Testing unstable properties in communication protocols

Antonio A.F. Loureiro *Osvaldo S.F. de Carvalho*

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Caixa Postal 702, 30161-970 Belo Horizonte, MG
E-mail: {loureiro,vado}@dcc.ufmg.br

Resumo

O teste de protocolos é um processo dinâmico que pode mostrar comportamentos válidos ou não durante a execução do protocolo num ambiente de teste. Neste trabalho nós estudamos o problema de validar propriedades dinâmicas instáveis durante o processo de teste e durante a execução normal do protocolo. Propriedades dinâmicas instáveis definem comportamentos temporais desejáveis ou não de um protocolo. Neste trabalho nós apresentamos um novo algoritmo para testar propriedades instáveis que funciona on-line de forma distribuída.

Abstract

Protocol testing is a run-time activity and we can only hope to detect valid or invalid behaviors in an actual execution of a protocol implementation embedded in a testing environment. In this paper we focus our attention in the validation of dynamic unstable properties during the testing process and afterwards, during normal execution. Dynamic unstable properties define desirable or undesirable temporal evolutions of the behavior of a communication protocol. We shall present a novel on-line distributed algorithm and the corresponding design principles that will improve the testing process and the reliability of a protocol implementation by checking unstable properties.

1 Introduction

The validation[1] of global predicates is a fundamental problem in distributed computing that has been used in different contexts such as design, simulation, testing, debugging, and monitoring of distributed programs [1, 7, 15, 16, 18]. A global predicate may either be stable or unstable. Informally, a stable predicate means that once it becomes true during a computation it will remain true after that, such as a system deadlock. An unstable predicate does not have this characteristic. An example is a predicate that relates the

[1] The terms checking, detection, and validation will be used interchangeably in this paper when referring to the validation of desirable behaviors (i.e., predicates or properties) in a protocol.

length of two queues, each one in a different process. In fact, an unstable predicate may switch from valid to invalid and vice-versa during the execution.

The initial work in the detection of global predicates has concentrated on the validation of stable properties such as distributed termination [5, 6] and deadlock detection [2]. Chandy and Lamport [3] proposed an algorithm to take snapshots in a distributed system that became the basis of other algorithms which check stable properties. The word *snapshot* in this algorithm means the local state of a process P_i in the system. Therefore, when P_i receives a message to take a snapshot it records its local state and "relays the 'take snapshot' message along all of its outgoing channels" [3]. These 'snapshot' messages are used by a global monitor to construct only consistent global states (see Definition 6).

Note that a stable property defined in terms of global predicates can be checked by a global monitor which takes snapshots and constructs the consistent global states. If the stable predicate is found to be true in at least one consistent global state constructed from the snapshots taken in the individual processes then it can be inferred that it will remain in that state at the end of the algorithm. If the predicate is false in the global states constructed, then it was also false at the beginning of the algorithm [3].

Unfortunately this approach does not work for unstable predicates which may be true during an execution but not checked, or found to be true in some states but it may have never happened because the global monitor constructs all possible consistent global states (see [1] for an example of this situation).

The testing process is a run-time activity and we can only hope to detect valid or invalid behaviors in an actual execution of a protocol implementation embedded in a testing environment. In this paper we focus our attention in the validation of dynamic unstable properties during the testing process and afterwards, during normal execution. Dynamic unstable properties define desirable or undesirable temporal evolutions of the behavior of a communication protocol. We shall present a novel algorithm and the corresponding design principles that will improve the testing process and the reliability of a protocol implementation by checking unstable properties. The goals to be accomplished are summarized as follows:

▹ Protocol testing:
- Mechanism to check global predicates based on local predicates.
- Identification of consistent global states where any type of predicate can be checked.

▹ Protocol execution:
- Obtain information to avoid the problem of state build-up ([11]).
- Obtain information for use in exception handling.

The rest of this paper is organized as follows. Section 2 describes the formal model used in this paper. Section 3 discusses the tasks involved in the detection of dynamic properties. Section 4 describes the design principles related to the testing process, including the algorithm to detect the properties. Section 5 describes the design principles related to the execution of the protocol implementation. Section 6 discusses the related work. Finally, Section 7 presents the conclusions for this paper.

2 Formal model

In this section we present some definitions that will be used in the algorithm described in Section 4. Our algorithm is based on the communicating extended finite state machine

(CEFSM) model. First we present the nomenclature for identifiers that will be used in this paper.

Nomenclature 1 (Identifiers) Identifiers can have a subscript and/or a superscript both of which are used to indicate an element in a set. Let $\Box_i^x$ represent an identifier. The subscript i refers to the i-th element of $\Box$. For example, process P_i. The superscript refers to the x-th element of $\Box_i$ in the case $\Box_i$ is also a set.

Definition 2 (Communicating extended finite state machines)
A communicating extended finite state machine is a labeled directed graph where vertices represent states and edges represent transitions. A designated vertex represents the initial state of the machine. Transitions are labeled with the pair $\frac{event}{action}$. An event is the sending of a message, the reception of a message, or an internal event not related to a channel (e.g., a timeout or a service request from a service user). Messages should be defined in a finite set Σ of message types. The communication between machines is asynchronous. The communicating channels between each pair of machines are not perfect so that messages can be reordered, corrupted or lost. The channels are assumed to have infinite capacity.[2]

Formally, a communicating extended finite state machine P_i $(i = 1 \ldots r)$ is a five-tuple

$$P_i = (S_i, \Sigma_i, \delta_i, \lambda_i, s_i^0),$$

where

- S_i is the set of local states of machine P_i.
- Σ_i is the set of message types that machine P_i can exchange with other machines. This is represented by the sets $\{\Sigma_{i,j}\}$ and $\{\Sigma_{j,i}\}$, respectively. Therefore, $\Sigma_i = \{\Sigma_{i,j}\} \cup \{\Sigma_{j,i}\}$. The set $\{\Sigma_{i,i}\}$ is empty, i.e., machine P_i cannot send or receive messages from itself.
- δ_i is the transition function and is defined as δ_i: $S_i \times \Sigma_i \rightarrow S_i$.
- λ_i is the output function and is defined as λ_i: $S_i \times \Sigma_i \rightarrow \Sigma_i$.
- s_i^0 is the initial state of machine P_i.

In the labeled directed graph that represents a CEFSM, a message preceded by a + sign means that it was received, and a message preceded by a − sign indicates that it was sent. ■

Let P be a protocol specification comprised of processes $P_1, P_2, \ldots, P_r$. Each process is modeled as a CEFSM and they are interconnected by a set of communicating channels $C_1, C_2, \ldots, C_s$. Each process P_i $(i = 1 \ldots r)$ has a finite set of variables $V_i^1, V_i^2, \ldots, V_i^k$. In this model, the concept of a global state plays a fundamental role in the correctness of a communication protocol.

