How to Observe Interoperability at the Service Level of Protocols¹

Claude JARD

IRISA/CNRS, Campus de Beaulieu, F-35042 RENNES FRANCE. jard@irisa.fr

Interoperability of protocol implementations may not be guaranteed by their development process, even including a phase of conformance testing. This is why conformance at the service level must be tested. We focus on the problem of diagnostic: considering an implementation of several protocol entities, and a global property specifying the intended behavior of the program, the testers have to decide whether the protocol entities conform or not with the expected behavior, as defined by the protocol service. We show how general methods like time-stamps can be used in this context. We propose general algorithms to detect global properties. More generally, we advocate in favor of bringing methods designed in the area of distributed debugging to the problem of protocol testing.

1 Introduction

1.1 Why observing interoperability?

There are at least three good reasons why interoperability of protocol implementations may not be guaranteed by the (even formal) development process:

- protocol standards remain ambiguous, despite the formalization effort: they usually allow many options and choices which may sometimes be contradictory. Implementations based on different options may not inter-operate.
- the verification effort is not complete: only some particular aspects can be verified. The result depends on different assumptions made on the protocol environment (at the user level, as well as at the communication level), which may be not fulfilled at runtime.
- conformance test suites are not complete: they do not guarantee conformance to protocol standards. Cost considerations limit the size of any usable test suite. Complete coverage cannot be achieved in practice, so non conformant implementations can pass the test.

This is why conformance at the service (user) level must be tested. The reader may consult [1], [17] to pursue the discussion.

¹This work was funded by the CELAR (French Army) under contract 0054192

1.2 Architecture

ISO proposes in [12] different methods to develop architectures for conformance testing: a local testing method and three global testing methods. In local testing, all the testing environment is located in the system under test (SUT) containing the implementation under test (IUT). This environment comprises an upper-tester (UT) interacting with the IUT service interface, a lower tester (LT) interacting with the IUT lower interface, and coordination procedures (PCT) between UT and LT. In global methods, LT is located outside the SUT and communicates with the IUT through the underlying service, which is supposed to be correct. Three global methods have been defined: a distributed method, a coordinated method and a distant method. They differ by their PCT, their ability to access the service provided by the IUT and the UT features. They do not have the same error detection power [7].

The problem of interoperability requires one to use these architectures differently. Considering an implementation (i.e. a black-box parallel program) and a global property specifying the intended behavior of the program, a natural approach to testing is to “close” the system with upper-testers, located on each protocol entity. These testers have to feed the protocol implementation with requests and to decide whether the reactions conform or not with the expected behavior, as defined by the protocol service. We focus our attention to the design of local observers, capable to decide a global property by observing only local requests and indications. Obviously, these observers must exchange some information to catch a global view of the system state. We will see this can be achieved by piggy-backing on the protocol interactions, and we will propose testing architectures to implement that in section 2.3.

Let us consider for example the architecture given in figure 1a. This is the simple case where two protocol entities cooperate through a communication medium. The protocol service can be tested at the user level.

Figure 1: Testing architecture (a) and a data transfer service (b)

Our problem is to reliably detect logical errors, where “logical error” is defined on the intended partial ordering of events. We are only concerned with relative event orderings; no consideration is made of the issues associated with real-time or “performance” debugging.

1.3 Example

Let us consider a data transfer protocol between a sender and a receiver. The sender can initiate a transfer by calling a Data Request service primitive. From the service point of view, this must cause a Data Indication on the receiver. After delivering the Indication, the protocol informs the sender by a Confirmation, which allows the sender to emit a new request. The service behavior can be modeled by a simple sequential finite automaton shown in figure 1b. Here, errors are simply defined as events that are not expected in the different states of the automaton.

We suppose the service is achieved through a faulty communication medium, using protocol data units for retransmissions and acknowledgments. Figure 2 shows the space-time diagram of erroneous execution.

Figure 2: Example of execution: the observable events are figured as dots on the "time-line" of processes. Arrows represent the message exchanges.

Clearly, local testers cannot give a reliable diagnostic without knowing the dependence relations between the occurrences of SDUs. Next section is devoted to the computation of these dependencies.

2 On-line computation of the causal dependencies

2.1 Message causality

Formally reasoning on execution traces requires a simple mathematical model. Two different levels can be distinguished:

- The user's level, where the programmer defines what he wants to trace: actually the calls of the different Service Data Units of the protocol. These are actions occurring during the computation. The action occurrences are called *observable events*. The action name defines the event type (or *label*).
- The run-time level which induces some constraints on the relative ordering of observable events. Precisely, that is the causality due to the physical exchange of messages between processes, ensuring that the sending of a message must always precede its reception.

