UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 635

DERIVING TRACE CHECKERS FOR DISTRIBUTED SYSTEMS

Claude JARD
Omar DRISSI-KAITOUNI

Mars 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone: 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

Publication Interne n°347

Février 1987

18 Pages

DERIVING TRACE CHECKERS FOR DISTRIBUTED SYSTEMS

GENERATION DE VERIFICATEURS DE TRACES POUR LES SYSTEMES DISTRIBUES

Claude Jard and
Omar Drissi-Kaitouni

IRISA
Campus de Beaulieu
35042 RENNES CEDEX FRANCE
Tel 99362000

Abstract

What we call a "trace checker" is a module which observes the execution of a system under test and compares its behavior with the formal specification given for that system. This paper provides a survey on the design of trace checkers to detect violation of service properties, for distributed systems. We focus our presentation on the automated derivation of trace checkers from different formalisms, synthesizing several recent papers in the testing and simulation research area. We first begin by extended finite state models and their applications using the Estelle language. The main part is devoted to temporal logic specifications over finite computations and their translation into finite acceptors. We end the paper by introducing some open problems with distributed trace checking.

Résumé

Qu'appelle-t-on "vérificateur de trace" . C'est un module qui observe l'exécution d'un système sous test et compare le comportement observé avec la spécification formelle de ce système. Cet article fait le point sur la conception de vérificateurs de traces, chargés de détecter la violation des propriétés de service des systèmes répartis. Notre exposé est centré sur le problème de la génération automatique de vérificateurs de traces à partir de différents formalismes exprimant le service. Nous offrons ainssi une synthése de récents travaux de recherche dans les domaines du test et de la simulation. La partie principale de l'article est consacrée aux spécifications en logique temporelle sur des traces d'exécution finies et leur compilation en automates accepteurs. Nous concluons en introduisant quelques problèmes liés à la vérification distribuée de traces.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

1 Introduction

It is now well-known that the validation activity for distributed systems can be conducted by two complementary ways:

- verification, which analyzes the specifications by logical means. This may take the form of program "correctness" proofs, exhaustive simulation, symbolic execution, etc.
- testing, which explores a large number of the possible execution histories of the system and compares the observed behavior with the expected one given by the specifications. This may take the form of simulation studies or traditional testing procedures.

Verification methods have difficulties to handle real protocols. This increases interest in testing methods. Our synthesis on "trace checking" is a contribution in this context.

What we call a "trace checker" is a module which observes the execution of the system under test (which may be a system implementation or an artificial execution of some refined system specifications) and compares its behavior with the formal specifications given for that system. A similar notion was first introduced by the worker-observer principle in [Ayache 79] and in [Molva 85].

Figure 1 presents a global trace checking architecture to detect violations of service properties for a distributed system.

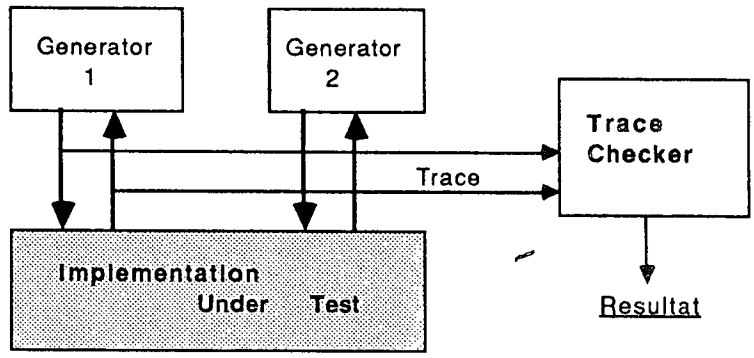


Figure 1. Global trace checking

In such an architecture, the implementation under test is stimulated by some test input sequences. These inputs, as well as the outputs produced by the implementation, are observed by the trace checker. The trace checker checks that the observed sequence of events is a possible sequence according to the given specification. This allows on-line testing. It is to be noted that an important requirement for validation is that observation must not disturb the system under test. Consequently in some cases, we cannot prevent a latency in error detection.

PAPIER RECUPERE ET RECYCLE

Since interactions at different service access points are considered, the observed trace is a global trace. For checking global properties, the relative order of interactions at different points must be determined. In a distributed system, this may be in general a difficult problem. However, there exist practical methods to obtain a total order of all observed interactions. At the end of this paper (section 5), we shall introduce some problems with distributed trace checking, when a global trace is not considered, as shown in figure 2.