Definition 3 (Causal precedence order) A causal precedence order defines a partial order of events in a system P. Let E be the set of all events that can occur in P and let

[2]In practice, we can model an infinite buffer using a finite buffer by discarding new incoming messages when the buffer is full.

$\prec$ ("happens before") be the binary relation denoting causal precedence between events as defined by Lamport [9]. Therefore, we have:

$$e_i^a \prec e_j^b \stackrel{\text{def}}{=} \begin{cases} \text{(i): } (i = j) \wedge (b = a + 1) \vee \\ \text{(ii): } (e_i^a = \textbf{send } \langle m \rangle \textbf{ to } j) \wedge (e_j^b = \textbf{rcv } \langle m \rangle \textbf{ from } i) \vee \\ \text{(iii): } \exists\, e_k^c : (e_i^a \prec e_k^c) \wedge (e_k^c \prec e_j^b) \end{cases}$$

Condition (i) says that all events that occur in P_i are totally ordered. Condition (ii) says that if we consider a message $\langle m \rangle$ then the sending event precedes its receiving event. And condition (iii) says that it is possible to define chains of related events based on causality, i.e., relation $\prec$ is transitive. ■

A protocol computation can be represented by a partially ordered set,[3] or poset for short, based on set E, i.e., $\mathcal{C}_P = (E, \prec)$. Graphically, we can represent a distributed computation using the space-time diagram, as shown in Figure 1.

If we analyze this space-time diagram we can realize that process P_i enters in a local state s_i^x after event e_i^x happens. It is easy to see that there is a duality between events and local states in this computation. Let S be the set of states in the computation $\mathcal{C}_P$. Therefore, we can rewrite the binary relation $\prec$ as follows:

$$s_i^a \prec s_j^b \stackrel{\text{def}}{=} \begin{cases} \text{(i): } (i = j) \wedge (b = a + 1) \vee \\ \text{(ii): } (e_i^{a+1} = \textbf{send } \langle m \rangle \textbf{ to } j) \wedge (e_j^b = \textbf{rcv } \langle m \rangle \textbf{ from } i) \vee \\ \text{(iii): } \exists\, s_k^c : (s_i^a \prec s_k^c) \wedge (s_k^c \prec s_j^b) \end{cases}$$

All three conditions have the same meaning as before but in this case we have states instead of events. Therefore, the protocol computation can be represented by another poset based on the set S, i.e., $\mathcal{C}_P = (S, \prec)$.

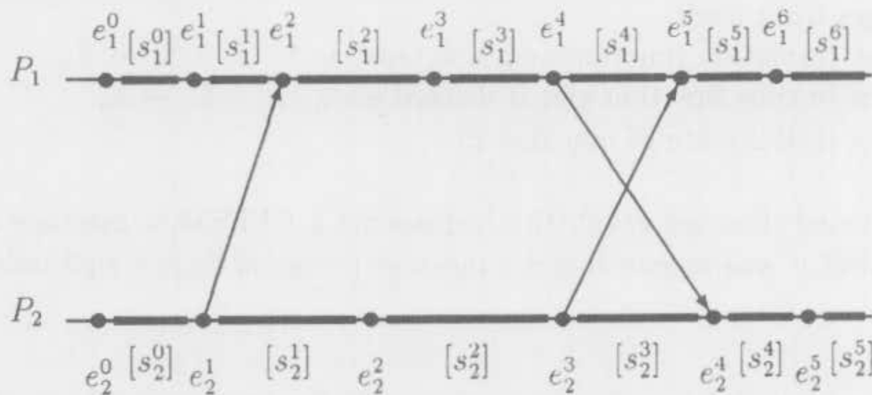


Figure 1: Example of a space-time diagram representing a distributed computation based on states.

[3]Since an event cannot happen before itself, the $\prec$ relation is an irreflexive partial ordering or a precedence relation, i.e., antisymmetric and transitive. Precedence relations share many of the properties of partial ordering relations. However, when a physical situation leads to the definition of a precedence relation, it is often expedient to include the reflexivity property so the terminology and results in connection with partial ordering relations can be applied [10].

Definition 4 (Local state) The local state γ_i of a process P_i is defined as the contents of each local variable $(V_i^1, \ldots, V_i^{k_i})$ in P_i. ∎

In the following, we give a definition of global states that does not include communication channels which can be encoded as part of each local state.

Definition 5 (Global state) A global state is an n-tuple of local states, one for each process P_i, and is represented as follows:

$$\Gamma = (\gamma_1, \ldots, \gamma_n)$$

where γ_i represents the local state of process P_i. ∎

Definition 6 (Consistent global state) Informally, a global state is consistent if it could occur during an execution and a global clock in the system could be used to label precisely the total order of events. Formally, a global state $\Gamma = (\gamma_1, \ldots, \gamma_n)$ is consistent iff for any pair $(\gamma_i, \gamma_j) \in \Gamma$, then either $\gamma_i \prec \gamma_j$ or $\gamma_j \prec \gamma_i$ for $1 \leq i, j \leq n$ and $i \neq j$, according to the transitive closure of relation $\prec$ between local states. ∎

The set of all consistent global states define exactly the states that could have happened in any computation with respect to the events that occurred in each process. Therefore, predicate values are meaningful only if evaluated in a consistent global state.

The set of all consistent global states Γ define a lattice structure $\mathcal{L}$ and its minimal element is the initial global state $\Gamma^0 = (\gamma_1^0, \ldots, \gamma_n^0)$. In the lattice $\mathcal{L}$ there is an edge - 1zfrom a node representing a global state $\Gamma^\alpha = (\gamma_1, \ldots, \gamma_i^a, \ldots, \gamma_n)$ to a node representing $\Gamma^{succ(\alpha)} = (\gamma_1, \ldots, \gamma_i^{a+1}, \ldots, \gamma_n)$ iff there exists an event ϵ that P_i can execute in its state γ_i^a.

Definition 7 (Sequence of global states) Informally, a sequence of global states represents a computation where the order of each global state in the sequence is given according to a global clock. Thus this sequence represents the serialization of the global states in a particular computation. Formally, a sequence of global states $\Gamma^0, \Gamma^1, \ldots, \Gamma^{\alpha-1}, \Gamma^\alpha, \ldots$ represents a sequence of events $e^1, e^2, \ldots$ that is consistent with the relation $\prec$. In this sequence, global state Γ^α is reached after executing event e^α in global state $\Gamma^{\alpha-1}$. ∎

From this definition we can see that there is a duality between sequences of global states and sequences of events. In other words, a sequence of global states define a possible computation of P where the events are implicit, and a sequence of events also define a possible computation of P where the states are implicit. In either case we have a valid computation of P that starts at the minimum element of lattice $\mathcal{L}$ and goes upwards along one path. Furthermore, this lattice of consistent global states represents all valid observations of P.