Let us consider a finite set of processes, denoted by P_1 to P_n (one considers for simplification this set is known in advance and stable). We denote $E = X \uplus \bar{X} \uplus O$ the finite set of events occurring during the execution. X contains all the sending events (unobservable in principle), $\bar{X}$ the corresponding receptions (for the sake of simplicity, communication channels are considered point-to-point) and O the observable events. One can consider that E is split up into the disjoint subsets E_i of events that are local to P_i : $E = \bigsqcup_{1 \leq i \leq n} E_i$.

Let $\prec_i$ be the sequential ordering of events appearing on process P_i . The causal relationship (whose definition is given in [14]) in E^2 is the smallest relation $\prec$ satisfying :

1. $\forall i \in \{1..n\}, \forall x, y \in E_i, x \prec_i y \implies x \prec y$
2. $\forall x \in X, x \prec \bar{x}$
3. $\prec$ is transitive

$\prec$ is a partial order (poset)². In the sequel, we will only consider the sub-order $\tilde{O}$ induced by the set O of observable events. This leads to our trace model: *a distributed trace is modeled by a finite poset $\tilde{O}$, split up into n disjoint chains³ and equipped with a labeling function λ from O to an action alphabet Σ* . Figure 3 shows the Hasse diagram⁴ of the associated poset of our trace example of figure 2.

Figure 3: The Hasse diagrams of the poset and the associated lattice of consistent cuts: each observable event is denoted by its vector time-stamp, and labeled by {REQ, IND and CNF}. The lattice figures all the different ways to totally order the events while preserving causality.

There is another interesting representation of the poset which is the “lattice of consistent cuts” or “ideal lattice”⁵, denoted by $\widetilde{I(\tilde{O})}$ (see figure 3). The maximal paths in

² $\prec \cup \prec^{-1} = \emptyset$ and $\prec^2 = \prec$

³ a chain is a totally ordered subset

⁴ an event e immediately precedes an event f iff e is drawn under f and there is an edge between them

⁵ Each node of this lattice is in one-to-one correspondence with subsets of events, closed by precedence (all the predecessors of events belonging to the subset are also in the subset): such subsets are called ideals in poset theory.

the diagram of this lattice are the different ways to totally order the events such that causality is preserved (i.e. linear extensions). So this lattice codes all possible interleaving of events and provides a simple way to deal with properties about concurrency. This notion is closely related to the concept of observation in distributed systems. We do not describe here algorithms to compute this structure. Efficient and on-line ones may be found in [6], [5]. The lattice is built on the fly, growing step by step at each occurrence of an observable event endowed with its time-stamp.

2.2 Vector time-stamps

The first idea for efficiently coding the causal relation is to consider the chain decomposition induced by processes. Let us consider the following coding into vectors of integers:

$$\begin{aligned} \Delta : O &\longrightarrow \mathbb{N}^n \\ x &\longmapsto \lfloor \downarrow x \cap O_i \rfloor_{i \in [1..n]} \end{aligned}$$

where $\downarrow x = \{y \in O / y \preceq x\}$.

One recognizes the vector clocks of Fidge and Mattern [8, 15] (see figure 4).

Figure 4: Vector time-stamps: they are associated to each observable event, and denoted by a couple of integers. We have also figured the piggy-backed values and the updating upon receptions.

This coding is obtained on-line as follows:

- each process P_i maintains a vector Δ_i of size n (initially set to the null vector).
- when an observable event e occurs on P_i , $\Delta_i[i] := \Delta_i[i] + 1$.
- when P_i sends a message, the current value V of the vector Δ_i is piggy-backed.
- upon reception on P_i of a message from P_j carrying a vector V ,
 $\forall k \in [1..n], \Delta_i[k] := \max(\Delta_i[k], V[k])$.

The vector time-stamps are the basis technique to code on-the-fly the causal dependence⁶. There exist different ways to minimize the amount of information to be piggy-backed. A

⁶Since $\forall x, y \in O, x \preceq y \Leftrightarrow \Delta(x) \leq \Delta(y)$

first kind of solution is to observe a less precise information, but enough informative to allow the reconstruction of the dependence relation (direct dependencies for example [9], [3]). A second approach is to exploit some characteristics of the networks of processors (Fifo channels for example [18]).

