8] or sequences of local predicates [15,11].

In this paper we consider behavioral pattern specifications that admit arbitrary predicates over global states as building blocks. We define the syntactic classes *SS* (simple sequence) and *ICS* (interval-constrained sequence) of global predicates, give their semantics, and develop algorithms to detect formulas belonging to them. Our formalism includes existential and universal quantification such that detection of behavioral patterns expressed in our language can be made independent of observations. The class *ICS* is more expressive than earlier proposals and allows specification of many interesting system properties concisely and naturally. Our results can be viewed as unifying and generalizing most existing proposals for global predicate detection [15,5,13,8,11].

In the next Section, we describe a formal model for distributed systems, and define distributed computations, global states, runs and observations within this model. In Section 3 we introduce our notation for describing predicates suitable for behavioral pattern specifications and give their semantics. Section 4 develops on-line algorithms for detecting these predicates. Section 5 discusses the cost of our algorithms and identifies possible techniques for reducing it. Section 6 concludes the paper.

2 System Model

2.1 Asynchronous Distributed Systems

We adopt the terminology and notation of [1]. A distributed system is a collection of sequential *processes* $p_1, p_2, \ldots, p_n$ that communicate by exchanging messages. 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.

The *local history* of process p_i during the computation is a (possibly infinite) sequence of events $h_i = e_i^1 e_i^2 \ldots$. 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 $h_i^k = e_i^1 e_i^2 \ldots e_i^k$ 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 a set $H = h_1 \cup \cdots \cup h_n$ containing all of its events.

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 [12]. It is common to depict distributed computations using an equivalent graphical representation called a *space-time diagram* as shown in Figure 1.

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 $\Sigma = (\sigma_1^{c_1}, \ldots, \sigma_n^{c_n})$, 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. Informally, consistent global states correspond to those that could have been 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*. Obviously only an external observer can construct the run for the computation. Any total order consistent with causal precedence that is constructed within the system (thus subject to all the uncertainties) is called a *sequential observation* (observation

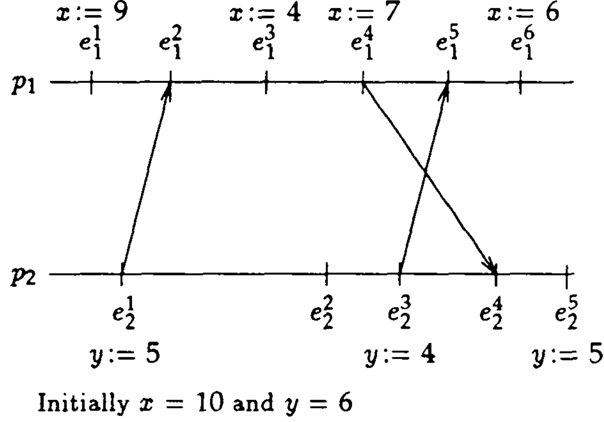

Figure 1: Space-Time Diagram Representation of a Distributed Computation

for short) of γ. In other words, observations are linearizations of the partial order defined by causal precedence.

Observations can be expressed as sequences of consistent global states rather than sequences of events. In particular, an observation $\Omega = e^1 e^2 \ldots$ of computation γ is equivalent to the sequence of global states $\Sigma^0\Sigma^1\Sigma^2\ldots$ where Σ^0 denotes the initial global state $(\sigma_1^0, \ldots, \sigma_n^0)$. Each global state Σ^i of the observation is obtained from the previous state Σ^{i-1} by some process executing the single event e^i in γ. For two such global states of observation Ω, we say that Σ^{i-1} *leads to* Σ^i *in* Ω. Let $\leadsto_\Omega$ 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 $\Sigma \leadsto_\Omega \Sigma'$. We drop the subscript 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 defines a *lattice*. The lattice consists of n distinct axes, with one axis for each process. Let $\Sigma^{k_1 \ldots k_n}$ 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*. Figure 2 illustrates the lattice of global states associated with the distributed computation of Figure 1. 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 from a single observation of the computation can be found in [5,13,1,6].