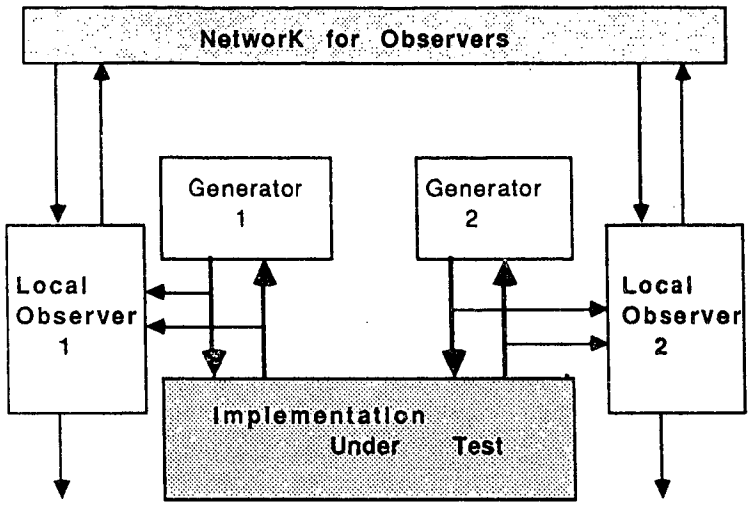


Figure 2. Distributed trace checking

This show how to derive trace checkers from various formalisms and discuss their practical use. We do not address the problem of the choice of a good language to express service properties of protocols or distributed algorithms (which still seems to be an open problem). Deriving trace chechers was also partly addressed in [Ural 84] and is complementary to the problem of automatically obtaining test sequences from formal specifications [Sarikaya 84,Castanet 86].

Our presentation is organized as follows:

In section 2, we describe models for trace checkers based upon state machines. Two different approaches are considered: use of finite acceptors and use of finite transducers. First of all, we have defined extended finite acceptors using the expressive power of an Estelle-like language. This experience was gained during the *Véda* Project [Jard 85a] and was presented last year in [Groz 86] in a simulation context. Finite transducers are then considered in order to reduce state explosion due to the non-determinism inherent to service specifications. This way to describe services is quite closed to the ISO philosophy using Estelle. Our experience was first reported in [Jard 83].

We take into account temporal logic specifications of services in sections 3 and 4. Deriving trace checkers was a difficult problem a few years ago. We give here a practical algorithm to do it,and we discuss the use of these "compiled" observers. Section 5 introduces the distributed checking problem.

2 Models based on finite state machines

Finite acceptors are obviously trace checkers. Each transition is labelled by a possible event of the observed traces. The automaton represents the set of all words (traces) spelled out by the set of all paths from the initial state to a terminal state. A trace is valid (i.e accepted) if it belongs to this set.

2.1 Non determinism in specification

A non-deterministic service specification is such that several different traces are valid for a given sequence of input interactions provided to the system under test. This is because the same inputs can produce different sets of outputs. A given prefix of a trace can be then accepted by several states of the automaton. All these states must be recorded and considered for the acceptance of the next events. A trace is not valid if no possible state accepts the current event. The number of states may be rather large and costly. Non-determinism is inherent to service specifications, since non ideal communication medium are generally considered between processes of the protocol under consideration. In practice, it is mostly due to the observation of "spontaneous" actions of the system for one of the following reasons:

- failures of system components which may occur any time,
- time-out mechanisms, which activate time-out transitions after a non-specified period of time,
- incompletely specified conditions, which may occur at certain times, such as "network congestion",
- concurrency, where the order in which two or more independent actions occur is not significant.

2.2 Experience with VEDA's observers

Our experience in using finite state observers was gained during the *Véda* project. *Véda* is a simulator of protocols described in Estelle [ISO 86] and was presented in [Jard 85a].

Last year was presented in [Groz 86] what we call an observer in *Véda*. Observers in *Véda* are basically state acceptors. They have their own variables, and use probes to access objects in the system under test. The observer is active between transitions of the system. When it is active, it can observe the state of all subsystems and all interactions that have taken place during the system transition just executed. At each step of observation, it computes its new state from the previous one and from the situation observed.
Practically, such observers are described in Estelle, with only a few extensions to use probes, and a few predefined types and procedures. Estelle being based on *Pascal*, we have the full power of a programming language for the verification process.
This method was developed for simulation studies. It is also applicable to trace checking of implementations. In that context, interactions have to be globally ordered. *Véda*'s experience

has shown (see [Jard 85b] for details about generated codes of Estelle processes) that such observers may be compiled into an executable code. For a real implementation, provided the availability of a global trace, care must be taken to:

- use a high-level decoder for service primitives which observes strings of bits and provides structured primitives (in Estelle terms for example). Specifications for such decoders can be found in [Ansart 82].
- link the generated code with a specific execution kernel to handle non-determinism and verification.

2.3 Active services

The OSI philosophy for describing protocol services using Estelle is to consider services as active components (which accept inputs and initiate outputs). These services are abstract machines which can be simulated for instance. Differently to the approach explained in the previous section, the underlying model is the class of finite transducers, which allows to distinguish between input and output service primitives.
State explosion due to the non-determinism of service specification can be reduced by this facility. In [Jard 83], we presented how to implement trace checkers from an Estelle description of services (see also [Bochmann 82] to have a general view on the use of Estelle during the validation activity). The execution kernel implements the following strategy: The analysis of the next interaction in the observed trace builds up a list of possible next states; For each of the states in the list mentioned above, the following action is executed:

- If the interaction is an input, then all possible input transitions are considered, each leading to a next state which is placed on the list of possible next states. Any output interaction generated by the transition is recorded in the next state. Then all internal transitions (spontaneous, and initiating no output) are considered from the new states found. This process is continued until no new states can be found.
- While an input interaction in the trace usually increases the number of possible states, output interactions usually reduce this number. When considering an output, and for each of possible states, all possible spontaneous transitions are considered to check whether they generate the observed output. Any different output initiated is kept as state information. After termination of that process, only those states are kept that have generated the observed output in the trace. All other states are discarded from the further analysis. An error is detected when the list of possible states is empty.