2.3 Possible implementations

At first glance, the technique seems difficult to implement, since the testing environment has to capture the messages, piggyback some information and perform the updating at every receptions, although only SDUs are observed. We show there are reasonable ways to implement these things.

Two different problems can be distinguished: the observation of events, especially the PDUs, and the piggy-backing of information.

The SDUs are observable since they are directly connected to the upper-testers. More difficult is the capture of the PDUs, sent and received by the protocol entities. One can think about two different methods as illustrated in figure 5.

Figure 5: Some testing architectures

The most satisfactory method is to use modern implementations of protocols by streams. In such implementations, it is not difficult to take out the message flow to manage the time-stamps. For the others implementations, it is always possible to catch the communication flow using network connections (see for example the Astride architecture [16]). In this architecture, the IUT is directly connected to the testers which relay the PDUs by communicating each other (see figure 5). This implicitly assumes that the local connection offers a reliable Fifo service.

The second problem is to implement piggy-backing. It does not seem reasonable to require that there is always enough place to insert information in PDUs. Most protocols fill completely the SDUs provided by the network service: this is the case notably when the protocol ensures fragmenting and reassembling. One solution is to send separately the time-stamp information, provided there is a way to associate the time-stamp with the corresponding message. The problem here is to guarantee the atomicity of the piggy-backing technique. A simple solution consists in assuming that connections are Fifo. In case of message loss, the receiver tester must also lose the time-stamp information. Also, loss of time-stamps must be considered as loss of the associated PDU (detection

might be achieved by a timeout mechanism). We do not elaborate further on this question: solutions do exist, their relevance depends on the protocol under test and of its environment.

3 General methods to compute global properties

Let us come back to our example. Now to each SDU is associated a time-stamp information, which can decide of their causal ordering. We see that error can be detected in the case of our example ($REQ(31) < IND(32) \not< CNF(41)$), provided that the different events can be localized to a single place.

We can distinguished two different methods:

- local checking [11, 2, 10], in which the property can be decided on a single local tester: this limits to ask questions only on the past of events, but detection is performed without delay (pronouncement of validity is made upon the occurrence of each observable event).
- global checking [4, 13], in which the analysis is transported to a global observer (see figure 6). In that case, more complex questions can be answered (concurrency between events for example).

These methods are under development in the domain of distributed debugging. We are involved in the process of unifying them by the mean of formal models like posets and lattices. We present in the following two general algorithms which support properties expressed by state machines.. Global checking allows to check complex properties which may characterize a single snapshot, but its cost is exponential in the size of the set of observable events⁷. On an other hand, local checking, which is linear, allows to check only properties dealing with events occurring only in the causal past of a single observable event. The exact characterization of the checking power of the two methods is still an open question. The other question is to know what class of properties is needed to check interoperability. At present, we consider that the whole protocol service must be testable, which advocates in favor of global methods.

3.1 Global checking

The diagram of the ideal lattice $I(\widetilde{O})$ of the poset $\widetilde{O}$ can be viewed as a labeled transition system:

$$Tr(I(\widetilde{O})) = \langle I(O), \Sigma, \longrightarrow \rangle$$

where the set of states is the set $I(O)$, Σ is the alphabet of actions (which occur as observable events), and $\longrightarrow$ is defined by the edges $I \xrightarrow{a} J$ of the diagram.

In each maximal path of the lattice, the successive encountered labels form a word on Σ .

A finite (deterministic) automaton is a 5-uple $\phi = \langle \Sigma, Q, q_0, F, \delta \rangle$, where Σ is the action alphabet, Q is the set of states, q_0 is the initial state, $F \subseteq Q$ is the set of accepting states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition function.

⁷This is quite standard for people involved in verification.

Figure 6: Global checking

These automata allow to specify properties about the potential or necessary sequencing of actions.

Let $\mathcal{L}(\phi) = \{u \in \Sigma^* : \delta^*(u, q_0) \in F\}$ be the language defined by ϕ . Confronting such an automaton with the diagram of the lattice requires to consider, for each vertex I , the set $Path(I)$ of all paths from bottom to I . This leads to two possibilities⁸ when defining the satisfaction of a property ϕ in an ideal I :

$$\begin{aligned} I \models_{\forall} \phi & \quad \text{iff} \quad \lambda(Path(I)) \subseteq \mathcal{L}(\phi) \\ I \models_{\exists} \phi & \quad \text{iff} \quad \lambda(Path(I)) \cap \mathcal{L}(\phi) \neq \emptyset \end{aligned}$$

The satisfaction relation $\models_{\forall}$ (resp. $\models_{\exists}$) states that a vertex I satisfies ϕ iff *every* (resp. *some*) path in $Path(I)$ is labeled with a sequence accepted by the automaton ϕ .