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

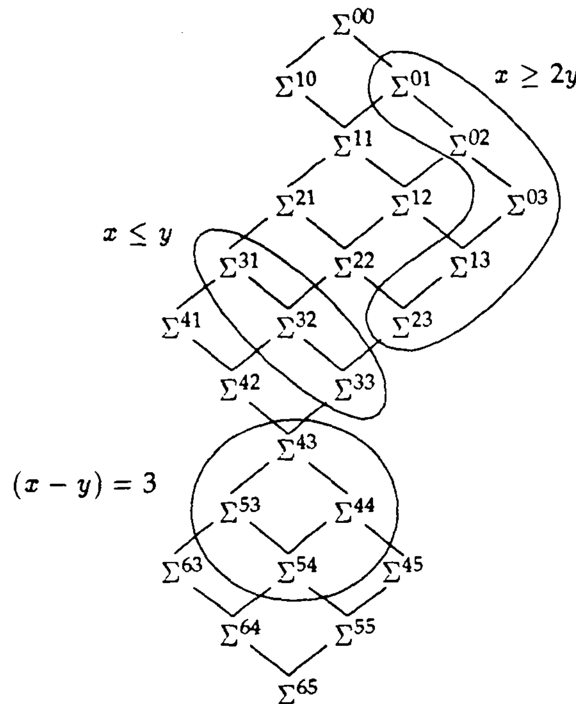

Figure 2: Lattice of Consistent Global States

predicate classes can be viewed as small languages for specifying system properties and detecting a global predicate can be viewed as interpreting a formula in a model, namely a distributed computation.

3.1 Simple Predicates

Syntactically, the simplest global predicate class we consider is called *SP* and corresponds to general boolean expressions defined over single global states. Formulas of *SP* may reference any global state variable, the constants *true* and *false*, and follow the usual syntax rules for boolean expressions without any restrictions. We let $\varphi(\Sigma)$ denote the value of predicate φ when evaluated in global state Σ. Figure 2 illustrates three simple predicates defined for the computation of Figure 1 and identifies the global states $\Sigma = (x, y)$ where each predicate holds.

Note that the global predicates of [8,11] 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 our class *SP*.

1. Load in the network is balanced.

2. Queueing delay along the route from A to B is less than 5msec.

3. No more than 100 total users logged in to machines A, B and C.

4. In computation of Figure 1, $x \geq 2y$.

Furthermore, expressing certain properties in SP' can be extremely cumbersome and costly. For example, consider the property "no more than m processes holding the resource" in a system with n processes. The global predicate in SP' will have to enumerate all possible configurations in which m or fewer processes out of n are holding the resource.

With respect to observations, the semantics of the class *SP* is defined as follows:

Definition 1 *Let φ be a formula in SP. Observation Ω* satisfies *φ, denoted $\Omega \models \varphi$, if and only if there exists global state $\Sigma \in \Omega$ such that $\varphi(\Sigma)$.*

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

Definition 2 *Predicate φ is* stable *if and only if*

$$\forall \Sigma, \Sigma' : \ \varphi(\Sigma) \wedge (\Sigma \leadsto \Sigma') \Rightarrow \varphi(\Sigma')$$

In other words, if a stable predicate is true in some global state, then it must be true in all global states reachable from it.[1]

Detection of non-stable 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 $x \geq 2y$ of Figure 2 illustrates the problem. 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 $\gamma \models$* **Pos** *φ, if and only if there exists an observation Ω of γ such that $\Omega \models \varphi$.*

2. *Distributed computation γ satisfies* **Def** *φ, denoted $\gamma \models$* **Def** *φ, if and only if for all observations Ω of γ it is the case that $\Omega \models \varphi$.*

Note that the above definitions of **Pos** and **Def** have been called *weak* and *strong* formulas, respectively, by Garg and Waldecker [8]. It is easy to show that if φ is a stable predicate, then **Pos** $\varphi \equiv$ **Def** φ.

[1] 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.