Several experiences have been performed using this approach: a distributed exclusion algorithm, the ISO transport and session protocols. These were validated by simulation as reported in [Jard 84]. Checking an implementation requires a decoder which sorts input and output interactions, and a specific kernel as presented above. This kind of trace checkers may be easily obtained by a compilation process using a version of the *Véda*'s compiler.

3 Temporal logic specifications

The logical context provides an interesting abstraction (independance with respect to implementation choices) and conciseness to describe protocol services. A trace checker is basically an observer; therefore the choice of a logic called "linear time" is naturally imposed. Moreover, with the observation period being finite, the checker must take into account of finite computations. In temporal logic, we consider services as defined by:

- a set of observable events: these are generally the service primitives exchanged between the application level and the protocol under consideration.
- a formula specifying an order relation between the occurrences of these events.

In the following, we will employ the definitions given in the recent survey of A. PNUELI about specification of reactive systems using a linear temporal logic [Pnueli 86].

3.1 Syntax

The set F of temporal logic formulas is built from:

- a set of observable events $E, E=\{e_1,\ldots,e_i,\ldots,e_n\}$
- the constants: *true* and *false*
- the classical boolean connectives: $\wedge$ and $\neg$
- some temporal operators: $\odot$ (next), $\otimes$ (previous), $*$ (until), and $\$$ (since)

The well-formed formulas are obtained through applying classical formation rules according to the fact that $\odot$ and $\otimes$ are unary operators and $*$ and $\$$ are binary operators.

Abbreviations: $\vee$, $\supset$, $\equiv$ are defined in the usual way.

Simplified temporal operators are defined by:

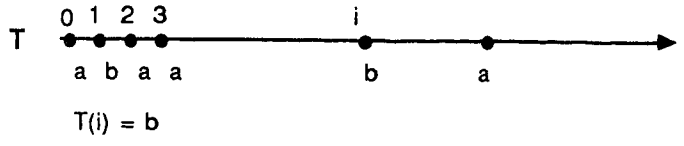
$\bigcirc f \equiv \neg\odot\neg f$ (weak next)	$\oplus f \equiv \neg\otimes\neg f$ (weak previous)
$\Diamond f \equiv true * f$ (eventually)	$\blacklozenge f \equiv true \$ f$ (sometimes in the past)
$\Box f \equiv \neg\Diamond\neg f$ (always)	$\blacksquare \equiv \neg\blacklozenge\neg f$ (always in the past)

In the following, when the set E is not precised for a formula f, this will be the set of atoms of f.

3.2 Semantics

The logical formulas are interpreted on the traces of execution. A trace is a countable set of occurrences of events of E. We can range over $\mathbb{N}$ the different points of the trace. Let $T(i)$ be the i^{th} point. $|T|$ is the lenght of finite traces.

Example: $E=\{$ a, b $\}$



Implicitly, there exists one and only one occurrence of an event at each point of the trace (no simultaneous events).
The truth value of a formula f, for a trace T is defined inductively by the notion of holding at position i $< |T|$ of the trace.(noted by i $\models$ f)

- $\forall e \in E, \quad i \models e \quad \Leftrightarrow T(i) = e$
- $\forall f_1, f_2 \in F, \quad i \models f_1 \wedge f_2 \quad \Leftrightarrow (i \models f_1)\ and\ (i \models f_2)$
- $\forall f \in F, \quad i \models \neg f \quad \Leftrightarrow not\,(i \models f)$
- $\forall f \in F, \quad i \models \odot f \quad \Leftrightarrow (i < |T|)\ and\ (i+1 \models f)$
- $\forall f_1, f_2 \in F, \quad i \models f_1 * f_2 \quad \Leftrightarrow [\exists j, i \leq j < |T|, (j \models f_2)]\ and\ [\forall k, i \leq k < j, (k \models f_1)]$
- $\forall f \in F, \quad i \models \otimes f \quad \Leftrightarrow (i > 0)\ and\ (i-1 \models f)$
- $\forall f_1, f_2 \in F, \quad i \models f_1 \$ f_2 \quad \Leftrightarrow [\exists j, 0 \leq j \leq i, (j \models f_2)]\ and\ [\forall k, j < k \leq i, (k \models f_1)]$

we say that a trace T *satisfies* a formula f if and only if 0 $\models$ f .