Now we present the definitions of local and global predicates that will be used in Section 4.

Definition 8 (Local predicate) A local predicate of a process P_i is a formula in propositional logic (i.e., a boolean expression) where each term of the formula is a local variable of P_i. The set of local predicates ϕ valid for protocol P can be expressed as:

$$\begin{aligned} \phi &= \{\phi_1^1, \ldots, \phi_1^{p_1}, \phi_2^1, \ldots, \phi_2^{p_2}, \ldots, \phi_n^1, \ldots, \phi_n^{p_n}\} \\ &= \bigcup_{i=1..n} \phi_i \end{aligned}$$

∎

Definition 9 (Global predicate) A global predicate of protocol P is a formula in propositional logic (i.e., a boolean expression) where each term of the formula is a local predicate of process P_i. A set of global predicates Φ valid for protocol P can be expressed as:

$$\Phi = \{\Phi_1, \Phi_2, \dots, \Phi_t\}$$

■

3 Tasks involved in the detection of unstable dynamic properties

The detection of unstable dynamic properties involves two tasks. The first one defines the ways a property (global predicate) can be expressed, and the second is the design of algorithms to detect such properties. Clearly the rules used for expressing the properties will guide the design of algorithms to detect such properties. For instance, a property can be defined in a general way [1, 4] involving relations among variables in different processes. In this case we need to identify all the consistent global states, which can be represented by a lattice, to determine whether the property definitely or possibly occurred [1].

In Section 6 we compare the different solutions proposed in the literature to represent predicates and their algorithms and the representation used in this thesis with the algorithm proposed in Section 4.3.

4 Design principles related to testing

This section has four parts. The first part describes how dynamic properties are represented (Section 4.1). The second part explains the basic principles used to detect dynamic properties (Section 4.2). The third part describes in details the algorithm to check dynamic properties based on local predicates (Section 4.3). Finally, the fourth part shows that for a specific type of protocol, namely 3-way handshake protocols, we can compute general properties using this algorithm (Section 4.4).

4.1 Representation of dynamic properties

Recall from the discussion in Section 1 that we are interested in checking dynamic properties during the testing phase as well as during normal execution of the protocol after the testing process. In this case, the representation of dynamic properties must consider two important requirements. First we need to check properties continuously, and second, a dynamic property should describe a valid trace for the protocol specification.

It is well known from automata theory that the languages accepted by finite automata are the languages defined by regular expressions. Furthermore, regular expressions can define infinite languages which, in our case, represent the valid traces for the protocol specification. From Definition 9 we can see that a global predicate p is a regular expression which defines a finite trace. If we want to extend the global predicate to represent an infinite language, we must include the operation for concatenation of expressions represented by the symbol $+$ as in p^+.

Definition 10 (Dynamic property) A dynamic property is a global predicate expressed in the form given in Definition 9 which may contain the operation for concatenation of expressions represented by the symbol +. ∎

The way a global predicate is expressed is different from equivalence of a regular expression L and a finite automaton A_L in two aspects. First, in the case of language recognition there is only one symbol to be processed at each state of the automaton A_L. In the case of property detection we shall see that we may have a set of valid local predicates in each state of the automaton A_Φ that corresponds to the property Φ. Second, a dynamic property can only use the symbol + for concatenation of expressions and not the symbol *. (See the discussion following Theorem 13 for not including the symbol *.)

From Definition 10 it follows that a dynamic property can be expressed by a deterministic finite automaton (DFA). In fact an automaton is a convenient representation for properties since:

- it represents the nature of communication protocols, that is, of reactive systems;
- it is compatible with the communicating extended finite state machines model used in this paper to represent protocols;
- it allows the protocol behavior to be partitioned, that is paths and phases, where paths represent partial traces.

Definition 11 (Deterministic finite automaton) A deterministic finite automaton (DFA) is a directed graph where each transition represents a local predicate, and each automaton state represents an evaluation of the global predicate in some state of the process P_i.

Formally, a deterministic finite automaton A_j $(j = 1 \ldots t)$ is a five-tuple

$$A_j = (Q_j, \Sigma_j, \delta_j, s_j^0, QF_j),$$

where

- Q_j is the set of states of automaton A_j.
- Σ_j is the set of local predicates associated with A_j.
- δ_j is the transition function and is defined as δ_j: $Q_j \times \Sigma_j \rightarrow Q_j$.
- q_i^0 is the initial state of automaton A_j.
- QF_j is the set of accepting states of automaton A_j. ∎

Figure 2 shows a binary relation between local predicates and global predicates. Let the binary relation μ_i $(i = 1 \ldots n)$ from ϕ_i to Σ be defined by

$$R_{\mu_i} = \{(\phi_i^x, \Sigma_j), \ \phi_i^x \in \phi_i, \ \Sigma_j \in \Sigma \ \mid \ \phi_i^x \text{ is a term of } \Phi_j\}.$$

The equivalent matrix representation M_{μ_i} for R_{μ_i} is given in Figure 3. The entry $M_{\mu_i}[\phi_i^x, \Sigma_j] = 1$ iff ϕ_i^x is a term in the definition of the global predicate Φ_j. Otherwise it is zero.

Note that if column j of M_{μ_i} has all entries equal to zero it means that predicate Φ_j does not contain any local predicate of ϕ_i and this column can be removed. If line x of M_{μ_i} has all entries equal to zero it means that predicate ϕ_i^x does not appear in any global predicate. The designer should analyze this problem which may be an indication of error.