3.2 Simple Sequences

As noted earlier, simple global predicates can effectively capture safety properties of systems but are unable to capture behavioral patterns. In order to include behavioral pattern specifications for distributed computations, global predicates must include a temporal component. The first such proposal is due to Miller and Choi [15] where they 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

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

where *SP* is as defined in the previous Section. In other words, formulas in *SS* consist of sequences of simple predicates, each defined over a single global state. Referring to Figure 2, 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 $\Omega \models \Phi$, if and only if there exist m global states $\Sigma^{i_1} \ldots \Sigma^{i_m} \in \Omega$ such that*

1. $\Sigma^{i_1} \leadsto \Sigma^{i_2} \leadsto \cdots \leadsto \Sigma^{i_m}$ *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 the observation, two conditions must hold. First, each of the component predicates must hold in distinct global states. Second, each 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

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

is not equivalent to that of

$$\Omega \models \varphi_1; \varphi_2; \ldots; \varphi_m$$

since the former is unable to capture the relative ordering property.[2]

The semantics of *SS* with respect to computations remains exactly as given in Definition 3 with the introduction of modal operators **Pos** and **Def** and the above semantics with respect to observations. For the computation of Figure 1, the following formulas hold:

Pos $(x \geq 2y); (x \leq y); (x - y = 3)$
$\neg$ **Pos** $(x \leq y); (x \geq 2y); (x - y = 3)$
Def $(x \leq y); (x - y = 3)$

[2]The two formulas specify without order and with order, respectively, satisfactions of a set of simple predicates. They differ from the formula $\Omega \models \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m$ that expresses the requirement where all predicates φ_i must be simultaneously true.

3.3 Interval-Constrained Sequences

There are cases when the behavioral pattern specification must describe not only sequences of desired states, it must also describe undesirable intervening states [11]. 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 being held by a particular process, the total number of outstanding requests does not exceed some threshold. Note that the intervals of interest with respect to these "forbidden" 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.

We formalize and extend these ideas by defining a new global predicate class called *ICS*, for *interval-constrained sequence*, with the syntax rule

$$ICS ::= ICS; [SP]\, SP \mid [SP]\, SP$$

where *SP* is the simple predicate class as before. In other words, formulas in *ICS* are strict alternations of two sequences of simple predicates: those that describe desired conditions and those that describe undesired conditions, written inside square brackets. Again referring to Figure 2, global predicates $[x \geq 2y](x \leq y); [true](x - y = 3)$ and $[x \leq y](x - y = 3)$ are instances of *ICS*.

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

Definition 5 *Let* $\Phi = [\theta_1]\varphi_1; \ldots; [\theta_m]\varphi_m$ *be a formula in ICS. Observation* Ω *satisfies* Φ*, denoted* $\Omega \models \Phi$*, if and only if there exist m global states* $\Sigma^{i_1} \ldots \Sigma^{i_m} \in \Omega$ *such that*

1. $\Sigma^{i_1} \rightsquigarrow \Sigma^{i_2} \rightsquigarrow \cdots \rightsquigarrow \Sigma^{i_m}$,
2. $\varphi_1(\Sigma^{i_1}) \wedge \cdots \wedge \varphi_m(\Sigma^{i_m})$,
3. $\forall \Sigma : \Sigma \rightsquigarrow \Sigma^{i_1} : \neg\theta_1(\Sigma)$ *and*
4. $k = 2, \ldots, m : \forall \Sigma : \Sigma^{i_{k-1}} \rightsquigarrow \Sigma \rightsquigarrow \Sigma^{i_k} : \neg\theta_k(\Sigma)$.

In other words, the semantics of *ICS* is the same as that of *SS* as far as the desired sequence is concerned, however, with the additional requirement that between the pair of global states satisfying predicates φ_{i-1} and φ_i, there can be no intermediate states satisfying θ_i. In particular, the prefix of the observation up to the state satisfying φ_1 must not include any state satisfying θ_1. Clearly a formula of *ICS* where all the predicates θ describing undesired conditions are *false* has exactly the same meaning as the corresponding formula in *SS* keeping only predicates φ for the desired conditions.