Derived interpretation of operators: $\Diamond$, $\Box$, $\blacklozenge$, $\blacksquare$

- $i \models \Diamond f \quad \Leftrightarrow \exists j, i \leq j < |T|, (j \models f).$
- $i \models \Box f \quad \Leftrightarrow \forall j, i \leq j < |T|, (j \models f).$
- $i \models \blacklozenge f \quad \Leftrightarrow \exists j, 0 \leq j \leq i, (j \models f).$
- $i \models \blacksquare f \quad \Leftrightarrow \forall j, 0 \leq j \leq i, (j \models f).$

nota: for infinite traces: $\odot f \Leftrightarrow \bigcirc f$, $\otimes f \Leftrightarrow \oplus f$
for finite traces: $\neg|T|-1 \models \odot f, and\ |T|-1| = \bigcirc f$
similarly for the past: $\neg 0 \models \otimes f and 0 \models \oplus f$
The logical theorems are the formulas f such that $\forall T \in E^*, T satisfies\ f$

Example :

$$\vdash \Box \Big[\bigvee_{e \in E} e \wedge \bigwedge_{\substack{e_1, e_2 \in E \\ e_1 \neq e_2}} \neg(e_1 \wedge e_2) \Big]$$

3.3 Examples

3.3.1 Mutual exclusion

Let n sites sharing a resource. The observable events at the service level are the exclusion requests to resource (de); the entry in critical section (er); and release (fe).

- All requests are served:

$$\forall i, 1 \leqq i \leqq n, \ \Box\ (de_i \ \supset\ \Diamond er_i)$$

- Only one request is served at a time (a site cannot request the resource again if it has not released it):

$$\forall i, 1 \leqq i \leqq n, \ \Box\ (de_i \ \supset\ \bigcirc(\neg de_i * fe_i))$$

- A site cannot release the resource if it does not possess it:

$$\forall i, 1 \leqq i \leqq n, \ \Box\ (fe_i \ \supset\ \bigcirc(\neg fe_i * er_i))$$

- The resource is released after a finite amount of time:

$$\forall i, 1 \leqq i \leqq n, \ \Box\ (er_i \ \supset\ \Diamond fe_i)$$

- Only one site at a time may have access to the critical section (one can't observe two accesses to resource without releasing the accessed one):

$$\forall i,j; 1 \leqq i,j \leqq n, \ \Box\ (er_i \ \supset\ \bigcirc \neg er_j * fe_i) \quad \text{(global property)}$$

- Initial conditions of observation (we can initially assume that all sites are idle) :

$$\forall 1 \leqq i \leqq n, [\neg(er_i \vee fe_i) * de_i] \wedge [\neg fe_i * er_i]$$

3.3.2 Transfer protocol (reliability and sequencing)

For a transfer protocol between two sites, a set of observable distincts events is

$$E = \{+\mathrm{m}_1, \ -\mathrm{m}_1, \ +\mathrm{m}_2, \ -\mathrm{m}_2, \ \ldots\ldots \}$$

$+m_i$ is the emission of the message m_i and $-m_i$ is its reception at an other site.

- No loss of messages:

$$\forall i \geqq 1, \ \ \Box\ (+m_i \ \supset\ \Diamond -m_i)$$

- Sequencing (for two transmissions, the reception of the second message must not take place before the first):

$$\forall i,j \geqq 1; \ j \neq i, \ \ \Box\ ((+m_i \wedge \Diamond -m_j) \ \supset\ ((\neg -m_j) * -m_i))$$

- No spontaneous generation:

$$\forall i \geqq 1, \ \ \Box\ (-m_i \ \supset\ \blacklozenge +m_i)$$

nota: $\Box\,(-m_i \supset \blacklozenge + m_i) \equiv \neg(-m_i \,\mathcal{S}\, + m_i)$

- No duplication (a reception of a given message can happen only once):

$$\forall i \geq 1,\ \Box\,(-m_i \supset \bigcirc \Box\, \neg(-m_i))$$

3.4 Some results from the theory

The theory of linear time logic over finite traces was linked to the automata theory several years ago. The main results were given in [Kamp 68] and presented to the protocol community in [Lichtenstein 85,Thomas 81,Pnueli 86]. We are concerned with the following proposition : Let L be a set of traces. The following two characterizations are equivalent:

i) L is definable by a temporal logic formula f.
ii) L is accepted by a counter-free automaton.

Proposition i) means that $\forall T \in L, T$ *satisfies* f. A counter-free automator [Mac Naughton 71] is a finite state automaton which can not perform any counting operations. Languages accepted by such automata (L) are such that :

$$\exists n \in \mathbb{N}, \forall u, v, w \in L, uv^{n+1}w \in L \Leftrightarrow uv^{n}w \in L$$

In particular, there exists a finite state machine $A(f;)$ for each formula f such that:

$$\forall T \in E^{*}, A(f)\ accepts\ T \Leftrightarrow T\ satisfies\ f$$

As explained in section 2, $A(f)$ can be executed as a trace checker.

