Compiling temporal logic specifications into observers

Omar Drissi-Kaitouni, Claude Jard

INRIA

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° 881

COMPILING TEMPORAL LOGIC SPECIFICATIONS INTO OBSERVERS

Omar DRISSI-KAITOUNI
Claude JARD

JUILLET 1988

RR-0881

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° 421
18 Pages - Juillet 1988

Compiling Temporal Logic Specifications into Observers
Compiler des spécifications exprimées en logique temporelle vers des observateurs

Omar DRISSI-KAITOUNI and
Claude JARD

Abstract

An observer is an object which observes the interactions taking place between different system modules during operation. It may compare the observed interactions, also called trace, with the specification of the system under test.

Temporal logic is used to specify the properties of the system under test.

We present a new algorithm to translate such specifications into observers. It is based on a classical derivation method. Our approach is examplified by the specification of a reliable data transfer service and its derivation into a trace checker.

Résumé

Un observateur est un objet informatique qui observe les interactions qui se produisent entre les différents modules d'un système. Il a pour tâche de comparer la suite des interactions observées, appelée trace, à la spécification du système sous test.

Cette spécification se compose d'un ensemble de propriétés exprimées en logique temporelle linéaire.

Un nouvel algorithme pour traduire de telles spécifications en observateurs est présenté. Il est fondé sur une approche classique de dérivation de termes. L'utilisation de l'algorithme est illustrée par la spécification d'un service de transfert de données fiable, et la donnée du vérificateur de traces correspondant.

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

We know that compiling temporal logic formulae in automata is theoretically feasible. So, we propose a new and efficient algorithm to do that. It is based on a derivation method, successfully applied by Brzozowski in 1964 to translate general regular expressions into automata.

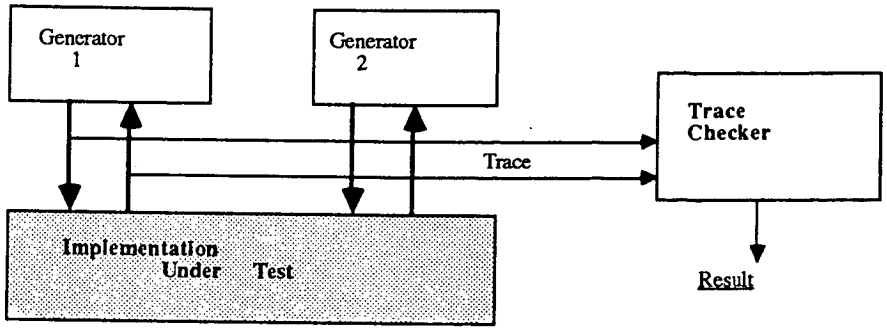
We think that this algorithm can be used to build observers for testing distributed systems. The concept of observer was introduced several years ago (see for instance [Ayache 79, Molva 85]) in the area of distributed system testing. An observer (also called "trace checker" in [Jard 83]) is a module 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. This concept allows for a clear separation of the test sequence selection problem [Sarikaya 84, Castanet 86], and the detection of any errors made by the system in respect to the specification. A survey on trace checking is available in [Drissi 87]. It is implicitely assumed that an observer cannot influence the system behavior in any way.

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

Since interactions at different access points are considered, the observed trace is a global trace. The relative order of interactions at different points must then be determined for checking global properties. 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. We do not discuss in this paper about ordering concurrent interactions, and how does it affect the validity of properties.

Our presentation is organized as follows. In section 2, we present our temporal logic specification language and its application to the specification of a reliable data transfer service. Deriving executable code from temporal formulae was a difficult problem a few years ago, although theory was well established. Section 3 presents and proves a practical algorithm to do it. This algorithm was implemented as a software package and is used currently to aid the specification phase. It produced the automaton associated to our service example.

Figure 1: Global trace checking architecture



2 Temporal logic specification

The logical context provides an interesting abstraction (implementation choices independence) and conciseness to describe properties of concurrent programs. Temporal logic allows the specification of the temporal ordering of actions (see [Pnueli 86] for a complete state of art).