The class *ICS* happens to be quite expressive and can be used to specify many behavioral patterns that are of interest in distributed systems. In particular, by defining appropriate forbidden states, we are able to express notions such as "atomic actions" where undesirable interleavings are excluded. In terms of observations, atomicity of actions can be expressed as producing desired states that are adjacent to each other. Here are certain *ICS* formulas with their intuitive meanings:

- $[false]\varphi_1;[true]\varphi_2 \equiv$ Predicates φ_1 and φ_2 satisfied in adjacent global states, in that order.
- $[false]\varphi_1;[true]\theta;[true]\varphi_2 \equiv$ Exactly one global state between those satisfying φ_1 and φ_2, and this state satisfies θ.
- $[true]\varphi \equiv$ Predicate φ satisfied in the initial global state.

As before, we define the semantics of *ICS* with respect to computations by introducing the modal operators **Pos** and **Def**. Then, the semantics of *ICS* remains exactly as given in Definition 3 using the above semantics with respect to observations. For the computation of Figure 1, the following formulas hold:

Pos $[x \geq 2y](x \leq y);[true](x-y=3)$
Pos $[false](x \geq 2y);[true](x \leq y);[true](x-y=3)$
$\neg$ **Pos** $[false](x \geq 2y);[x \leq y](x-y=3)$
Def $[false](x \leq y);[x \geq 2y](x-y=3)$
Def $[true](x=10 \wedge y=6);[false](x \leq y);[false](x-y=3)$

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 states of the lattice structure corresponding to the computation.

4.1 Detection of Simple Sequences

The algorithm shown in Figure 3 detects if **Def** Φ is satisfied during a computation where predicate $\Phi = \varphi_1;\varphi_2;\ldots;\varphi_m$ 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 [5,13,1].

For notational convenience, we assume that a fictitious global state, Σ^{-1}, precedes all other states of the computation. The function $pred(\Sigma)$ returns a set of global states containing all of 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. In other words, $prefix(\Sigma) = u$ means that **Def** $\varphi_1;\varphi_2;\ldots;\varphi_u$ is satisfied up to global state Σ. Note that formulas in *SS* are monotonic with respect to the length of observations. In other words, if a formula is satisfied by an observation of length ℓ, it can never be invalidated by extending the observation beyond ℓ.

```
function Detect_SS(Φ);
    var previous, current, verified: set of global states;
        prefix, u: integer;
    begin
        previous := {Σ^-1};                                                    % level := -1
        prefix(Σ^-1) := 0;
        repeat
            verified := {};                                                    % level := level+1
            current :={global states directly reachable from those in previous};
            foreach Σ ∈ current do
                u :=     min     (prefix(Σ'));                                  (*)
                     Σ'∈pred(Σ)
                if φ_{u+1}(Σ) then prefix(Σ) := u + 1
                else prefix(Σ) := u;
                fi
                if (prefix(Σ) = m) then verified := verified ∪ {Σ} fi
            od
            if ((verified ≠ {}) ∧ (verified = current)) then return  true fi    (**)
            previous := current − verified;
        until (previous = {});
        return  false;
    end
```

Figure 3: Algorithm for Detecting Global Predicates in the Class SS

The algorithm uses three variables *previous, current* and *verified,* all sets of global states. The test for satisfaction of the formula proceeds by levels of the lattice structure. We first compute *current* as the set of states directly reachable from those in *previous* (the set of states at the next level in the lattice). Note that *previous* is initialized to contain only the fictitious state Σ^{-1}. For each state Σ in *current,* we next compute $prefix(\Sigma)$. The prefix of the formula satisfied up to Σ is at least as long as the minimum among the prefixes satisfied by its predecessors. If a new term of the formula holds in state Σ, then the prefix up to Σ can be extended by one term. If the entire formula is satisfied up to Σ, then the state is added to the set *verified.* If the entire formula is satisfied by all states of the current level, then 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.

Theorem 1 *Given a global predicate* $\Phi = \varphi_1; \varphi_2; \ldots; \varphi_m$ *belonging to the class SS, function* $Detect_SS(\Phi)$ *returns true for computation* γ *if and only if* $\gamma \models$ **Def** Φ.