4 From temporal logic to trace checkers

The problem is to automatically derive the automaton $A(f)$ from a formula f. Using only future temporal operators, Z. MANNA and P. WOLPER in [Manna 84] turn away a satisfiability algorithm in order to synthesize programs. Here, we present a similar algorithm of construction in which the problems of its termination and of determining the terminal states are clearly solved. This allowed us to program (in PROLOG) a compiler of formulas.

4.1 From temporal logic without past

The principle of operation is the following:
Each state is characterized by a formula which represents the remaining condition to be verified in the trace. Initially, the automata begins in the state F. To calculate the successors of F, and for each possible event e, we begin by "scrolling" F through decomposition to obtain a formula containing only atomic events and $\odot$-formulas. The condition on the current state is then

calculated for the event e by replacing it with *true* and the others with *false*, and realizing the boolean reductions. After elimination of the $\odot$-operators at the first level, the formula represents the next condition to be realized after having observed the event e. We check if this one is equivalent (according to the propositional calculus) to an already-seen state, otherwise a new state is created. The operation is performed again for all states, until termination. A transition leading to the state *false* means that the event is not accepted in this state (such transition is generally not represented). A state is terminal if after "scrolling" and replacement of the atoms with *false* and $\odot$-formulas with *true*, the formula is reduced to *true*.

Scrolling operation D : Deduced from the interpretation of the *until* operator, the rule of scrolling $f_1 * f_2$ is as follows:

$$f_1 * f_2 = f_2 \vee (f_1 \wedge \odot(f_1 * f_2))$$
$$(then \quad \Diamond f = f \vee \odot \Diamond f \ , and \quad \Box f = f \wedge \circ \ \Box f)$$

$\forall x \in E, \ \forall f, f_1, f_2 \in F$

- $D(x) = x$
- $D(\odot f) = \odot f, and \ D(\neg f) = \neg D(f)$
- $D(f_1 \wedge f_2) = D(f_1) \wedge D(f_2)$
- $D(f_1 * f_2) = \neg(\neg D(f_2) \wedge \neg(D(f_1) \wedge \odot(f_1 * f_2)))$

Example: $D(\Box(a \supset \Diamond b)) = (\neg a \vee (b \vee \odot \Diamond b)) \wedge \neg \odot \Diamond \neg(a \supset \Diamond b)$.

The operation of instantiation and reduction R

This consists, for a formula F and a given event e, in changing event e by *true* and the other events by *false* (only on event at a time). Then we perform the boolean reductions, to obtain a $\odot$-formula. The characteristic formula of the next state is obtained by elimination of $\odot$-operators at the first level.

$\forall x, y \in E, \quad \forall f, f_1, f_2 \in F$

- $R(\odot f, x) = f,$ (path to next state)
- $R(f_1 \wedge f_2, x) = R(R(f_1, x) \wedge R(f_2, x), x)$
- $R(\neg f, x) = R(\neg R(f, x), x)$
- $R(x, x) = true, and \quad R(x, y) = false$
- $R(true \wedge f, x) = f, and \ R(f \wedge true, x) = f$
- $R(false \wedge f, x) = false, and \ R(f \wedge false, x) = false$
- $R(\neg false, x) = true, and \ R(\neg true, x) = false$

Examples : $R(D(\Box\ (a \supset \Diamond b)), b) = \Box\ (a \supset \Diamond b)$.
$R(D(\Box\ (a \supset \Diamond b)), a) = \Diamond b \wedge \Box\ (a \supset \Diamond b)$.

Condition of acceptance: The condition of acceptance for a state is calculated in a similar way to operation R, replacing the rules for events and $\odot$-formulas by:

$$\forall x \in E,\ R^a(x) = false, and \quad \forall f \in F,\ R^a(\odot f) = false$$

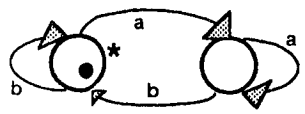
A state is terminal if the recursive application of R^a to the formula produces the constant *true*.
Examples : $R^a(D(\Box\ (a \supset \Diamond b))) = true$,
$R^a(D(\Diamond b \wedge \Box\ (a \supset \Diamond b))) = false$.

Propositional equivalence E:
$\forall f,\ h \in F, f$"*P-equivalent to* "h if and only if f $\Leftrightarrow$ h considering the temporal expressions of f and h as atoms of the propositional calculus. The computation of E uses a classical decision procedure for the propositional calculus. Two states of the automaton are equivalent if their formulas are P-equivalent.
The termination of the construction algorithm is one of the essential point of this method. That is to say that the set of states of the automaton is finished. The idea of the proof is to establish that $R(D(f),e)$ produces a formula composed of sub-formulas of f. The number of these sub-formulas being finished, this provides the result. Clearly, the automaton produced is not necessarily minimal knowing that two temporal formulas f and h may be equivalent without being detected as propositionally equivalent.