From the above definitions, we can easily deduce that:

$$\begin{aligned} \forall I \in I(O), I \models_{\forall} \phi & \quad \Leftrightarrow \quad \phi(I) \subseteq F \\ \forall I \in I(O), I \models_{\exists} \phi & \quad \Leftrightarrow \quad \phi(I) \cap F \neq \emptyset \end{aligned}$$

where $\phi(I) = \delta^*(\lambda(Path(I)), q_0)$ is the set of states of the automaton ϕ reachable by paths ending in I .

The checking algorithm is essentially based on the computation of $\phi(I)$. It is easy to see that $\phi(I)$ can be computed on-line during the lattice construction since it is sufficient to attribute the lattice vertices by states of the automaton ϕ . In fact, the attribute $\phi(I)$ of a vertex I can be computed from the attributes of its immediate predecessors in the lattice⁹:

$$\phi(I) = \bigcup_{J: J \rightarrow x} \bigcup_{q \in \phi(J)} \delta(x, q)$$

The algorithm can be implemented by a breadth-first traversal of the lattice during which each vertex is attributed with a boolean array of size $|Q|$, the i^{th} element indicating whether some sequence leading to I can put the automaton in state i .

The complexity of the checking algorithm is linear in the size of the lattice.

Figure 7 illustrates the result of algorithms for checking $\models_{\forall}$ and $\models_{\exists}$.

⁸Let $\neg\phi$ denote the complementary automaton of ϕ . $I \models_{\forall} \phi \Leftrightarrow I \not\models_{\exists} \neg\phi$.

⁹This supposes that events are treated according to a linear extension of the poset.

Figure 7: Automaton verification $\models_{\forall}$ and $\models_{\exists}$. The values of $\phi(I)$ are written on each vertex I : vertices I such that $I \models_{\forall} \phi$ are colored in grey; Vertices I such that $I \not\models_{\exists} \phi$ are colored in black. E is the error state of ϕ . We conclude that there exists two (black) global states (figured by cut lines in figure 3), such that all the observations detect an error (IND expected). For the grey states, there exists at least one observation which would discover the error.

3.2 Local checking

Now, we consider that the automaton interprets the chains of the causality relation, instead of the interleaving. This means we are concerned only with the control flow of events and not with the different possible observations as previously. For an event x , we consider words which are the concatenation of label of events leading to x , i.e. label of chains in $\downarrow x$. We are going to interpret those chains as a path, a sequence of actions which *must be done* before x , thus we will only consider the maximal chains.

Formally, let

$$\begin{aligned} \text{Path}(x) &= \{A \subseteq \downarrow x, A \text{ is a chain of } \tilde{O} \text{ and} \\ &\forall B \subseteq \downarrow x \text{ such that } B \text{ is a chain of } \tilde{O}, A \not\subseteq B\} \end{aligned}$$

be the set of maximal chains of $\downarrow x$.

Similarly to the preceding section, the problem of deciding $x \models_{\forall} \phi$ or $x \models_{\exists} \phi$ comes to compute

$$\phi(x) = \bigcup_{y \in \text{max}(\downarrow x)} \bigcup_{q \in \phi(y)} \delta(\lambda(x), q)$$

The problem when computing $\phi(x)$ is to distinguish immediate predecessors of x from transitive predecessors. Each process P_i will maintain a vector of states ϕ_i , which is the set of reached states of the automaton. We have to manage the “contribution” of each process P_j for ϕ_i and detect the transitive contribution (a contribution can become transitive after a while). The distributed algorithm is the following :

- Each process P_i maintains a vector ϕ_i , initialized to $\forall k \in 1..n, \phi_i[k] = \{q_0\}$. It also manages the vector time-stamp V_i , initialized to $\forall k \in 1..n, V_i[k] = 0..$

- Upon observation of $e \in O_i$:
 $V_i[i] := V_i[i] + 1$;
 $\phi_i[i] := \bigcup_{k \in 1..n} \bigcup_{q \in \phi_i[k]} \delta(\lambda(e), q); \forall k \neq i, \phi_i[k] := \emptyset; / * \phi(e) = \phi_i[i] */$
- Upon sending : join ϕ_i and V_i to the message
- Upon reception of a message from P_j , containing ϕ_j and V_j :
 $\forall k \in 1..n$
 if $V_j[k] > V_i[k]$ then $V_i[k] := V_j[k]; \phi_i[k] := \phi_j[k];$
 else if $V_j[k] = V_i[k]$ then
 if $\phi_i[k] \neq \emptyset$ then $\phi_i[k] := \phi_j[k];$

In local checking, the global information is accumulated through the communication network, and coded by vectors of integers and vectors of states. Figure 8 shows how this information flows, and figure 9 explains the result of the verification.

Figure 8: On-the-fly analysis: the different values are written $V[1]\phi[1]V[2]\phi[2]$. “.” denotes the empty set $\emptyset$. The values of $\phi(x)$ for each observable event x are circled.

4 Conclusion

Dealing with the problem of interoperability testing, we have studied how to observe a global behavior defined at the service level. Because of the distributed nature of protocols, some special techniques are required to catch the causal dependencies and the possible interleaving of events, in order to provide a reliable diagnostic. We advocated in favor of techniques developed in the area of distributed debugging: time-stamping and trace-checking. We have just presented the general motivations, the relevant techniques and gave some hints of checking algorithms. Their effective use for observing interoperability has to be further elaborated. Some possible implementations were sketched. This paper is a first piece of work aiming at bringing the conceptual framework of distributed system modeling to its practical use for testing real protocols.

Figure 9: Automaton verification $\models_{\forall}$ and $\models_{\exists}$: we see that there exists two events (black) for which all the causal paths leading to them violates the property. For the grey events, this is true for at least one path.

References

- [1] N. Arakawa, M. Phalippou, N. Risser, and T. Soneoka. Combination of conformance and interoperability testing. In *Formal Description Techniques, V*, pages 397–412, Ifip Transactions C-10, North-Holland Publishing Company, 1992.
- [2] O. Babaoglu and M. Raynal. Specification and detection of behavioral patterns in distributed computations. In San Diego Springer Verlag Series, editor, *4th IFIP WG10.4 Int. Conference on Dependable Computing for Critical Application*, 1994.
- [3] P. Baldy, H. Dicky, R. Medina, M. Morvan, and JF. Vilarem. *Efficient reconstruction of the causal relationship in distributed computations*. Technical Report 92-013, LIRMM, June 1992.
- [4] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, May 1991.
- [5] C. Diehl, C. Jard, and JX. Rampon. *Computing on-line the covering graph of the ideal lattice of posets*. Internal Research Report 703, IRISA, Rennes, February 1993.
- [6] C. Diehl, C. Jard, and J.X. Rampon. Reachability analysis on distributed executions. In JP. Jouannaud MC. Gaudel, editor, *Proc. TAPSOFT,93 LNCS 668*, pages 629–643, Springer-Verlag, Orsay, Paris, April 1993.
- [7] R. Dssouli and G.V. Bochmann. Conformance testing with multiple observers. In *6th IFIP International Workshop on Protocol Specification, Testing and Verification, Montréal, Gray rock*, North Holland, June 1986.

- [8] J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Proc. 11th Australian Computer Science Conference*, pages 55–66, february 1988.
- [9] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *10th IEEE International Conference on Distributed Computing Systems*, pages 134–141, 1990.
- [10] E. Fromentin, C. Jard, G.V. Jourdan, and N. Raynal. *On-the-fly Analysis of Distributed Computations*. Technical Report To appear, INRIA, Centre IRISA, Rennes, September 1994.
- [11] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of the 12th conf. Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, Lecture Notes in Computer Science 652, Springer-Verlag, New Delhi, India, December 1992.
- [12] ISO. *Information Processing Systems, Open Systems Interconnection, OSI Conformance Testing Methodology and Framework*. ISO 9646.
- [13] C. Jard, T. Jéron, G.V. Jourdan, and J.X. Rampon. A general approach to trace-checking in distributed computing systems. In Cellary W., editor, *Proc. ICDCS*, page , Poznan, Poland, June 1994.
- [14] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [15] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *Proc. Int. Workshop on Parallel and Distributed Algorithms Bonas, France, Oct. 1988*, North Holland, 1989.
- [16] O. Rafiq. The astride testing approach: principles, tools and carrying out. *Computer Standards and Interfaces, Vol 11(2)*, 85–94, 1990.
- [17] O. Rafiq. Le test d’interopérabilité des protocoles. In *CFIP’91, Ingenierie des protocoles*, Hermes, Septembre 1991.
- [18] M. Singhal and A. Kshemkalyani. *An efficient implementation of vector clocks*. Technical Report, Ohio State University, Departement of Computer and Information Science, Oct. 1990.