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

```
function Detect_ICS(Φ);
    var previous, current, verified: set of global states;
        path, pp: array[0..m] of boolean;
        u: integer;
    begin
        previous := {Σ^-1};                                                    % level := -1
        path(Σ^-1)[0] := true;
        for u := 1...m do path(Σ^-1)[u] := false od
        repeat
            verified := {};                                                    % level := level+1
            current :={global states directly reachable from those in previous};
            foreach Σ ∈ current do
                for u := 0...m do pp[u] :=      ⋀          path(Σ')[u] od            (*)
                                            Σ'∈pred(Σ)
                path(Σ)[0] := pp[0] ∧ ¬θ_1(Σ);
                for u := 1...m do path(Σ)[u] := (pp[u] ∧ ¬θ_{u+1}(Σ)) ∨ (pp[u − 1] ∧ φ_u(Σ)) od
                if path(Σ)[m] then verified := verified ∪ {Σ} fi
            od
            if ((verified ≠ {}) ∧ (verified = current)) then return true fi      (**)
            previous := current − verified;
        until (previous = {});
        return false;
    end
```

Figure 4: Algorithm for Detecting Global Predicates in the Class *ICS*

Theorem 2 *Given a global predicate* $\Phi = \varphi_1; \varphi_2; \ldots; \varphi_m$ *belonging to the class SS, function* $Detect_SS(\Phi)$, *with the following modifications, returns true for computation* γ *if and only if* $\gamma \models$ **Pos** Φ.

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

4.2 Detection of Interval-Constrained Sequences

The algorithm shown in Figure 4 detects **Def** Φ for predicates belonging to the class *ICS* and has a structure similar to that of algorithm $Detect_SS$. Rather than associating a single integer variable with each global state Σ, algorithm $Detect_ICS$ associates a boolean array $path(\Sigma)$ of dimension $m+1$. An element $path(\Sigma)[u]$ of this array is true if all observations starting with the initial state Σ^0 and ending at state Σ satisfy the prefix $[\theta_1]\varphi_1; \ldots; [\theta_u]\varphi_u$ of Φ without being invalidated by the next undesired predicate θ_{u+1}. We assume that the formula contains a fictitious final predicate

θ_{m+1} which is the constant false. The array associated with (fictitious) state Σ^{-1} is initialized to be true for a prefix of length zero and false for all others.

As before, the algorithm proceeds by trying to extend the prefix of the formula that is satisfied by increasing levels of the lattice. 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 θ_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 avoid future forbidden states (where θ_u holds).

The algorithm uses the same variables $previous$, $current$ and $verified$, all sets of global states, with the same meaning as before. For each global state Σ under consideration, the prefix of the formula that can be satisfied is extended to u terms if either all predecessors of Σ can satisfy the prefix of u terms and Σ is not a forbidden state, or all predecessors of Σ satisfy the prefix of $u-1$ terms and Σ is the next desired state.

Theorem 3 *Given a global predicate* $\Phi = [\theta_1]\varphi_1; [\theta_2]\varphi_2; \ldots; [\theta_m]\varphi_m$ *belonging to the class ICS, function* $Detect_ICS(\Phi)$ *returns true for computation* γ *if and only if* $\gamma \models$ **Def** Φ.

As with the previous algorithm, verifying that a computation satisfies **Pos** Φ rather than **Def** Φ requires two simple syntactic modifications to algorithm $Detect_ICS(\Phi)$.

Theorem 4 *Given a global predicate* $\Phi = [\theta_1]\varphi_1; [\theta_2]\varphi_2; \ldots; [\theta_m]\varphi_m$ *belonging to the class ICS, function* $Detect_ICS(\Phi)$*, with the following modifications, returns true for computation* γ *if and only if* $\gamma \models$ **Pos** Φ.

1. *The* $\wedge$ *(logical and) operator of line (*) is changed to* $\vee$ *(logical or),*
2. *The test in line (**) is changed to "***if*** $((verified \neq \{\})$ ***then return true fi*** *".*

5 Discussion

The class *ICS* can be extended or modified in several ways. Some of these could be characterized as "syntactic sugar" in that they do not increase the expressiveness of the class, but simplify notation. As we have seen in Section 3.3, *ICS* formulas can be constructed to express notions such as "no intermediate states" or "exactly one intermediate state." We could devise a shorthand notation for these and related notions based on regular expressions. Similar proposals such as *data path expressions* have been made for the case of behavior pattern specification through event sequences [9,10].