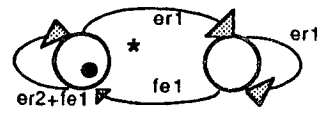
4.2 Past and future

Theory shows us that all temporal formulas composed of future and past operators may be expressed solely with future operators. Unfortunately, the translation may be extremely complicated and we do not yet know a practical algorithm to calculate this. Without doubt, what is most interesting is to know how to process past operators while constructing the automaton.
The construction algorithm becomes much more complicated. A past formula concerns an already observed trace sequence. For each state of the automaton, we thus calculate and record the truth value of past sub-formulas. These values are then used in the reduction R. The precise description and proof of this algorithm go beyong the needs of this paper. A more general view based on linear logic of the first order, is currently under development by R.GROZ to solve these problems more directly, as announced in [Groz 86].
Examples: A state noted with star is terminal ; an initial state is represented with a point.

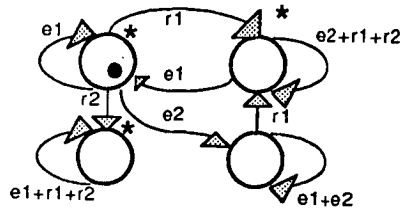
- $\Box\ [a \supset \Diamond b]$

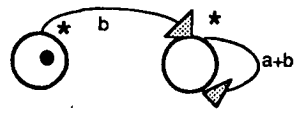


- $\Box\ [er_1 \supset (\neg er_2 * fe_1)]$



- $\Box\ [(e_1 \wedge \Diamond e_2) \supset (\neg r_2 * r_1)]$
- $\Box\ [a \supset \blacklozenge b]$





4.3 Practical use

Concerning the adequacy for the description of service properties, temporal logic seems better adapted to the description of global properties of execution sequences such as the exclusion property defined in section 3. The associated automaton may be very large since all the interleavings of possible events must be considered. On the contrary, machine based formalisms are interesting to characterize situations where there exist "tight" relationships between events such as sequencing or precedence properties. This is often the case of local properties (see section 2), for which a logic specification is presented as a conjunction of many formulas. J.SIFAKIS exposes a similar point of view in [Sifakis 86] and combines the behavioral and logic based approaches together as advocated in [Graf 86].
On the other hand, the expression power of temporal logic is very weak: we saw that it is inferior to that of finite state automata. However, when the description is possible, abstraction make it an interesting tool. The idea, therefore, consists in only using logic to generate the *skeleton* of the completed automaton which we will be able to further develop with the techniques mentioned in section 2.

5 Introduction to distributed trace checking

The purpose of this section is to introduce some problems raised by distributed testing. The distributed verification of traces generalizes the notion of checkers that we have presented, but puts important theoretical problems bound to the distributed observation.

5.1 Distributed testing and error detection power

The global approach presents a great interest from the point view of the verification process. However it is sometimes difficult to realize (the modification of the application code to derive a total order, etc.). Recently, an important work was accomplished in defining distributed testing architectures (see [Rayner 85]).
In that context, we must use many observers to verify the (local and global) properties of the system. Each observer sees only one part of the system. Verifying a global property with local observers needs the cooperation among them (i.e a testing protocol) (figure 3).

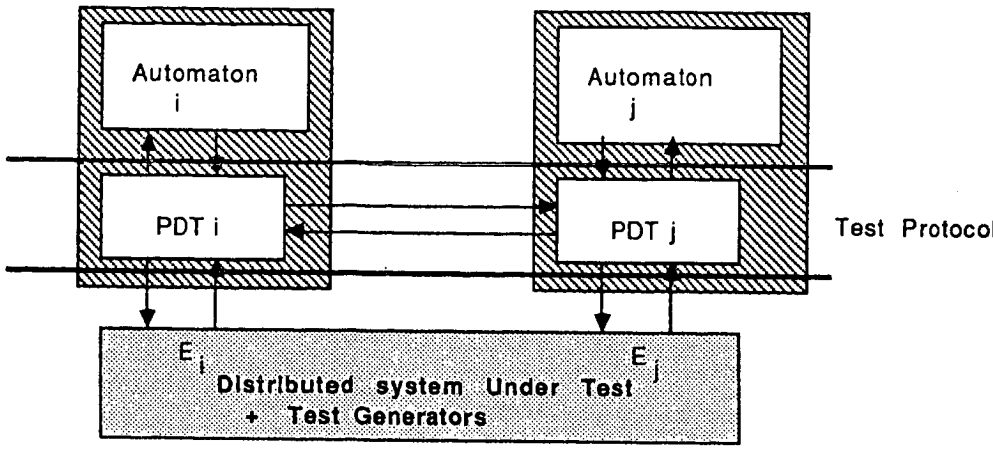


Figure 3. Distributed testing architecture

Let G be the global trace checker, introduced in section 2. $L(G)$ is the set of accepted global traces. The error detection power (*PDE*) [Dssouli 85] is defined as the set of traces which invalidate the service properties and belong to $L(G)$. In our case, we suppose that the global observer has a *PDE* reduced to empty (it is said maximum). The different observation points define a partition of E, E_1, ... ,E_n; E_i contains the set of events produced at the observation point i.