Since we are interesting in deriving observers for implementations, we consider the linear time version of the temporal logic. Specifications are event-oriented specifications and interpreted only over finite computations.

Using this logic, a property is defined with :

- a set of observable events (actions) : these are generally communication events (e.g. primitives exchanged between the application level and the protocol under consideration).
- a formula specifying an order relation between the occurrences of these events.

Let us precise the definition of the logic.

2.1 Syntax

The set $\mathcal{F}$ of temporal logic formulae is built of :

- a set of observable events $E = \{e_1, ..., e_i, ..., e_n\}$ (atomic formulae).

- the constants : $\top$ (true) and $\bot$ (false),
- the classical boolean connectives : $\wedge$ (and) and $\neg$ (not),
- some temporal operators : $\odot$ (next) and $\mathcal{U}$ (strong until).

The well-formed formulae are obtained through applying the classical formation rules (considering $\odot$ as a unary operator and $\mathcal{U}$ binary).

We shall use abbreviations :

- $\vee$ (or), $\supset$ (implies) and $\equiv$ are defined in the usual way,
- $\forall \varphi \in \mathcal{F}, \bigcirc \varphi \equiv \neg \odot \neg \varphi$ (weak next), $\Diamond \varphi \equiv \top \mathcal{U} \varphi$ (eventually) and $\Box \varphi \equiv \neg \Diamond \neg \varphi$ (always).

2.2 Semantics

The logical formulae are interpreted over the finite computations (global traces) of the system under test. We can range over naturals the different actions of a computation C. Let C_i be the part of C starting from the i^{th} point, and $C(i)$ the event occurring at this point. $\mid C \mid$ denotes the length of the computation. No simultaneous events are considered in our model: there is exactly one occurrence of an event at each point of the computation.

The validity of a formula φ, for a computation C, is defined inductively from a position $i <\mid C \mid$ of the computation. This is denoted by $C_i \models \varphi$.

$C_i \models \top$		*always*
$C_i \models \bot$		*never*
$C_i \models \alpha$	*iff*	$C_i = \alpha \, (\forall \alpha \in E)$
$C_i \models \neg \varphi$	*iff*	$\neg (C(i) \models \varphi)$
$C_i \models \varphi \wedge \varphi'$	*iff*	$C_i \models \varphi \wedge C_i \models \varphi'$
$C_i \models \odot \varphi$	*iff*	$i+1 <\mid C \mid \wedge \, C_{i+1} \models \varphi$
$C_i \models \varphi \mathcal{U} \varphi'$	*iff*	$\exists j, i \leq j <\mid C \mid, C_j \models \varphi' \wedge \forall k, i \leq k < j, C_k \models \varphi$

We say that a computation C satisfies a formula φ if and only if φ holds at position 0 of the computation C. Since we are concerned with the relations between automata and logic formulae, we force the empty

computation (written λ) to be included in the models of our logic. This will allow us to express boolean combinations of temporal formulae in terms of automata compositions. This is an original aspect of our work.

Satisfaction of a formula φ by a computation C is then defined with :

$$C \models \varphi \quad \Leftrightarrow \quad \begin{array}{ll} C_0 \models \varphi & if\ C \neq \lambda \\ \delta(\varphi) & if\ C = \lambda \end{array}$$

where $\delta(\varphi)$ (true iff the empty trace satisfies φ) is recursively defined as :

$$\begin{array}{ll} \delta(\top) \equiv \top & \delta(\bot) \equiv \bot \\ \delta(\alpha) \equiv \bot\ (\forall \alpha \in E) & \\ \delta(\neg\varphi) \equiv \neg\delta(\varphi) & \delta(\varphi \wedge \varphi') \equiv \delta(\varphi) \wedge \delta(\varphi') \\ \delta(\odot\varphi) \equiv \bot & \delta(\varphi \mathcal{U} \varphi') \equiv \delta(\varphi') \end{array}$$

As usual, two logic formulae are said equivalent if they are satisfied by the same set of computations. A formula which is satisfied by all the possible computations E^*(including the empty word) is a tautology. For instance, we have always exactly one event at each time :

$$\models \Box \left[\xi \wedge \neg \bigwedge_{\alpha \neq \beta \in E} (\alpha \wedge \beta) \right] \vee \Box \neg \xi$$

where

$$\xi \equiv \bigvee_{\alpha \in E} \alpha$$

ξ is true only for a non-empty computation (and thus $\delta(\xi)$ is always false).