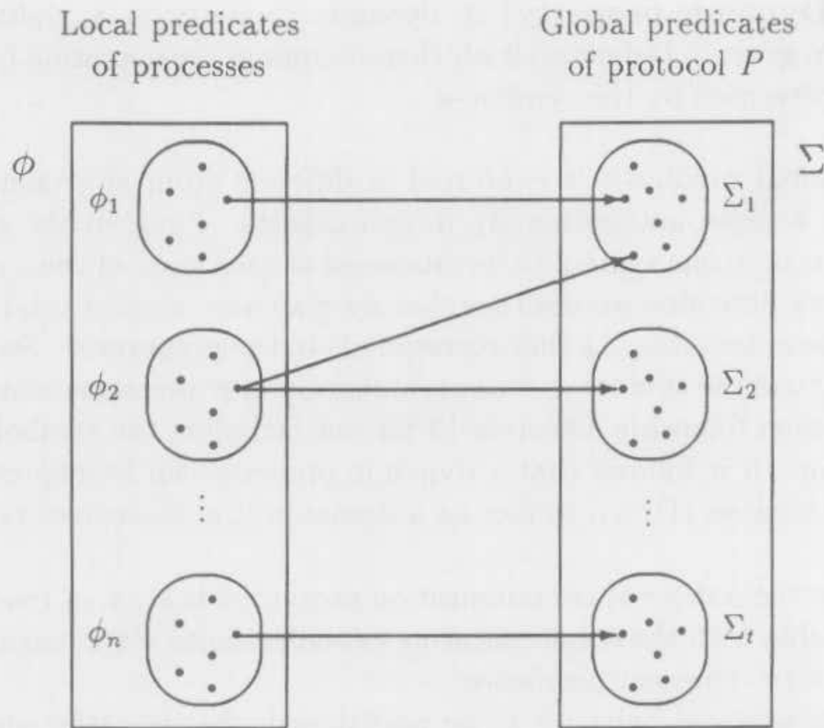


Figure 2: Binary relation R_{μ_i} between local and global predicates.

$M_{\mu_i} =$

	Σ_1	$\dots$	Σ_j	$\dots$	Σ_t
ϕ_i^1					
$\vdots$					
ϕ_i^x			1		
$\vdots$					
$\phi_i^{p_i}$					

Figure 3: Matrix representation of binary relation R_{μ_i}.

4.2 Detection of dynamic properties

To detect a property we will build the automaton A_j that represents the global predicate Φ_j. The idea of the detection algorithm is to superimpose onto each process P_i $(i = 1 \dots n)$ that implements protocol P a local checking procedure that repeatedly performs the following steps:

(i) check and determine the valid local predicates in the set ϕ_i when process P_i goes to a new state, and move the automaton A_j to a new state if there are transitions associated with the valid local predicates;

(ii) append the current state of the automaton A_j^i when P_i sends a message to another process; and

(iii) check the local predicates as in (i) when P_i receives a message from P_k and move the automaton A_j to a new state based on the current state and the state received from P_k.

The basic idea of this procedure is to use the automaton A_j to keep track of the protocol behavior. If the behavior exhibited by the protocol is valid, the automaton A_j, which represents property Φ, will eventually reach an accepting state.

Note that this high level algorithm can be applied to any protocol or distributed program in general. In Section 4.3 we shall describe this algorithm in detail. The properties to be checked depend on each protocol and we shall assume that they are part of the input to the algorithm.

In the following we apply this high level algorithm to an example. Suppose we have two processes P_1 and P_2 that implement a protocol P such as the ISO Transport Protocol. Furthermore, each process implements a different class of the transport protocol. Given this scenario we would like to make sure, for instance, that the parameters negotiated during the connection establishment remain valid for the entire connection. Let $\phi_1 = \{\phi_1^1, \phi_1^2, \phi_1^3\}$ and $\phi_2 = \{\phi_2^1, \phi_2^2, \phi_2^3, \phi_2^4\}$ represent the set of local requirements for processes P_1 and P_2 respectively. Let us assume that the property that says that "the parameters selected must remain valid for the entire connection" can be expressed as $\Phi_j = (\phi_2^1 \vee \phi_2^2)^+ \wedge \phi_1^1 \wedge \phi_1^2 \wedge (\phi_2^4)^+$. This predicate can be represented by the automaton in Figure 4-(a). Figure 4-(b) depicts a possible computation for this protocol with the relevant events and messages shown. Next to each event is the set of local predicates that hold at that moment. The current state of the automaton is given inside a circle. Without loss of generality, assume that the protocol P executed the following computation according to the lattice of consistent global states as shown in Figure 4-(c):

$$(0,0), (0,1), (1,1), (1,2), (2,2), (2,3), (3,3), (4,3), (4,4)$$

The initial state of the automaton for both processes is q_j^0. When the event e_2^1 occurs the state of the automaton A_j in P_2 goes to q_j^1 after checking the local predicates. At this point, a message $\langle m, q_j^1\rangle$ is sent to P_1. Process P_1 receives the message and checks its local predicates. At this moment P_1 is in state q_j^0 and P_2 in state q_j^1. Note that the only possible transition in the automaton of P_1 is (q_j^1, ϕ_1^1, q_j^2) and therefore this transition is executed. If the local predicate ϕ_1^1 is not valid when event e_1^1 happens then we must take the automaton to its initial state since we cannot have a valid computation that started at a valid state with an invalid prefix. Then computation proceeds as before. When the event e_2^4 occurs, the state of the automaton A_j in P_2 goes to q_j^4 after checking the local predicates. At this point, a message $\langle m, q_j^3\rangle$ was received from P_1. Again, the only possible transition in the automaton of P_2 is (q_j^3, ϕ_2^4, q_j^4) and therefore this transition is executed. When the state q_j^4 is reached, the property Φ_j is true in process P_2. Process P_1 will eventually detect this property if the local predicates associated with Φ_j remain true (i.e., ϕ_1^1 and ϕ_1^2) and P_2 sends a message to P_1. If the former condition does not hold it just reflects the temporal behavior of the protocol. If the latter condition does not happen it is because the protocol was not designed to send another message to P_1.

There are two important points that should be noted in the algorithm proposed. First, the properties do not specify interleaved behaviors of the protocol but conditions that should hold with time. This can be seen from the lattice of consistent global states of the distributed computation as shown in Figure 4-(c). All possible interleavings can

be obtained from the lattice. Independent of which path of computation occurs, the property will hold iff along the path the local predicates of each process are valid when the checking is performed. Second, the checking procedure does not modify the behavior given by the protocol specification, it simply appends the current state of the automaton to each message sent.

4.3 Algorithm to check dynamic properties based on local predicates

The algorithm to check dynamic properties is given in Figure 5 and is divided into several parts as explained below.

4.3.1 Data structures and algorithm

▷ Data structures

The data structures used in the algorithm are described in the following:

- *StateOfAutomaton*$[1 \ldots t]$: each entry of this array represents the current state of the Automaton A_j $(j \ldots t)$ in the Process P_i.
- $LP[1 \ldots p_i]$: the x-th entry ($x = 1 \ldots p_i$) of this array indicates whether the local predicate ϕ_i^x is true at the current state of Process P_i.
  The variable p_i represents the cardinality of the set ϕ_i as given in the Definition 8.
- *ValidLP*: set of valid local predicates when process P_i moves to a new state.
- *EndOfTransition*: boolean variable that indicates whether or not there is no more transition to be executed.

▷ Initialization: Lines 1–3

Initializes each automaton to its initial state that is by definition the state q_j^0.