As for the class *SS*, an alternative "cumulative" semantics could be proposed. Intuitively, given a formula $\varphi_1; \varphi_2; \ldots; \varphi_m$, an observation would satisfy it if first φ_1 holds and continues to hold on all states until the state where φ_m holds; next φ_2 holds and continues to hold on all states until the state where φ_m holds; etc. Note that this is different from requiring that each of the φ_i be stable since once the final predicate φ_m holds, none of them are obliged to continue to hold. While interesting for certain properties, this proposal can also be viewed as syntactic sugar since we can express the same properties using formulas (albeit more complex) in the class *ICS*. While

our arguments have not been formal, we feel that the class *ICS* is in some sense "minimal" with respect to achieving a desired level of expressiveness.

Any proposal for increased expressiveness 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 feel that many properties have an inherent total cost which is independent of the formalism used to detect them. Consider the example cited earlier where we are interested in verifying that at most m processes out of n may be holding a resource. If we adopt a formalism where global predicates are restricted to be conjunctions or disjunctions of local predicates, detection becomes inexpensive but the formula necessary to express the property has an exponential number of terms. On the other hand, in a formalism such as ours, the property has an extremely compact expression but the detection phase is potentially expensive.

The cost of our detection algorithms may be reduced in two ways. We could either restrict the class *ICS* so as to admit efficient detection (as is done in [8,11]) or we can try to limit the size of the lattice. It is well known that certain communication patterns in the computation will lead to very "lean" lattice structures avoiding exponential number of states. Another possibility is to avoid examining the entire lattice by employing certain heuristics in the detection algorithms for constructing the set *current* from the set *previous*. In this case, detection of **Def** Φ and $\neg$**Pos** Φ will take on a probabilistic interpretation.

6 Conclusions

We have proposed two new classes of global predicates for expressing behavior patterns in distributed computations. These classes admit arbitrary boolean expressions over global states as building blocks and include temporal specifications as causal chains. The possibility to specify forbidden states between desired ones adds significantly to the expressiveness of our proposal.

We have developed in-line algorithms for detecting when computations satisfy formulas belonging to the two classes. The cost of our detection algorithms may be prohibitive for certain computations due to the size of the search space. We discussed several directions we are pursuing for limiting these costs.

References

[1] Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: fundamental concepts and mechanisms. In S.J. Mullender, editor, *Distributed Systems*, chapter 4. ACM Press, 1993.

[2] P.C. Bates and J.C. Wileden. High-level debugging of distributed systems: the behavioral abstraction approach. *Journal of Systems and Software*, 4(3):255–264, December 1983.

[3] L. Bouge. Repeated snapshots in distributed systems with synchronous communication. *Theoretical Computer Science*, 49:145–169, 1987.

[4] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[5] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, May 1991.

[6] C. Diehl, C. Jard, and F.X. Rampon. Computing on-line the covering graph of the ideal lattice of posets. Research Report 703, IRISA, University of Rennes, France, February 1993.

[7] C.J. Fidge. Partial orders for parallel debugging. *ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[8] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proceedings of the Twelfth International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes on Computer Science*, pages 253–264. Springer-Verlag, New Delhi, India, December 1992.

[9] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 166–175, January 1988.

[10] W. Hseush and G.E. Kaiser. Modeling concurrency in parallel debugging. *ACM SIGPLAN Notices*, 25(3):11-20, March 1990.

[11] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the ACM Conference on Parallel and Distributed Debugging*, San Diego, California, May 1993.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *Proceedings of the Fifth International Workshop on Distributed Algorithms (WDAG-91)*, Lecture Notes on Computer Science. Springer-Verlag, Delphi, Greece, October 1991.

[14] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 1993. To appear.

[15] B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 316–323, San Jose, California, July 1988.

[16] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the Holy Grail. Technical Report SFB124-15/92, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 1992.

[17] M. Spezialetti and J.P. Kearns. Efficient distributed snapshots. In *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[18] M. Spezialetti and J.P. Kearns. A general approach to recognizing event occurrences in distributed computations. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 300–307, San Jose, California, July 1988.

Unité de Recherche INRIA Rennes
IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399