Function δ preserves the usual axioms of the linear temporal logic. Note that for the finite case $\bigcirc\varphi \equiv \odot\varphi$ is not a tautology (this is false for unit length traces).

2.3 Formal specification of the example

2.3.1 General description

We consider the service provided by a data transfer protocol. Such protocols, as described for instance in [Stenning 76] or [ISO 86], secure an unidirectional flow of data using a positive handshake on each transfer. Flow control is provided by a window technique for acknowledgements assuming that every transmitted message is numbered modulo k. The communication environment may lose, duplicate or reorder messages.

A previous attempt to specify the properties of the example can be found in [Richier 87] using the Xesar branching time specification language.

This section provides a linear time specification which is intended to be almost complete as the associated automaton shows. We do not impose any kind of synchronization to start trace checking.

2.3.2 Service specification

The service specification (also called external specification) describes the behavior of the whole system as expected by the users of the protocol. At this level, the system is considered as a black box with an input port in_i and an output port out_i where the subscript i represents the associated sequence number. Observation of these events can be easily implemented by the numbering function of the transmitter, and by the acceptance condition of the receiver, for in_i and out_i respectively.

The (finite) set of observable events is then defined by :

$$E = \{in_i, out_i\}_{i \in [0..k-1]}$$

The adding operator + is interpreted modulo k.

1. *Initial specification* :

   This part of the specification is arbitrary and depends only on the instant of the beginning of the observation period. We do not impose any constraint on that.

2. *Input sequence specification* :

   (P_1) The input numbering is strictly increasing modulo k, e.g in_{i+1} is always preceded by in_i.

$$\forall i \in [0..k-1],\ \Box\left[(in_i \wedge \Diamond in_{i+1}) \supset \bigcirc\left(\neg(\bigvee_{j\in[0..k-1]} in_j)\mathcal{U} in_{i+1}\right)\right]$$

   (P_2) The input numbering is unambiguous, e.g in_i is possible if and only if the previous in_i (modulo k) has been transmitted and delivered.

$$\forall i \in [0..k-1],\ \Box\left[(in_i \wedge \bigcirc\Diamond in_i) \supset \bigcirc\neg(\neg out_i \mathcal{U} in_i)\right]$$

3. *Output sequence specification* :(same as input changing in_i in out_i)

4. *Specification of the transmission* :

   (P_5) Every transmitted message is delivered in a finite delay.

$$\forall i \in [0..k-1],\ \Box\left[in_i \supset \Diamond out_i\right]$$

   (P_6) Since we do not impose any constraint on the beginning of the observation, some requirement is needed to synchronize the output specification on the input one. The following property expresses that the two sequences $in_i..out_i..in_{i+1}..out_{i+1}$ and $in_i..in_{i+1}..out_i..out_{i+1}$ are the only valid orders.
   $\forall i \in [0..k-1]$,

$$\Box\left[(in_i \wedge \Diamond out_{i+1}) \supset \left[(\neg in_{i+1} \mathcal{U} out_{i+1}) \equiv (\neg out_i \mathcal{U} out_{i+1})\right]\right]$$

We claim that the conjunction of these ($6k$ terms) temporal logic properties provides a complete specification of the protocol service (the associated automaton is shown in section 3.5).

3 From temporal logic to finite state machines

3.1 Theory

The theory of linear time logic was linked to the automata theory twenty years ago [Kamp 68]. We are concerned with the following proposition.

Proposition 3.1.1 *Let $\mathcal{L}$ be a set of finite computations. The following two characterizations are equivalent :*
$\mathcal{L}$ is definable by a temporal logic formula φ ($\mathcal{L} = \{C \mid C \models \varphi\}$),
$\mathcal{L}$ is accepted by a counter-free automaton.

Let us present our notation :

a deterministic finite state machine is a tuple (Q, A, ρ, q_0, F) where Q is a finite set of states, A is a finite set of elementary actions, ρ is the transition function (from $Q \times A$ to $Q \cup \{\emptyset\}$), F is a subset of states, called terminal states, and q_0 is the initial state.

In this definition, ρ describes the effect of the execution of an elementary action a. For a state $q \in Q$, $\rho(q,a) \neq \emptyset$ indicates that performing action a, when the automaton is in state q, leads it in state $q' = \rho(q,a)$.

We extent the function ρ in the usual way on sequences of actions :

$$\begin{array}{lll} \forall q \in Q,\ \rho(q,\lambda) & = & q \\ \forall \sigma \in A^*, \forall a \in A,\ \rho(q,\sigma a) & = & \rho(\rho(q,\sigma),a) \end{array}$$

The language accepted by a deterministic finite state machine is the language $\mathcal{L}$ defined as :

$$\mathcal{L} = \{l \in A^* \mid \rho(q_0, l) \in F\}$$

Since counter-free automata [McNaughton 71] are a subclass of finite state machines, the theory allows to consider the finite state machine $\mathcal{A}_\varphi$ associated to a formula φ defined by :

$$\mathcal{L}(\mathcal{A}_\varphi) = \{C \in E^* \mid C \models \varphi\}$$

Proposition 3.1.1 was first applied in [Manna 84] to synthetize synchronization skeletons of protocols from their temporal logic specification.

Proposition 3.1.2 *A boolean combination of temporal formulae may be characterized by an automata composition:*

$$\forall \varphi, \varphi' \in \mathcal{F},\ \mathcal{A}_{\neg\varphi} = C(\mathcal{A}_\varphi),\ \mathcal{A}_{\varphi\wedge\varphi'} = \mathcal{A}_\varphi \cap \mathcal{A}_{\varphi'}$$

("C" denotes the complementation function)

Proof proceeds directly from the propagation of the boolean operators by the satisfaction relation at point 0 for non-empty computations and by the δ-function for the empty one.

3.2 Derivatives of temporal formulae

The term "derivative" was first introduced in [Brzozowski 64] in order to derive deterministic automata from general regular expressions. We adopt a similar presentation for temporal formulae.

Definition 3.2.1 *Given a formula φ and a finite sequence s, the derivative of φ with respect to s is a formula $D_s\varphi$ such that :*

$$\forall t \in E^*,\ t \models D_s\varphi \Leftrightarrow st \models \varphi$$

This defines a class of formulae which are all equivalent. $D_s\varphi$ represents any formula of that class. Derivation provides for calculating the satisfaction relation, using proposition 3.2.1.

Proposition 3.2.1 *Satisfaction may be characterized by the emptyness acceptation of a derivative.*

$$\forall s \in E^*, \forall \varphi \in \mathcal{F},\ s \models \varphi \Leftrightarrow \delta(D_s\varphi)$$

This proposition follows the definition 3.2.1.

Derivation may be calculated using the following propositions 3.2.2 and 3.2.3.

Proposition 3.2.2 *If φ is a temporal formula, the derivative of φ with respect to a sequence α of unit length can be found recursively as follows :*

$$D_\alpha \top \equiv \top \qquad D_\alpha \bot \equiv \bot$$
$$D_\alpha \alpha \equiv \top \qquad D_\alpha \beta \equiv \bot \ (\forall \beta \in E,\ \alpha \neq \beta)$$
$$D_\alpha \neg\varphi \equiv \neg D_\alpha \varphi \qquad D_\alpha(\varphi \wedge \varphi') \equiv D_\alpha\varphi \wedge D_\alpha\varphi'$$
$$D_\alpha \odot \varphi \equiv \varphi \wedge \xi$$
$$D_\alpha(\varphi \mathcal{U} \varphi') \equiv D_\alpha\varphi' \vee (D_\alpha\varphi \wedge \xi \wedge \varphi \mathcal{U} \varphi')$$

The proof is obvious for the constants, atomic formulae and boolean formulae using the properties of the satisfaction relation.

The $\mathcal{U}$-case is deduced from the $\odot$-case since we can establish the equivalence :

$$\varphi \mathcal{U} \varphi' \equiv \varphi' \vee (\varphi \wedge \bigodot \varphi \mathcal{U} \varphi')$$

The $\odot$-case requires more attention and is proved by the following equivalences, given a sequence t :

$$\begin{aligned} t \models D_\alpha \odot \varphi &\Leftrightarrow \alpha t \models \odot \varphi \\ &\Leftrightarrow t \models \varphi \wedge (t \neq \lambda) \\ &\Leftrightarrow t \models \varphi \wedge t \models \xi \\ &\Leftrightarrow t \models \varphi \wedge \xi \end{aligned}$$

Let us note that $D_\alpha \xi \equiv \top$.

Proposition 3.2.3 *The derivative of a formula φ with respect to a finite sequence s can be found recursively as follows :*

$$\begin{aligned} \forall \alpha \in E \quad D_{s\alpha}\varphi &\equiv D_\alpha D_s \varphi \\ D_\lambda \varphi &\equiv \varphi \end{aligned}$$

The proof follows the definition 3.2.1.

Since D_s and δ can be calculated for any formula φ, we are able to decide if a given computation s satisfies a given formula φ. This is established by proposition 3.2.1.

We shali point out in the next paragraph that we can calculate a priori all the derivatives of a formula φ.

3.3 Construction of $\mathcal{A}_\varphi$

Definition 3.3.1 *Two temporal formulae φ and φ' are said boolean-equivalent if they are equivalent according to the boolean calculus considering pure temporal sub-formulae (rooted by a temporal operator) as atoms.*

Propositional equivalence can be easily recognized by a classical decision procedure (e.g. in translating $\neg(\varphi \equiv \varphi')$ into disjonctive normal form and verifying contradiction in each term). It does not imply equivalence, since two formulae may be equivalent and not recognized as boolean-equivalent.

Proposition 3.3.1 *Every formula φ has only a finite number of non-boolean-equivalent derivatives. All the distinct derivatives can be calculated considering sequences of increasing length.*

This is proved by induction on the length of the formula, in the annex of the paper.

Proposition 3.3.2 *From all the previous propositions, we can calculate the automaton*
$\mathcal{A}_\varphi = (Q, A, \rho, q_0, F)$ *:*

- *The set of actions A is the set of observable events E,*
- *The set of states Q is the finite set of the non-boolean-equivalent derivatives of φ,*

- *The transition function is determined by the existence of a derivative*
  $\forall \psi, \psi' \in Q, \forall \alpha \in E,$
  $\psi' = \rho(\psi, \alpha) \Leftrightarrow \exists s \in E^* \ \psi \equiv D_s\varphi \wedge \psi' \equiv D_\alpha \psi,$
- *The initial state q_0 is φ, and a state ψ is a terminal state iff $\delta(\psi)$.*

Note 1: Enforcing the test of equivalence using proposition 3.3.3 reduces the number of distinct derivatives.

Note 2: Since we can decide if $\mathcal{L}(\mathcal{A}_\varphi)$ equals to $\mathcal{L}(\mathcal{A}_\top)$, the automaton construction provides a decision procedure.

Note 3: An algorithmic description is given by figure 3.

Proposition 3.3.3 *There is equivalence between φ and $\varphi \vee \neg\xi$ if the empty word satisfies φ.*

$$\forall \varphi \in \mathcal{F}, \ \delta(\varphi) \Rightarrow \varphi \equiv \varphi \vee \neg\xi$$

Proof :

- $\delta(\varphi \vee \neg\xi) \Leftrightarrow \delta(\varphi)$
- $\forall s \in E^+ \ s \models \varphi \vee \neg\xi \Leftrightarrow s \models \varphi$

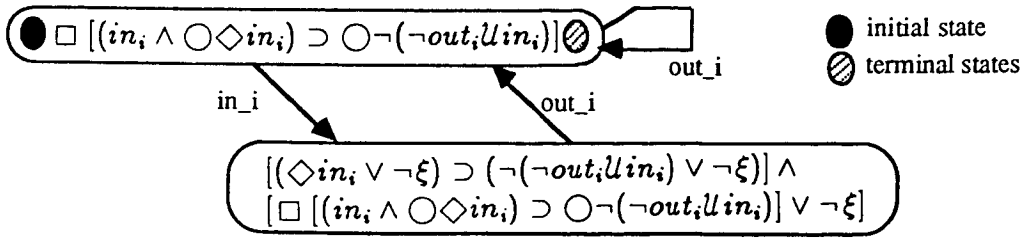
3.4 Example of derivatives

In order to illustrate the derivation process, we give here the automaton and derivatives of a sub-formula of our service specification.

Let us consider the following property (general form of P_2).

$$\begin{array}{lcl}
E & = & \{in_i, out_i\} \\
\varphi & \equiv & \Box\,[(in_i \wedge \bigcirc\Diamond in_i) \supset \bigcirc\neg(\neg out_i \mathcal{U} in_i)] \\
 & & \\
\delta(\varphi) & \equiv & \top \\
D_{in_i}\varphi & \equiv & [(\Diamond in_i \vee \neg\xi) \supset (\neg(\neg out_i \mathcal{U} in_i) \vee \neg\xi)] \wedge \\
 & & [\Box\,[(in_i \wedge \bigcirc\Diamond in_i) \supset \bigcirc\neg(\neg out_i \mathcal{U} in_i)] \vee \neg\xi] \\
\delta(D_{in_i}\varphi) & \equiv & \top \\
D_{out_i}\varphi & \equiv & \Box\,[(in_i \wedge \bigcirc\Diamond in_i) \supset \bigcirc\neg(\neg out_i \mathcal{U} in_i)] \vee \neg\xi \\
 & \equiv & \varphi \ (\textit{detected equivalent}) \\
D_{in_i in_i}\varphi & \equiv & \bot \\
D_{in_i out_i}\varphi & \equiv & \varphi \ (\textit{detected equivalent})
\end{array}$$

Figure 2: The P_2 automaton



The automaton is shown in figure 2.

3.5 Generating observers

The algorithm above has been implemented as a software package written in Pascal. We give here some hints which allowed us to reduce the time and space complexities.

Temporal logic formulae (in which or- and and-formulae are considered as n-ary) are represented by trees. Derivatives of some pure temporal formulae are kept during the computation to avoid re-derivation of previous terms, since the derivation rules can produce trees having common parts.

A lot of time is spent to test boolean-equivalence. In order to improve this procedure, we generate for each formula, the associated disjonctive normal form. Testing equivalence between normal forms is then straightforward.

The final program is 2500 lines length. It compiled the specification for k equals 2 in 15 seconds, and produces a 24 states automaton. The minimal equivalent automaton is shown in figure 4.

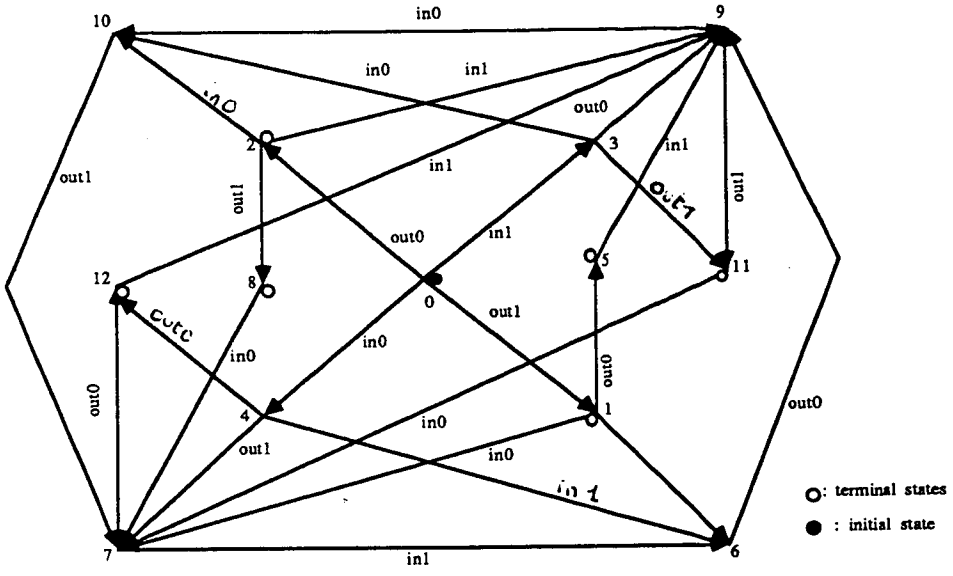
Let us comment the example. An interesting feature of the method was to consider that the observation could start at any time of the execution under test. This simplifies (a bit) the temporal specification although it would be difficult to synthetize directly the automaton.

The complementarity between the behavioral and logic based approaches to the specification was already advocated in [Graf 86]. Our experience

Figure 3: Text of the algorithm

```
type evt_type = scalar_type;
    state = record
            num : integer; — unique id of a state —
            fml : temporal_formula; — associated formula in a coded form —
            term : boolean; — true if terminal state —
        end;
    edge  = record
            source,destination : integer;
            evt : evt_type; — label of the edge —
        end;
var Edges : set of edge init ∅;
    States : set of state init {(0, φ, δ(φ))};
    number : integer init 1;
function exists_boolean_equivalent (f: temporal_formula; var q : state ) : boolean ;
    — 'same' refers the same syntactic structure —
    — 'boolean_decision' is a boolean decision procedure —
    function boolean_equivalent (f,f':temporal_formula): boolean ;
        begin
            if same(f,f') then boolean_equivalent := true
            else if δ(f) and same(f',f ∨¬ξ) then boolean_equivalent := true
            else boolean_equivalent := boolean_decision(f ≡ f')
        end
    begin
        exists_boolean_equivalent := false;
        all x in States do
            if boolean_equivalent(f,x.fml) then begin
                exists_boolean_equivalent := true ;q := x end
    end
procedure succs (d : state );
    var y : temporal_formula; q : state; a : edge;
    begin
        all e in E do begin
            y := De(d.fml);
            if not exists_boolean_equivalent(y,q) then begin
                q.num := number; q.fml := y; q.term := δ(y);
                States := States ∪ { q } ;
                number := number +1; succs(q) end;
            a.source := d.num; a.destination := q.num; a.evt := e;
            Edges := Edges ∪ { a } end
    end
main
    succs((0, φ, δ(φ)))
```

Figure 4: The data transfer checker



enforces that point of view : temporal logic seems better suited to the description of global properties. The associated automaton may be very large since all the interleavings of possible events must be considered. In contrast, 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 for local properties.

Executing the automaton as a trace checker is straightforward. This could be part of an integrated system for observation as presented in [Groz 86].

4 Conclusion

We tried to explore a new approach to design observers. Observers are derived from the temporal logic specification of some system properties.

The translation has been implemented and works with reasonable performance.

Obviously, finite non-counting automata are a rather restrictive class of observers. We think however that they could provide interesting skeletons for real observers. It will not be difficult (from the point of view of the observer generation) to extent the input logic language. Quantifiers and predicates on parameters associated to the observable events may be introduced.

Such compiled observers could be useful for protocol test centers to analyze (possibly off-line) complex traces produced during a test phase. The knowledge of the formula associated to the current state of the automaton may provide a kind of explanation when an error is detected.

Mixing logical and behavioral techniques for specification is a challenge. We contributed to that direction for the finite case. Our experience is that programming the translation between logic and automata provides a significant aid to describe service properties of protocols. We can then compare two different points of view of the same specification.

5 References

[Ayache 79] JM. 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.

[Brzozowski 64] JA. Brzozowski, *Derivatives of Regular Expressions*, JACM, Vol. 11, Nu. 4 (October 1964), pp. 481-494.

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

[**Drissi 87**] O. Drissi-Kaitouni, C. Jard, *Deriving Trace Checkers for Distributed Systems*, research report INRIA nu. 635, March 1987, 17 p.

[**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, Montreal, June 1986, North-Holland, Gv.Bochmann and B.Sarikaya ed.

[**ISO 86**] ISO/TC97/SC21, *Guidelines for the Application of Formal Description Techniques to OSI*, September 1986.

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

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

[**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.

[**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.

[**Richier 87**] JL. Richier, C. Rodrigez, J. Sifakis, J. Voiron, *Verification in Xesar of the Sliding-window Protocol*, VII IFIP WG6.1 workshop, Zurich, May 1987, Switzerland, H¿ Rudin, CH. West ed.

[**Sarikaya 84**] B.Sarikaya, *Test Design for Computer Network Protocols*, PhD thesis, March 1984, School of Computer Science, McGill, Montreal.

[**Stenning 76**] VN. Stenning, *A Data Transfer Protocol*, Computer Networks, 1(1976), pp. 99-110.

6 Annex

Proof of proposition 3.3.1

First part of the proposition :
By induction on the number of basic operators of the formula ψ, noted N.
Let Δ_φ be the set of distinct derivatives ($\Delta_\varphi = \{D_s\varphi \mid s \in E^*\}$).

1) $N = 0$
   ψ is $\bot, \top$ or $e \in E$
   $\Delta_\bot = \{\bot\}$, $\Delta_\top = \{\top\}$, $\Delta_e = \{\bot, \top, e\}$

2) let us suppose that $\forall \varphi \in \mathcal{F}$ with numbers of operators less than N, Δ_φ is finite.

case 1 $\psi = \neg\varphi$
Δ_ψ is finite since $\forall f,\ f \in \Delta_\psi \equiv f \in \Delta_\varphi$

case 2 $\psi = \varphi \wedge \varphi'$

$$\forall s \in E^*,\ D_s\psi = D_s\varphi \wedge D_s\varphi'$$

Thus

$$\Delta_\psi = \{f \wedge g \mid f \in \Delta_\varphi,\ g \in \Delta_{\varphi'}\}$$

Since Δ_φ and $\Delta_{\varphi'}$ are bounded, Δ_ψ is finite.

case 3 $\psi = \varphi \mathcal{U} \varphi'$

$\forall e_0, e_1, ..., e_n \in E$

$$
\begin{aligned}
D_{e_0 e_1 e_2 .. e_n}\psi \;=\; & D_{e_0 e_1 .. e_n}\varphi' \vee (D_{e_0 e_1 .. e_n}\varphi \wedge \\
& (D_{e_1 .. e_n}\varphi' \vee (D_{e1 .. e_n}\varphi \wedge \\
& (D_{e_2 .. e_n}\varphi' \vee (D_{e2 .. e_n}\varphi \wedge \\
& .... \\
& (D_{e_{n-1} e_n}\varphi' \vee (D_{e_{n-1} e_n}\varphi \wedge \\
& (D_{e_n}\varphi' \vee (D_{e_n}\varphi \wedge \xi) \quad ...)))
\end{aligned}
$$

$D_s\psi$ has $2n$ terms of the form $D_t\varphi$ and $D_t\varphi'$.
Since these terms have finite non-boolean equivalent forms, boolean reduction is possible. Reduction deletes redondant terms.
$D_s\psi$ is then boolean equivalent to a function with a finite number of variables. Thus Δ_ψ is finite.

Second part of the proposition :
Let us consider the sequences s of increasing length.
Suppose that :$\forall \varphi, \varphi' \in \mathcal{F}$,

$$\exists s, t \in E^*,\ |s| = |t| \ \wedge\ D_s\varphi \textit{ boolean equivalent to } D_t\varphi'$$

We can conclude that

$$\forall a \in E^*, D_{sa}\varphi \textit{ boolean equivalent to } D_{ta}\varphi'$$

This proves that when an equivalence is detected, it is not necessary to derive with respect to a longer sequence.

□

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

ISSN 0249-6399