▷ Main part: Lines 4–9

This is the main part of the program that repeatedly checks the properties when the process P_i goes to a new state or appends the array *StateOfAutomaton* to $\langle m \rangle$ when a message is sent.

The part that checks the properties can be seen as a function and is given in lines 10 to 46.

▷ Check each local predicate defined in P_i: Lines 10–12

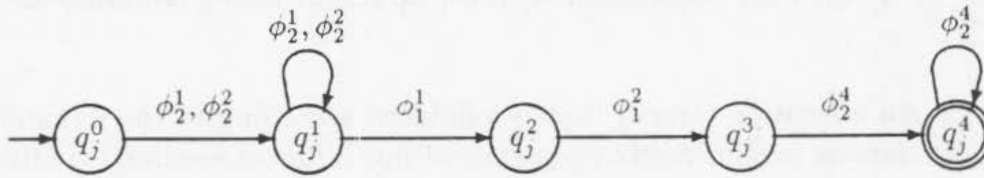
When process P_i goes to a new state, we need to check each local predicate in order to evaluate the global properties. The truth-value of each local predicate depends on its definition. We give a generic function called *CheckLP* that checks a local predicate.

▷ Validation of global properties: Lines 13–44

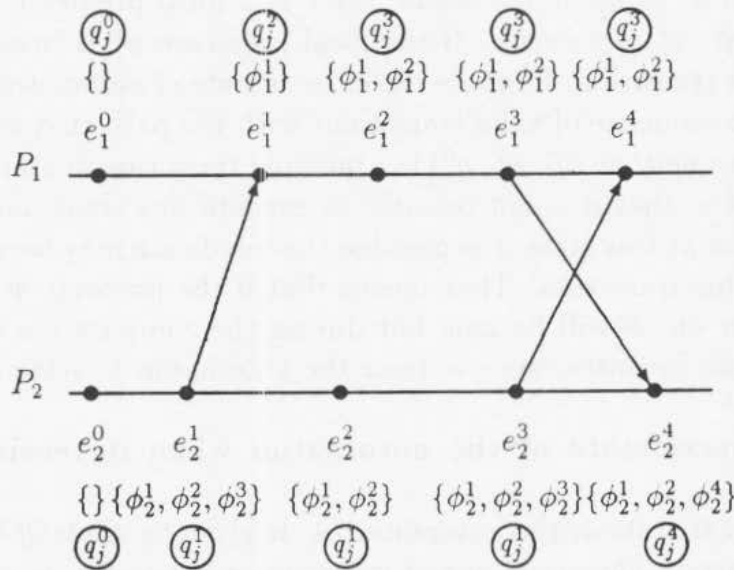
In this part we check each property in the set Φ.

▷ Determine the set of valid local predicates in Φ at the current moment: Lines 14–23

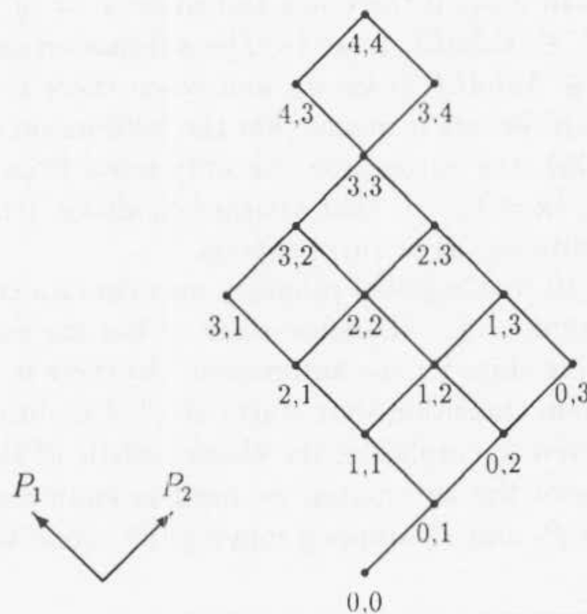
An automaton A_j^i can only execute a transition for a local predicate ϕ_i^x iff this local



(a) Global property and the corresponding DFA.



(b) Time-space diagram of the distributed computation with the valid local predicates at each state.



(c) Lattice of consistent global states of the distributed computation.

Figure 4: Checking of properties in a distributed computation.

predicate is a term in Φ_j and is valid at the current moment in the process P_i. If the local

predicate is a term of Φ_j but is invalid and there is a transition associated with ϕ_i^x at the current state of A_j^i then the automaton A_j^i must be reset. This guarantees the validity of Theorem 12.

Theorem 12 An execution (trace) that is validated according to the dynamic property Φ does not contain an invalid prefix (sequence of invalid local predicates) after its initial state q_j^0.

Proof. By definition, the initial state q_j^0 is a valid state for the automaton A_j. If A_j changes its state from q_j^0 to q_j^1, it is because there is a local predicate ϕ_i^x valid at state q_j^0 and a transition (q_j^0, ϕ_i^x, q_j^1) exists. If the local predicate ϕ_i^x is invalid then it is not possible to execute the transition. Suppose the current state of automaton A_j is q_j' and it is possible to associate a sequence of valid transitions with the path that led the automaton to this state. Now suppose that (q_j', ϕ_i^x, q_j'') is a possible transition at state q_j' but the local predicate ϕ_i^x is invalid so that it is not possible to execute this transition at this time. If we leave the automaton at this state it is possible this predicate may become true later on and we will execute this transition. That means that if the property Φ eventually holds after the state q_j' later on, Φ will be true but during the computation there was a false condition. To avoid this inconsistency, we reset the automaton to state q_j^0. □

▷ Determine the next state of the automaton when it receives a message: Lines 24–26

At this point the current state of the automaton A_j is given by *StateOfAutomaton*[j] and a new state q_j^x is received. Therefore, one of these states has to be chosen so the process of checking the property can be carried on. Before showing how this state is determined. let us examine the situations in which an automaton A_j can and cannot move.

The automaton in P_i can move if there is a transition $\mathcal{T} = (q_j', \phi_i^x, q_j'')$ where q_j' is the current state of A_j^i and $\phi_i^x \in$ *ValidLP* (case i). The automaton cannot move when there is a transition $\mathcal{T}$ but $\phi_i^x \notin$ *ValidLP* (case ii), and when there is no transition $\mathcal{T}$ where $\phi_i^x \in \phi_i$ (case iii). If case (ii) occurs it means that the automaton must be reset as shown in Theorem 12. In case (iii), the automaton can only move from this state if there is a predicate ϕ_k^x in process P_k ($k = 1 \ldots n$) that satisfies condition (i). The automaton must move until one of the conditions (ii) or (iii) happens.

Recall from Definition 10 that a global property may contain concatenation of expressions represented by the symbol $+$. Therefore state q_j^0 has the following characteristics: (a) it cannot be an accepting state for the automaton; (b) there is no transition that ends at q_j^0; and (c) it has just one transition that starts at q_j^0. Condition (a) follows from (b) but it was intentionally given to emphasize the characteristic of this state.

To determine the state of the automaton we need to enumerate the possible combinations in which processes P_1 and P_2 stopped moving (i.e., conditions (ii) or (iii) above). The combinations are:

Combination	1	2	3	4
P_1	(ii)	(ii)	(iii)	(iii)
P_2	(ii)	(iii)	(ii)	(iii)

Condition (ii) means a reset and condition (iii) means that the move depends on the other process.

Suppose that P_1 is the process that receives a message. (For P_2 is equivalent.) The first combination is trivial and the state of the automaton A_j^1 remains unchanged. The fourth combination shows that the next state of A_j^1 must be the state of the automaton A_j^2 received from the process P_2. The second and third combinations are symmetric. Let us analyze the second combination. The automaton A_j^1 is in the state q_j^0 and the automaton A_j^2 is in the state q_j^x. If the state q_j^x in A_j^2 can be reached directly from state q_j^0 (i.e., with transitions involving only local predicates in P_2) then this situation is similar to the fourth combination and the next state of the automaton must be q_j^x. However, if the state q_j^x cannot be reached directly from q_j^0, it means that process P_1 has already contributed to this path, i.e., there is at least one transition with a local predicate of P_1 before reaching state q_j^x. Since there is no transition that ends at q_j^0 (condition (b) above) it means that the automaton A_j^1 was reset and therefore the next state of A_j^1 must be q_j^0 according to Theorem 12. This analysis also applies to the third combination.

Theorem 13 When a process P_i receives a message with the state of the automaton A_j in P_k the next state of automaton A_j in P_i can be uniquely determined.

Proof. Given above. □

Note that if we allow the concatenation of expressions represented by the symbol $*$ in the definition of global properties, then the conditions (a), (b), and (c) above do not hold anymore and we cannot apply Theorem 13 directly as stated. This does not seem to be an important restriction since if a local property can happen zero or more times, it is probably meaningful if it has to happen at least once.

▷ Determine whether the automaton can move: Lines 27–40

At this point we know the set of valid local predicates and the current state of the automaton A_j and would like to determine if the automaton can move to a new state. This must be done following the conditions (i), (ii), and (iii) described above.

▷ Check whether the current state of the automaton satisfies the global property: Lines 41–43

If the current state of the automaton A_j is an accepting state then the property Φ is valid at the current state of process P_i.

4.3.2 Complexity of the algorithm

We provide an analysis of the complexity of the algorithm to detect one property Φ_j in lines 13 to 44. Let p_i be the cardinality of the set of local predicates ϕ_i and e the cardinality of the set δ_j. The algorithm has five sequential parts. The first part (lines 10–12) validates each local predicate ϕ_i^x. Suppose that each predicate can be checked in constant time. Then the validation of the local predicates can be carried out in time $O(p_i)$. The second part (lines 14–23) determines the set of valid local predicates in Φ_j which is also executed in time $O(p_i)$. The third part (lines 24–26) determines the next state of the automaton when it receives a message. There are four cases to be analyzed and this part is executed in constant time. The fourth part (lines 27–40) moves the automaton A_j^i, if possible. Since this depends on the number of transitions in A_j this is bounded by $O(e)$. The fifth part checks if the current state of the automaton satisfies the global property

Begin of algorithm to check dynamic properties

Input:
- Set $A = \{A_1, A_2, \ldots\}$ of automata that represents the set of properties $\Phi = \{\Phi_1, \Phi_2, \ldots\}$ to be detected by process P_i.
- Set $\phi_i = \{\phi_i^1, \ldots, \phi_i^{p_i}\}$ that represents the local predicates valid for process P_i.
- Matrix M_{μ_i} that represents the binary relation R_{μ_i}.

Output:
- Validation of each local predicate in ϕ_i.
- Validation of each dynamic property represented by an automaton in the set A.

```
     /* Initialization */
(1)  foreach automaton A_j ∈ A do
(2)     StateOfAutomaton[j] ← q_j^0;
(3)  od;

     /* Main part */
(4)  do forever
(5)        (when process P_i goes to a new state) ⟼
(6)           Check properties;
(7)        (when process P_i sends a message ⟨m⟩) ⟼
(8)           Appends the array StateOfAutomaton to ⟨m⟩;
(9)  od;
```

↪

Figure 5: Algorithm to check dynamic properties (Part 1 of 3).

and this can done in constant time. Therefore, the five parts together are bounded by $O(p_i + e)$.

4.4 Special global states in the computation

In [1] Babaoğlu and Marzullo show that a property Φ involving relations among variables in different autonomous[4] processes (i.e., general predicates) can only be detected by building the lattice of consistent global states and then traversing it to determine whether Φ definitely or possibly is true. This problem arises because we are considering asynchronous distributed systems where the processes are autonomous and may execute at different speeds.

In a synchronous distributed system the coordination among processes happens in global synchronization points. Intuitively this means that the computations of the co-operating processes participating in a synchronization converge to a single global state since all computations have to reach that specific point (state). At that synchronization point, general properties can be evaluated. In asynchronous distributed systems there is a similar situation if there is only one autonomous process P_A participating in a synchronization and P_A is the last process to enter the synchronization state. This is exemplified

[4] A non-autonomous process can only initiate a communication in response to a message received.

```
        /*** Check properties ***/
        /* Validation of each local predicate defined in P_i */
(10)    foreach local predicate φ_i^x ∈ φ_i do
(11)        LP[φ_i^x] ← CheckLP(φ_i^x);
(12)    od;
        /* (P1) */

        /* Validation of global properties */
(13)    foreach automaton A_j ∈ A do

            /* Determine the set of valid local predicates in Φ at the current moment. */
(14)        ValidLP ← {};
(15)        foreach local predicate φ_i^x ∈ φ_i do
(16)            if M_{μ_i}[φ_i^x, Σ_j] = 1 then
(17)                if LP[φ_i^x] = true then
(18)                    ValidLP ← ValidLP ∪ {φ_i^x};
(19)                else /* go to the next iteration */
(20)                fi;
(21)            else /* go to the next iteration */
(22)            fi;
(23)        od;
            /* At this point set "ValidLP" contains the local predicates valid at the current
            moment in P_i. */
                                                                                    ↪
```

Figure 5: Algorithm to check dynamic properties (Part 2 of 3).

in Figure 6 with two processes.

Suppose process P_2 is autonomous and P_1 is not. To execute a service in this protocol, P_2 sends a request to P_1 which may reply with a positive or negative response. If the response is positive, P_2 can go to a state $\mathcal{S}_2$ that represents "service accepted." Note that process P_1 is already in a state $\mathcal{S}_1$ that represents "service accepted" by the time P_2 receives the response message. Therefore P_2 was the last process to enter state $\mathcal{S}$ that is our synchronization state. In the executions of Figures 6-(a) and 6-(b) a positive response is represented by events $\boxed{e_2^2}$, and $\boxed{e_2^4}$ and $\boxed{e_2^5}$ respectively. The global states corresponding to these events in the time-space diagrams are in the lattices. This type of protocol is often called 3-way handshake.

Independent of which computation sequence occurred that contains state $\mathcal{S}_2$ (e.g., $[\ldots,(3,3),(4,3),(4,4),\ldots]$ or $[\ldots,(3,3),(3,4),(4,4),\ldots]$ in Figure 6-(b)), we can check general properties at this state since P_1 is in state $\mathcal{S}$ as well. Furthermore, P_1 is not autonomous and therefore will remain at $\mathcal{S}_1$. The same argument can be extended to systems consisting of two or more processes that are non-autonomous and only one autonomous

```
        /* Determine the next state of the automaton if a message ⟨m, q_j^r⟩ was received
        from P_k (k = 1...n ∧ k ≠ i). */
(24)    if received message ⟨m, q_j^r⟩ then
(25)        StateOfAutomaton[j] ← state according to Theorem 13;
(26)    fi;

        /* Determine whether the automaton can move */
(27)    EndOfTransition ← false;        δ'_j ← δ_j;
(28)    while ¬EndOfTransition do
(29)        if (∃ (StateOfAutomaton[j], φ_i^r, q'_j) ∈ δ'_j)
(30)            if φ_i^r ∈ ValidLP then
(31)                if StateOfAutomaton[j] = q'_j then
(32)                    δ'_j ← δ'_j − (q'_j, φ_i^r, q'_j);
(33)                fi;
(34)                StateOfAutomaton[j] ← q'_j;
(35)            else StateOfAutomaton[j] ← q_j^0;
(36)                EndOfTransition ← true;
(37)            fi;
(38)        else EndOfTransition ← true;
(39)        fi;
(40)    od;

        /* Check whether the current state of the automaton satisfies the global property.
        */
(41)    if StateOfAutomaton[j] ∈ QF_j then
(42)        /* Global predicate Φ_j is valid at this state. */
            /* (P2) */
(43)    fi;
(44) od;
```

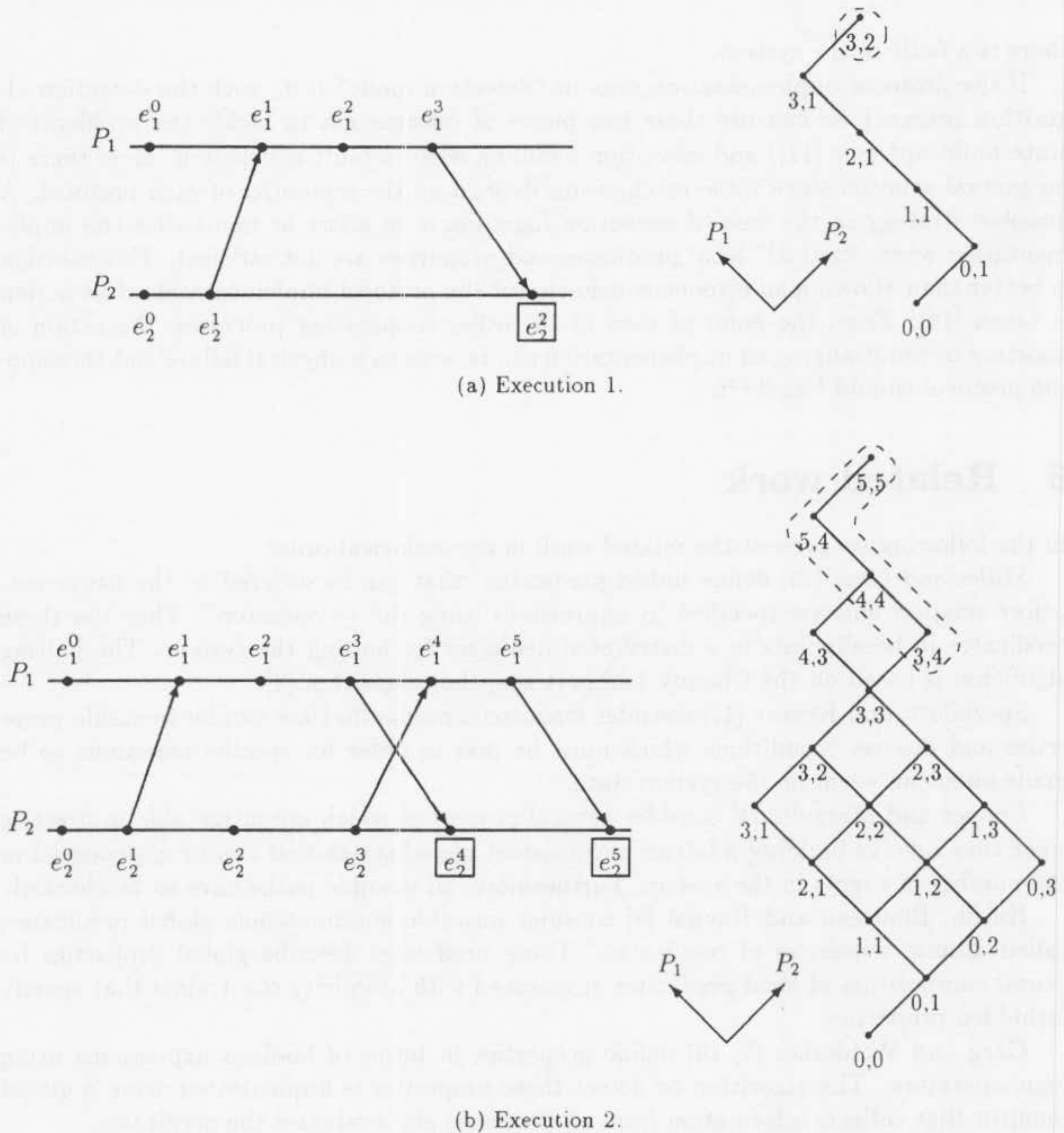
End of algorithm to check dynamic properties

Figure 5: Algorithm to check dynamic properties (Part 3 of 3).

process. This leads to the following theorem.

Theorem 14 3-way handshake protocols with two or more non-autonomous processes and only one autonomous process have a global state where general properties can be checked.

Proof. Given above.



(a) Execution 1.

(b) Execution 2.

Figure 6: Special global states in asynchronous distributed systems.

5 Design principles related to the execution

The algorithm to detect dynamic properties in Figure 5 provides two types of information when process P_i goes to a new state:

1. if each local predicate $\phi_i^x \in \phi_i$ is valid or not, and
2. if each global property $\Phi_j \in \Phi$ is valid or not.

The first information is provided in point (P1) and the second in point (P2) of the algorithm.

The fault/failure model [13] relates program inputs, faults, data state errors, and failures. A failure is a manifestation of a fault. Although it may or may not occur when

there is a fault in the system.

If the protocol implementation runs in "detection mode" (i.e., with the detection algorithm present) we can use these two pieces of information to tackle the problems of state build-up[5] (see [11]) and exception handling when a fault is detected. Here there is no general solution since these mechanisms depend on the semantics of each protocol. A possible strategy in the case of exception handling is to abort or reinitialize the implementation when "critical" local predicates and properties are not satisfied. This solution is better than allowing an erroneous behavior of the protocol implementation if no action is taken [14]. From the point of view of the other cooperating processes, the action of aborting or reinitializing an implementation can be seen as a physical failure and therefore the protocol should handle it.

6 Related work

In the following we present the related work in chronological order.

Miller and Choi [12] define linked predicates "that can be ordered by the happened-before relation and are specified by expressions using the $\rightarrow$ operator." They use these predicates in breakpoints in a distributed debugger for halting the system. The halting algorithm is based on the Chandy-Lamport snapshot algorithm [3].

Spezialetti and Kearns [17] consider monotonic events that are similar to stable properties and discuss "conditions which must be met in order for specific assertions to be made about an event or the system state."

Cooper and Marzullo [4] consider general properties which are intractable in practice since they involve building a lattice of consistent global states that can be exponential in the number of events in the system. Furthermore, all possible paths have to be checked.

Hurfin, Plouzeau and Raynal [8] consider unstable nonmonotonic global predicates, called atomic sequences of predicates. These predicates describe global properties by causal composition of local predicates augmented with atomicity constraints that specify forbidden properties.

Garg and Waldecker [7, 19] define properties in terms of boolean expressions using logic operators. The algorithm to detect these properties is implemented using a global monitor that collects information from all processes and evaluates the predicates.

Venkatesan and Dathan [18] also define properties in terms of boolean expressions and give a distributed algorithm to detect these properties. However, the evaluation of properties is performed off-line and they assume that executions of the system are reproducible. Furthermore, they only consider FIFO channels.

The algorithm proposed in this paper considers properties expressed as boolean expressions with concatenation of expressions represented by the symbol $+$. We give a fully distributed detection algorithm that works on-line and does not modify the protocol specification.

[5] A reactive system interacts continuously with its environment. In this case valid input events may lead to an erroneous state in the protocol implementation. When the implementation reaches an erroneous state it may either continue to run but producing erroneous output or may crash and stop. An erroneous state was reached in the implementation because the values associated to its local variables and the contents of its communication channel are not correct according to constraints or requirements given in the protocol specification. Note that this behavior can only be identified if test sequences are as long as the length of the faulty path.

In the case of a centralized monitor P_Φ, each process P_i ($i = 1 \ldots n$) has to send a message to P_Φ so the property can be checked. Furthermore, if we want to use the information provided by the algorithm during normal execution (see Section 5) then the execution in process P_i has to be delayed. The algorithm proposed in this paper does not have the cost of sending extra messages since it is distributed and does not delay the execution of process P_i.

For an overview of the concepts related to this paper and some of the approaches proposed in the literature, the interested reader is referred to [16].

7 Conclusions

Verification and testing are two complementary techniques and should be used during the protocol development cycle.

In the context of protocol testing, most of the properties that characterize valid or invalid behaviors cannot be expressed in terms of stable properties. In fact, these properties are more properly expressed in terms of desirable or undesirable temporal evolutions of the communication protocol behavior. These properties are inherently unstable since there is no guarantee that they will remain either valid or invalid in a protocol computation.

In this paper we have presented a new algorithm to detect dynamic unstable properties that can be used in the testing of distributed processes (modules) of a communication protocol. This algorithm provides two types of information that can be used for tackling two problems during program execution: state build-up and exception handling.

The algorithm is based on the observation that given a protocol specification there are multiple valid computations (traces) each of which can be defined by a causal precedence order. Dynamic properties are specified by stating conditions (using local predicates) that should hold on all possible computations.

We have also presented a new theorem that can be used to check general properties for a specific type of communication protocol, namely 3-way handshake protocols.

From the point of view of testing communication protocols or concurrent reactive systems the algorithm presented in this paper is an improvement over the ones presented in [7, 19] and [18].

References

[1] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, ACM Press Frontier Series, chapter 4, pages 55–96. ACM Press and Addison-Wesley, second edition, 1993.

[2] K. Mani Chandy, L.M. Haas, and Jayadev Misra. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[3] 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.

[4] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174. Santa Cruz, CA, USA, 20–21 May 1991. Published as ACM SIGPLAN Notices, 26(12), December 1991.

[5] Edsger W. Dijkstra and Carel S. Scholten. Termination detection for diffusing computation. *Information Processing Letters*, 11:217–219, August 1980.

[6] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.

[7] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[8] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, CA, USA, 17–18 June 1993. Published as SIGPLAN Notices, 28(12), December 1993.

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

[10] C. L. Liu. *Elements of Discrete Mathematics*. McGraw-Hill Computer Science Series. McGraw-Hill, second edition, 1985.

[11] Antonio A.F. Loureiro, Samuel T. Chanson, and Son T. Vuong. A critical assessment of design for testability in communication protocols. In *13º Simpósio Brasileiro de Redes de Computadores*, pages 61–80, Belo Horizonte, MG, May 1195.

[12] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, CA, USA, 13–17 June 1988.

[13] D. Richardson and M. Thomas. The RELAY model of error detection and its application. In *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, Banff, Alberta, Canada, July 1988.

[14] John Rushby. Formal methods and the certification of critical systems. Technical Report CSL–93–7, SRI International, Computer Science Laboratory, Menlo Park, CA, USA, 1993.

[15] Beth A. Schroeder. On-line monitoring: A tutorial. *Computer*, 28(6):72–78, June 1995.

[16] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[17] M. Spezialetti and J. P. Kearns. A general approach to recognizing event occurrences in distributed computations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 300–307, San Jose, CA, USA, 13–17 June 1988.

[18] S. Venkatesan and Brahma Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, February 1995.

[19] Brian Waldecker and Vijay K. Garg. Detection of strong predicates in distributed programs. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 692–699, Dallas, Texas, USA, 2–5 December 1991.