At this point, let L_i be the set of events external to that point:$\forall e \in E, \quad e \in L_i \Leftrightarrow e \notin E_i$.

These events are communicated by other observers. At each observation point i, we associate a local observer O_i The observer's checking automaton is obtained through the projection of G for the events $E_i \cup L_i$.

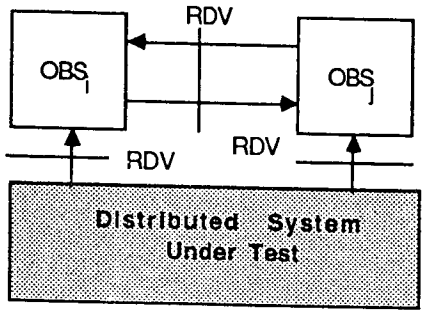
$$L(O_i) = L(G) / E_i \cup L_i$$

The error detection power of a local observer O_i (PDE_i) is deduced from PDE :

$$PDE_i = \{T_i \in L(O_i), T_i \in PDE / E_i \cup L_i\}$$

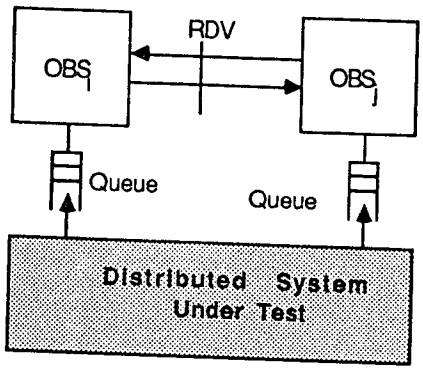
5.2 Some interesting classes of testing protocols

The coupled model



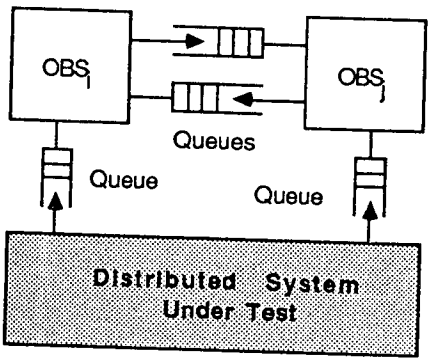
In this model [Merlin 83], every communication is made through rendez-vous. The local observer has always an exact image of the system, since all external or internal events are communicated. Unfortunately, this kind of active observation is inapplicable since it disturbs in general the distributed system under test. The systems for which this perturbation is not significant (i.e do not alter the execution sequences) are very synchronized .

Asynchronous observation



For this kind of passive observation, the observers receive the (internal) events of the system through queues ; where as, the external events are communicated through rendez-vous. The error detection power of the local observer (PDE_i) decreases in general. Systems which guarantee a maximal *PDE* are transactionel systems where the transactions are numbered [Ecault 84], or better, those systems which verify the "ping-pong" property defined in [Dssouli 86].

Distributed observation



In this case, every communication is made through queues. Its error detection power is weak:
$\forall$ i , PDE_i = $PDE/E_i \otimes PDE/\bar{L}_i$,($\otimes$: Shuffle operator).
The valid properties are therefore the formulas which are shuffle-invariant. This class of valid formulas is very reduced.

Knowledge of a total order relation

Some protocols maintain a total order relation between the observable events (for example when all messages are time-stamped). If this relation may be known to the observers, it is easy to build a testing protocol which resequences the observable events for each local observer. This allows for a maximum error detection power.

5.3 Towards easily testable protocols

The previous examples demonstrate the difficulty in controlling the quality of diagnosys with distributed testing. An important research is to be done to give some answers to the following questions:

- A testing architecture being given, what are the observable properties?.

- Service properties being given, what testing architecture may be proposed to observe these properties?.

The correctness of existing testing protocols is often due to implicite assumptions made upon the tested protocol itself. We could define observation points and specific actions in order to make easier subsequent testing. This approach was effective in the hardware design.

6 Conclusion

Different formalisms seem to be interesting and complementary for the service description activity. We focused our attention on finite state machine based models and on temporal logic. Deriving trace checkers from these formalisms is possible and hints for practical use of checkers on real implementations were given.
We think that our experience, which was performed essentially in a simulation context, is mature enough to be used for distributed applications in a more industrial context. In particular, it might be useful for protocol test centers to analyze (possibly off-line) complex traces produced during a test phase. Trace checkers can then decide their correctness.

References

[Ansart 82] J.P. ANSART, *Genepi/ A Protocol Independent System for Testing Protocol Implementation*, II IFIP WG6.1 Workshop, 1982.

[Ayache 79] J.M. AYACHE, P. AZEMA, M. DIAZ, *Observer: a Concept for On-line Detection for Control Errors in Concurrent Systems*, 9th Int. Symp FTC, Madison, June 1979.

[Bochmann 82] G.V. BOCHMANN, E.CERNY, M.GAGNÉ, C.JARD, A.LÉVEILLÉ, C.LACAILLE, M.MAKSUD, KS.RAGHUNATHAN, B.SARIKAYA, *Experience with Formal Specifications Using an Extented State Transition Model*, IEEE trans on Comm, Vol COM-30, Nu 12, dec 1982, pp 2506-2513.

[Castanet 86] R.CASTANET, R. SIJELMASSI, *Methods and Semi-automatic Tools for Preparing Distributed Testing*, VI IFIP WG6.1 Workshop, Gray Rocks, Montréal, June 1986, North-Holland, Gv.Bochmann and B.Sarikaya ed.

[Dssouli 85] R.DSSOULI, G.V.BOCHMANN, *Error Detection with Multiple observers*, V IFIP WG6.1 Workshop, Moissac, June 1985, France, North-Holland, M.Diaz ed.

[Dssouli 86] R.DSSOULI, *Etude des Methodes de Test pour les Implantations de Protocoles de Communication Basées sur les Spécifications Formelles*, PhD Thesis, Université de Montréal, Dec 1986.

[Ecault 84] C.ECAULT, *Une Experience dans la Spécification et la Validation des Systèmes Distribués*, Thèse de 3ieme cycle, Université Rennes I, France, Avril 85.

[Graf 86] S.GRAF, J.SIFAKIS, *A Logic for Description of non Deterministic Programs and their Properties*, Information and control 68, 1-3, 1986.

[Groz 86] R.GROZ, *Unrestricted Verification of Protocol Properties on a Simulation using an Observer Approach*, VI IFIP WG6.1 Workshop, Gray Rocks, Montréal, June 1986, North-Holland, Gv.Bochmann and B.Sarikaya ed.

[ISO 86] ISO/TC97/SC21/WG16-1 DP9074, *Estelle : a Formal Description Technique based on an Extented State Transition Model*, Sept 1986.

[Jard 83] C.JARD, G.V.BOCHMANN,*An Approach to Testing Specification*, The Journal of Systems and Software, 3, pp. 315-323, 1983.

[Jard 84] C.JARD, *Protocoles et Services : Test des Spécifications*, PhD thesis, Université Rennes I, France, May 1984.

[Jard 85a] C.JARD, R.GROZ, J.F.MONIN, *Véda: a Software Simulator for the Validation of Protocol Specifications*, COMMET'85, Budapest, Oct 1985, published by North-Holland in Computer Network Usage: Recent Experiences, L.Csaba, K.Tarnay, T.Szentivanyi ed.

[Jard 85b] C.JARD, J.F.MONIN, R.GROZ, *Experience in Implementing X250 in Véda*, V IFIP WG6.1 workshop, Moissac, June 1985, France, North-holland, M.Diaz ed.

[Kamp 68] HW.KAMP, *Tense Logic and the Theory of Linear Order*, PhD Thesis, 1968, UCLA.

[Lichtenschein 85] O.LICHTENSTEIN, A.PNUELI, L.ZUCH, *The Glory of the Past*, Logics of Programs, LNCS, 1985.

[Manna 84] Z.MANNA, P.WOLPER, *Synthesis of Communication Processes from Temporal Logic Specifications*, ACM Trans on Programming Languages and Systems, Vol 6, Nu 1, January 1984, pp. 68-98.

[McNaughton 71] R.MCNAUGHTON, S.PAPERT, *Counter Free Automata*, MIT Press, Cambridge, Mars 1971.

[Merlin 83] P.MERLIN, GV.BOCHMANN, *On the Construction of Submodule Specification and Communication Protocols*, ACM Trans on Programming Languages and Systems, Val 6, Nu 1, January 1983.

[Molva 85] R. MOLVA, M. DIAZ, J.M. AYACHE, *Observer: a Run-time Checking Tool for Local Area Network*, V IFIP WG6.1 workshop, Moissac, June 1985, France, North-Holland, M.Diaz ed.

[Pnueli 86] A.PNUELI, *Application of Temporal logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends*, LNCS 224, 1986, pp. 510-584.

[Rayner 85] D.RAYNER, *Towards Standardized OSI Conformance Tests*, V IFIP WG6.1 workshop, Moissac, June 1985, France, North-Holland, M.Diaz ed.

[Sarikaya 84] B.SARIKAYA, *Test Design for Computer Network Protocols*, PhD thesis, March 1984, School of Computer Science, Mc Gill, Montréal.

[Sifakis 86] J.SIFAKIS,*A response to Amir Pnueli's "Specification and development of reactive systems"*, IFIP Int. Congress, 1986, Dublin.

[Thomas 81] W.THOMAS,*A Conbinatorial Approach to the Theory of w-Automata*, Information and Control 48, 3, pp. 261-283, 1981.

[Ural 84] H.URAL, *An Approach to Life Cycle Testing of Communication Protocols*, PhD Thesis, 1984, Dept Computing Science, University of Ottawa.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique