Observer—A Concept for Formal On-Line Validation of Distributed Systems

Michel Diaz, *Senior Member, IEEE,* Guy Juanole, and Jean-Pierre Courtiat, *Member, IEEE*

***Abstract*— This paper proposes the observer concept for designing self-checking distributed systems, i.e., systems that detect erroneous behaviors as soon as errors act at some observable output level. The approach provides a solution to build systems whose on-line behavior is checked against a formal model derived from a formal description. In other words, the actual implementation is continuously checked against a reference, this reference being a formal and verified model of some adequately selected aspects of the system behavior. The corresponding methodology, the software concepts and some applications of the observer are presented. General definitions are given first that theoretically define self-checking systems as systems that include and implement complete on-line validation. The basic concepts and the difficulties to implement self-checking validation are then given. In order to provide simple implementations, the previous definitions will be weakened to design quasi self-checking observers for LAN's using a broadcast service. Three specific applications are given to illustrate the proposed approach: testing a virtual ring MAC protocol, checking the Link and Transport layers in an industrial LAN, and managing a complete OSI layering, from layer 2 to layer 6 in an open system architecture.**

***Index Terms*—Distributed systems, run-time validation, testing, verification, Petri nets based models, performance measurements, layered distributed architectures, formal description techniques.**

I. Introduction

RECENT developments lead to the design of complex distributed systems where functional and performance validations are of high importance. Designing such systems features, as an ultimate goal, the validation of actual implementations, which consists in certifying the conformance of the implemented system with respect to some predetermined specifications. The conformance is usually carried out by defining, during the design, a set of adequate test suites that are applied to the system that will then react to these specific stimuli. Of course, such testing experiments are always performed while stopping the normal behavior of the system.

Moreover, as faults or errors of any sort can occur during the life of the system, it is obvious that they cannot be tested at the very moment they occur by using those conventional approaches. This is because run-time checking of the effects of faults on system behaviors needs to be carried-out continuously. Enforcing on-line testing would allow one to detect, during normal behavior and as soon as they occur at some observable level, the influence of hardware failures, software bugs or human malfunctions. As checking is continuous, it is also able to provide a simple way to obtain experimental figures about actual system reliability and actual system performances. This paper aims at proposing the observer concept for validating run-time behaviors of distributed systems, and more precisely for designing self-checking communicating systems, i.e., systems that detect erroneous behaviors as soon as errors act at some observable output level. The given approach also provides a way to design systems whose on-line behavior is checked against a formal model derived from a formal description. In other words, the actual implementation is continuously checked against a reference, this reference being a formal and verified model.

In distributed systems, distributed processes run in parallel on several distant processors, and the interactions taking place between these processes can be very complex. Hence, enforcing on-line behavioral validation is quite challenging and important: distributed architectures are very difficult to define, build, and test, due to their geographical distances and intricate interactions. Because of their interests, this paper only addresses distributed systems and shows how their interactions can be formally described and processed in order to derive formal run-time checking.

The observer concept was first proposed for parallel, non distributed systems [1], and afterwards extended to distributed systems [17]. This paper generalises the previous work by presenting a general design methodology, applicable to any distributed system. The corresponding approach starts from the formal concept of self-checking systems. Unfortunately, on-line self-checking systems appear to be quite difficult to implement. It is then shown how the formal on-line checking definitions can be weakened in order to lead to a real and simple implementation of an on-line observer. Developing the observer provides an efficient method for designing quasi self-checking distributed systems. It also includes the development of a implementation support tool that proved to be able to debug, observe and evaluate the system behavior. The designed on-line checked system is made up of two parts: a worker, which is the actual implementation, and an observer, which is defined as a design of selected distributed mechanisms. As illustrative examples, some applications in industrial distributed control and office automation networks will be given. Experiments on another important application area, electronic switching systems, appear in [2].

Section II discusses the general validation problem in distributed systems and gives the main principles that define formal observation and formal checking of communicating

Manuscript received July, 1989; revised May, 1993.

The authors are with the Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique, 31077 Toulouse-Cedex, France.

IEEE Log Number 9406729.

systems. It defines in the most general way on-line self-checking systems, introduces the observer concept, and emphasises its interests for on-line checking. The conceptual difficulties are then discussed and a weaker solution is proposed that leads to the concept of quasi self-checking systems. It is then shown how the needed formal observers can be designed and easily implemented to validate the behavior of multilayered LAN's based on the use of a broadcast service.

Section III gives two significant areas of application of the observer. The first one presents a protocol observer for a local network, REBUS, used in order to check a MAC (Media Access Control) layer, a token-bus protocol, and reports some significant results. The design of an open multilayered protocol observer is given in the second part and applied to two specific LAN's of interest: a multilevel industrial LAN and an open office automation LAN. These multilayered observers are able to separately consider each of the protocol layers and provide, as main facilities, run-time checking and performance analysis.

II. Formal Observation of Distributed Systems

The key to distributed software quality is reliability [4]. Two aspects must be investigated in order to design correct software. The first one concerns the reliability with which the software specifications are adequately described and correctly implemented in the actual implementation; the second one deals with checking, during run time, the correct behavior of the implemented system in actual environments including hardware failures, software bugs and human errors. This paper also gives an approach that relates these two design phases.

Usually, distributed systems are tested by suspending either the whole system operation or part of it [5]. Starting with hardware, different studies have been carried out in order to handle errors on-line, at run time. Those studies are based on the concepts of self-checking systems or, more generally, on the concepts of distinctness [8]–[10], where distinctness globally means alternative ways of performing a specific task, by using as far as possible separate hardware and different softwares.

Previous approaches developed for system validation either do not address run time checking by using formal approaches or do not consider distributed systems, as validation functions are located in different processes and are unable to check the global behavior.

The on-line validation approach for complex distributed systems considered in this paper uses as a starting basis the concepts resulting from hardware self-checking systems. The following definitions extend to distributed systems the ones proposed for sequential machines [26], [27]. They will serve as the main basis for discussing distributed on-line checking.

Let us consider a given system and assume that some outputs belong to a well defined set of admissible values S.

Definition 1: A system is fault-secure for a set of faults F, if for any fault f belonging to F, and for any run time behavior B of the system, the outputs deliver:

a) either the correct value, as if there were no fault,
b) or one or more erroneous faulty values, but these faulty values are such that they do not belong to the admissible value set S.

Then, fault-secure systems are systems where faults may be enforced not to propagate. This is because: either the faults are not visible and have no effect; or the faults have visible effects that affect the outputs, but it is easy to notice that an error exists at the output (and if needed take appropriate action) as the output value is outside the known admissible set of error free values.

Fault-secureness is a safety property: undetected output errors cannot occur as an output value cannot at the same time be faulty and belong to the set of the nonfaulty values. It follows that either the output is correct or the output is incorrect but the false value can be detected as soon as it appears at the output level.

Note that one possible mechanism for such a detection is to use adequate coding at the output level: if the fault has an effect, then the output value must not belong to the selected code.

Definition 2: A system is self-testing for a set of faults F, if whatever a fault f belonging to F, there exists a specific testing behavior Bt, occurring during the run-time behavior of the system, such that this fault will be propagated to the output as a value out of the admissible set S.

This property is of importance as it enforces the detection of all faults in a selected set F of fault assumptions. This results because, assuming that fault $f1$ exists and is not detected, then another fault $f2$ can occur and the resulting compound fault "$f1$ together with $f2$" may possibly be out of the selected fault set F. The effects of the "$f1$ together with $f2$" fault has not been considered during the design because of the fault assumption, and the resulting output value resulting from the "$f1$ together with $f2$" fault may be wrong, i.e., faulty and belonging to the set of admissible values. In the case of a parity coding, such a possible behavior is obtained when two errors arise and leads to a value which is false but belong to the code.

Self-testing is a liveness property as it implies that any fault will eventually be propagated as a faulty output value.

Note that self-testing alone allows the fault, before being propagated out of S, to give a wrong undetected output value, i.e., an output value different from the output value without fault but where this output value belongs to the set S. Self-testing is not sufficient as: if the system is only self-testing, then faults should appear at the output as a value out of S; the problem is that, as nothing else is stated, before becoming observable at the output, the faults may give output values which are both false and belong to S, so are not detected.

From the previous comments, it follows that an adequate system behavior is obtained by the following definition.

Definition 3: A system is self-checking for a set of faults F, if whatever a fault f belonging to F, it is fault-secure and self-testing.

A self-checking system has safeness and liveness properties regarding the effects of its faults. Definitions 1 and 2 show that B and Bt behaviors are of fundamental importance. It will be shown that the difficulty come with Definition 2, because of Bt, the set of the test behavior, that includes the needed test sequences: as checking is performed on-line then the sequences in Bt must belong to the set of the sequences that occur during run-time behavior.

Definition 1 can be fulfilled by using coding. Unfortunately, it is generally quite difficult to define and process coded values in distributed software.

As an extension of the duplication principle in hardware, a possible solution, that will be the one considered in this paper, is to restrict the classes of self-checking implementations to the ones that use the distinctness concept: the system to be validated is designed as being composed of two distinct subsystems, a worker and an observer.

Definition 4: An observer-worker system is a (potentially run time checked) system that is constituted of two distinct components, a worker and an observer: the worker is a classical implementation of the system behavior and the observer is a given redundant implementation whose outputs are comparable with the outputs of the worker.

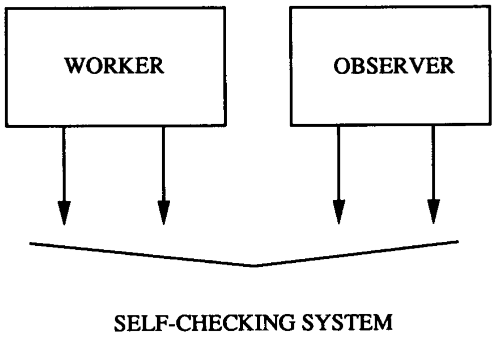


Fig. 1. Principle of a simple self-checking.

A. Observer Principle

Comparing outputs implies observing behaviors. This requires:

1) Redundancy. If a system has redundant copies, run-time faults occurring within one copy lead to a discrepancy between the behavior of this copy and the behavior of the nonfaulty copies. Detecting the fault means detecting the different behavior of the faulty copy. Consequently, the simplest redundant system can be implemented using two distinct copies, here called the worker and the observer.
2) Reference. The behavior of the subsystem that is being checked must be precisely known and defined. It will be shown in the sequel how this well defined representation of the global behavior, the observational model, can be derived from a formal specification.
3) Visibility. Checking the worker behavior implies that the observer must know and access given events of the worker. The resulting observer-worker relationship which has to be implemented can be of two different types:
   a) the worker cooperates with the observer by explicitly informing the observer when significant events occur,
   b) the worker behavior can be spied by the observer, needing no specific action from the worker.

An observer-worker cooperation of type a) enables the observer to get any information it needs from the worker but both the worker design and its software have to be modified to send this adequate information to the observer.

In the observer-worker cooperation of type b), the knowledge of the worker behavior is obtained from some set of the worker information that is directly accessible to the observer. The worker is not modified by the presence of the observer:

—observer and worker designs and implementations can be made independent, and can be performed by distinct approaches and teams,

—the same observer can be used for checking different implementations of the worker.

As a consequence, type b) spying cooperation will be considered in this paper.

An attempt to use type a) cooperation appears in [2]. Fig. 1 gives the basic design architecture. From Fig. 1 and the previous definitions, it follows:

Property 1: In a observer-worker system, if

a) F is the set of all faults occurring only in one of the two distinct subsystems,
b) each subsystem is such that its set of fault detection sequences are applied during run time,

then the global system is self-checking.

Proof: Fault secureness follows because the design is fault-secure: a fault in one of the two subsystems will only affect its outputs; checking its output values with respect to the ones of the non faulty part enforces the needed detection.

Fault testing comes from the fact that each of the two subsystems is tested during run time, giving an erroneous output. Comparing the erroneous tested outputs with the correct ones of the other component indicates the error. Self-checking follows from Definition 3.

Of course, part b) of Property 1 needs the system to be designed in a specific way. The difficult assumption to be fulfilled is to enforce that all possible faults in one (each) component are tested during run-time. Furthermore, from the definitions, this set of tests must be applied before one fault occurs in the other subsystem. This is because if another fault occurs, the resulting compound fault falsifies the fault assumption: both faults, one in each subpart, may compensate each other and give a global output whose value could be wrong and could have equal (identical) values.

Due to the resulting complexity for large systems, such a formally proved run-time checked design, although possible, will not be considered here. For sake of applicability to any distributed systems, it is assumed in what follows that the worker is not designed in any specific way, and as a consequence can be any distributed system implementation.

It follows that as many as possible different sequences should be applied to the worker during its usual behavior in order to (as soon as possible) detect faults that may occur.

Let us now consider the design of the observer. Part b) of Property 1 implies that a run-time tested implementation of the observer also needs a nonclassical, specific test based design. Again, such a specific design, although simpler that the one of the worker, still seems not easy to realize for observing any complex distributed system.

Although being of interest for highly reliable systems, the approach developed here does not consider such a self-test based design, but instead proposes to develop a quasi self-checking observer, called a formal observer: this quasi self-checking observer will be designed as reliable as possible, being based on a formal model and on an exhaustive verification of this formal model.

Definition 5: A formal observer is a subsystem designed to check distributed behaviors where:

a) its software is independent of the specific protocols to be checked in the considered system;
b) its data are defined by the protocols to be checked and this data can be formally specified and verified.

Property a) implies that the observer can be built and used for a family of distributed systems. So it can be made very reliable.

From property b), there must exist a method to specify and verify the correctness of the observer data, i.e., the specific system data depending of the distributed system protocols, before these data are given as an observation input to the observer. Such a formal observer can be seen as a correct implementation of the formal model of some observed aspects of the behavior. It will be seen that Petri nets have been used to build the formal models. It follows that the design of the system consists of the following steps: write a description of the behavior of the system to be implemented; implement the system itself, i.e., the worker; from the description of the worker, select (based on experience) that part of the behavior which should be observed and write a formal model of it. This specific formal model, corresponding to the selected behavior, defines the data of the observer.

Definition 6: A system is quasi self-checking if

a) it is an observer-worker system, and
b) the observer is a formal observer.

Of course, the input data of the observer are of prime importance. It is proposed in this paper to use formal models of the behaviors to be checked as input data for the observer. These inputs, being used as references for checking behaviors, must be reliable and correct: formal models must be used, as they provide precise descriptions and support validation and verification algorithms. In order to be fully integrated in the software life cycle, formal models will be derived from system specifications: the relationships between the specification, the implementation and the observer are given in Fig. 2(a) that shows how an observer is related to a system specification.

It follows that comparing the worker and observer behaviors, the former being the actual implementation and the latter being a formal model, provides a basic on-line mechanism able to detect any discrepancy between both behaviors. This quasi self-checking approach owns some features of interest:

a) on-line fault detection follows from detecting a mismatch between the two subsystem behaviors, one being formally described,
b) well defined events can be selected for validation and run-time checking and appear in the formal model.

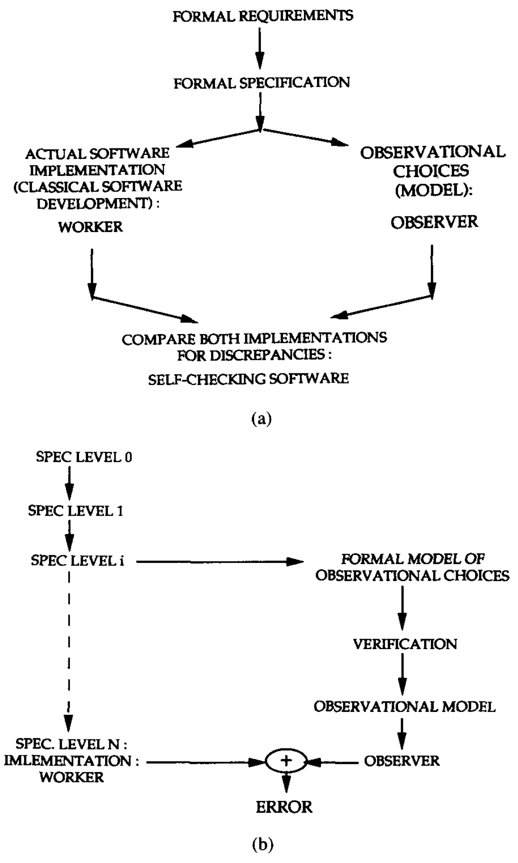

FORMAL REQUIREMENTS
FORMAL SPECIFICATION
ACTUAL SOFTWARE IMPLEMENTATION (CLASSICAL SOFTWARE DEVELOPMENT) : WORKER
OBSERVATIONAL CHOICES (MODEL): OBSERVER
COMPARE BOTH IMPLEMENTATIONS FOR DISCREPANCIES : SELF-CHECKING SOFTWARE
(a)
SPEC LEVEL 0
SPEC LEVEL 1
SPEC LEVEL i
FORMAL MODEL OF OBSERVATIONAL CHOICES
VERIFICATION
OBSERVATIONAL MODEL
SPEC. LEVEL N : IMLEMENTATION : WORKER
+
OBSERVER
ERROR
(b)

Fig. 2. Relationship between implementation and checking. (a) Observation and checking. (b) The levels of specification and observation.

For actual complex implementations, it does not make any sense to observe the complete behavior of the distributed systems. Too many events may occur in the worker. Many of these events can be classified as internal, with little interest, and in practice cannot be traced during run-time. Consequently, observational models do not need to represent complete behaviors but only partial behaviors of interest [Fig. 2(b)]. With this in mind, selecting the formal model becomes one important design choice. It should be able to specify at some level of detail, any system specifications and be validated with respect to requirements of correctness.

It follows that the formal model must be able:

—to express simplified specifications of distributed systems;
—to support verification procedures;
—to be able to act as a basis for implementing the observer.

Different formal models and description techniques [12] have been developed to specify behaviors of distributed systems. The most known ones are Extended State Machines [13], [15], Petri nets based models [11], [14], the ISO and CCITT formal description techniques ESTELLE, LDS, and LOTOS [20], [21], [30].

Petri nets have been selected in this paper for defining the observer models as they can be:

—analyzed by efficient methods and tools,

—animated by easy to implement simulators,

—derived as observational models from protocol descriptions.

Classical Place-Transition Petri nets do not represent data. Then if Place-Transition Petri nets are used, only control structures can be taken into account by such Petri net based observers, as it will be shown later on.

Of course, if more descriptive power is needed, then more sophisticated models must be used in a similar way as observational models. For instance, should an observer have to check complex data, and as complex data cannot be easily represented by Petri nets, then Predicate-Transition Petri nets [14], the Formal Descriptions Techniques Estelle [20] or LOTOS [21], could be selected, at the expense of a more difficult verification.

It follows here that the worker is an implementation performing the required behavior and that the observer contains an observational Petri net model, also called the mission model, derived from the specifications of the communicating system. Let us now consider how to derive the observational model.

B. Observers in Distributed Systems

Validation of distributed systems must account for all global interactions and must address all properties related to the communications occuring among processes. As processes interact using protocols, observing the global system behavior means observing the protocols, i.e., the way processes communicate.

The more general framework in which quasi self-checking distributed systems could be developed is heterogeneous open architectures. As a consequence, it was decided to design the observer concept inside the ISO-CCITT OSI Reference Model framework [7]. The OSI Reference Model assumes a layered architecture and defines a set of layers, each layer providing services to the layer above it; within a layer, the corresponding entities communicate by exchanging messages following a given protocol and using the services of the layer below. It seems that such a structuring, consisting of a given set of layers, is the only way to successfully design complex communication systems.

In layered architectures, the worker is the set of processes distributed on several processors which perform all application functions and the observer is a dedicated processor.

As the spying cooperation mode has been selected (see Paragraph II-1), it follows that any layered architecture has to provide a way to read the required information. Considering protocols, and given the fact that workers cannot be modified, then the observer must be made able to know all or some of the exchanged information, i.e., the messages.

For achieving a simple design of the observer concept, it will be assumed here that observers are designed in order to be applied to distributed systems that communicate using broadcast mechanisms among the worker processors, for instance using a physical bus in local area networks. Of course, spying observers will then be easy to implement as all exchanged messages are potentially able to be received by an observer connected to the broadcast service (Fig. 3). It will be shown

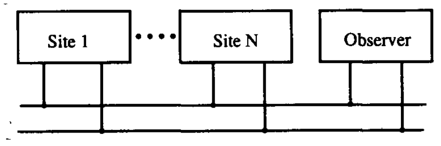


Fig. 3. The observer on a bus architecture.

that the difference with classical hardware protocol monitors is that the observer is programmed at a very high level of description by using the formal model: then any change of the protocol to be checked will only mean to modify the formal model.

The observer can be connected to the worker through the shared broadcast support as simply as any functional node of the network. Any broadcast information becomes directly available to the observer as, by listening to the physical layer bus, it can follow all exchanges of messages between the communicating processes: it must be able to receive at the physical layer all messages which are sent on the broadcast medium.

Of course, layered architecture means that, starting from the physical layer, the observer has to check a hierarchy of protocols. Let us note that, for simplicity, it seems adequate to debug and validate the system by considering one layer at a time.

1) Protocol Filtering: In the general case, all observed software protocols are those of all layers of the hierarchy. Let us consider for instance layer N. To check the layer N protocol, the observer must receive the corresponding N-layer protocol data units, the N-PDU's. For the messages to be received at layer N, the observer has to perform a set of lower layer functions, i.e., the functions of layers 1 to N which are needed in order to build the considered N-PDU's from the physical (layer 1) PDU's. The observer has to capture the physical messages, the physical PDU's, and derive from them the N-PDU's, as indicated in Fig. 4 [24]. Note that in the case where a broadcast service is available at layer K, with $K < N$, the observer has to restore the N-PDU's from the K-PDU's.

In order to build upper layer PDU's from physical messages, a basic filtering set of functions, named (N)-F, to be associated to each layer, from K to N, has been identified. Let us note M one of these layers. The set of functions is derived from the general model of layer M by considering all functions related to the up-down information transfer between two ISO adjacent layers [Fig. 4(a)].

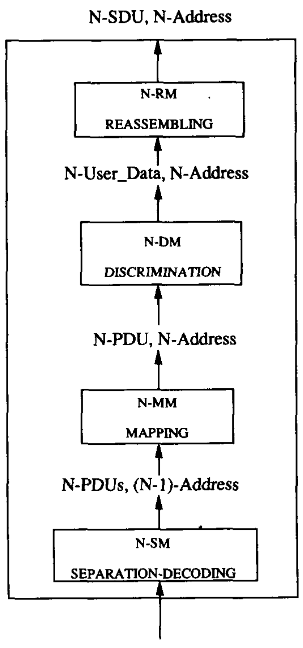
These functions are:

—concatenation and separation, which perform the mapping between $(M - 1)$-SDUs and (M)-PDU's

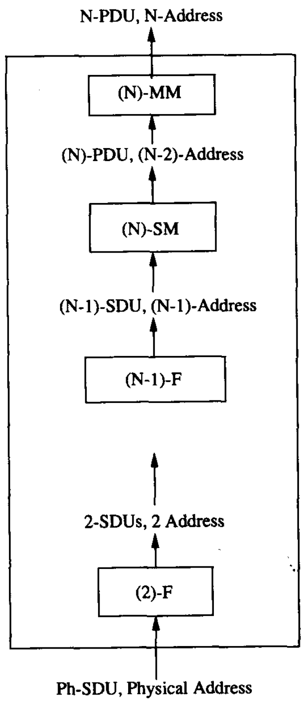
—encoding and decoding, which translate (M)-PDU's from their external representation into the internal representation of layer M and conversely;

—mapping of connections or associations, between layer $(M-1)$ and layer (M), using SAP and connection identifiers as source and destination references;

—segmenting and reassembling, which implement the mapping between the data of the (M)-PDU's and of the (M)-USER-DATA;



(a)



(b)

Fig. 4. Protocol filtering for layer N. (a) From layer $(N-1)$ to layer N. (b) From layer 1 to layer N.

—protocol control of the exchange of data between peer (M)-entities and between adjacent layers; this function includes the resequencing, the retransmission of transferred data PDU's if needed, and the delivery to layer $(M+1)$ of the valid data contained in (M)-PDU's.

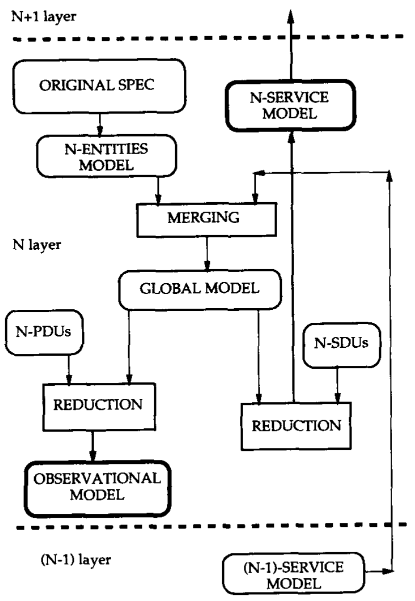


Fig. 5. Methodology for deriving observational models.

As the observer only receives messages and do not send data, the basic filtering functions (M)-F, associated with layer M, are derived from the layer M functions that are required only for receiving data.

Hence (M)-F is built up by elementary functions which are:

—(M)-SM: separating and decoding machine for layer M; these two functions are joined functions as separation of PDU's is an implicit result of decoding;

—(M)-MM: mapping machine for layer M, handling the association between $(M-1)$ addresses and (M) addresses;

—(M)-DM: discrimination machine, the reduction of the M protocol control function to a simple mechanism that orders the sequence of valid data from the received data PDU's;

—(M)-RM: reassembling machine for layer M which detects complete sequences of data segments and builds (M) SDUs.

The main feature of these functions is that except for (M)-SM all functions are protocol independent; this is because the (M)-SM function translates protocol specific external representations into common internal representations.

Finally, protocol filtering capability for layer N is obtained by layering the basic filtering functions (M)-F from the broadcast service, for instance layer 2 in a bus architecture, to layer M as shown in Fig. 4(b). This design allows the observer to receive, for any layer M between 2 and N, all M-PDU's derived from the physical PDU's.

2) Modeling Technique: As the observer can then validate any of the M-protocols, it needs as an input an observable model of these protocols, i.e., of the exchanged PDU's. It is proposed to obtain the formal observational model for one layer using the following three main steps methodology illustrated in the first part of Fig. 5 [11], [13], [22], [23]:

S-1) At any layer, each distinct protocol entity is represented by a labelled Petri net deduced from the protocol specifications (finite state machines are a specific case of Petri nets). Hence a hierarchy of local models, one for each protocol entity, is obtained. The corresponding descriptions, i.e., the descriptions of the entities, are simplifications of the complete behavior of the entities, these simplification being based on a set of relevant observable mechanisms. For instance, the establishment and release of the connections could be part of the model, as any designer could find relevant to validate and control them.

S-2) The models of the peer entities are connected together using a Petri net description of the service provided by the adjacent lower layer, i.e., their abstract communication medium. Note that a model of the underlying service has to be provided to the designer. Hence a global model of the layer is constructed. The correctness of the global model, i.e., the model that has been build by connecting together the local models and the service model, can be checked using exhaustive simulation or formal verification, e.g., by checking boundedness, liveness and proving temporal logic properties [11], [13], [14]. For instance, these sets of properties can be verified using appropriate software tools as PIPN [31].

S-3) After having been shown correct, the global models of the interacting entities are defined to be the data to be given as input to the observer. For instance, the model of layer M will be used to control the valid on-line behavior, i.e., the validity of the actual sequencing of the M-PDU's. Let us now consider the relationships that exist between the model and the actual behavior of the protocol on a LAN.

While listening to the broadcast medium, the observer is able to receive all events transmitted on this media and can derive all needed M-PDU's. The problem is that the observer is not aware of the events that are internal to any processor. Now, the formal descriptions that are built in the general case use all needed events, including the internal ones. Starting from the global model including all needed events, the set of events will have to be partitioned into two subsets, the subset of events internal to the processors, and the subset of events that are sent on the media. The latter set then defines for the observer the set of the observable events: this set can include all the PDU's that are exchanged between the communicating entities on the bus and does not include all primitives and exchange of information that are local to the processors.

Definition 7: Observable events in broadcast systems are those which lead to a partial or complete message sent on the broadcast service. Note that this defines "sending messages" events.

a) Deriving the observational model from the global model and the set of observable events implies keeping the names of the observable events and labelling as don't care, on the global model, the transitions which represent the nonobservable events, i.e., the events not visible by the observer;

b) Of course, the complexity of the corresponding models depends on the don't care transitions and on the set of mechanisms that have been selected. When a model is too complex, it is proposed to use reduced observable models, which are equivalent and simpler descriptions. Finding simpler models equivalent to the global one means keeping only the set of the observable transitions and so eliminating as much as possible the transitions that are marked as don't care transitions.

Let us note that for unconfirmed services, sent messages are not always received. The observer will be able to notice their losts by noticing that the new message (received after the lost one) is not the expected message and corresponds to one following the lost one, and so does not belong to the normal behavior.

3) Observational Models: Two alternatives exist for implementing the selected model.

Case 1: Implementing the observational model means simulating the behavior of the global Petri net. Starting from the initial state, i.e., the initial marking, the model has to evolve by firing the firable and not observable transitions until the model reaches a state where an observable event (the sending of a message) is enabled (is firable, in terms of Petri nets). Then the firing mechanism of the Petri net is stopped before firing this transition and the model simulator waits until:

a) the corresponding observable event occurs, i.e., the corresponding message is received by the observer. When this event occurs the model simulator fires this transition and subsequently starts again firing the internal transitions until another new observable event is reached, where the firing again waits for this new event, and so on.

b) an observable event occurs which is not expected, i.e., does not correspond to a PDU labelling one of the firable transitions of the model. This means that the worker and its model are not in agreement: an error is being detected and all present and past needed state information are saved. There is a discrepancy between the specification and the actual behavior and predefined signatures will be delivered to the users and designers.

Case 2: The observable model is too complex. Then, it can be simplified by deleting from it as many unuseful events as possible, i.e., as many nonobservable events as possible. This can be done by deriving a reduced model, equivalent to the global one, but containing a minimum number of unobservable events.

Deriving an observable model means deriving a model that is, in some sense, equivalent to the global one, and is a simplification of it. A lot of equivalence relations exist [32]. Trace equivalence [33], [34] and observational equivalence [35] are the main proposals, defined on labelled transition systems. They are defined by selecting specific observable events amongst the set of all events.

Definition 8: Two labelled transition systems P and Q are trace equivalent when they have the same set of traces (when $\text{Traces}(P) = \text{Traces}(Q)$). Two labelled transition systems P and Q are observational equivalent when they are related by a bisimulation [35].

When the observer receives messages from the bus or from the broadcast service as any normal user, it receives the messages one after the other. If one message at a time is received for conducting the observation, then the observer receives a flow of words belonging to a language, the language of the PDU's that are sent on the bus, i.e., that are generated by the observable events [17].

Many equivalence relations can be selected [32]. From the previous reasons, the one that has been selected for deriving the observational models is the equivalence with respect to sequential traces, or trace-equivalence. This is because the observer receives the messages from the broadcast medium one after the other, in sequence. The observation becomes equivalent to checking the correctness of the traces on the bus that represent the run-time behavior of the distributed system implementation.

The following procedure can be used to derive the simple observational model (second part of Fig. 5):

—describe and validate the global model of the protocol,

—the marking graph, the labelled reachability graph, of the global Petri net model is defined,

—all nonobservable transitions in this graph are labelled by a specific word, for instance the empty word,

—the minimal state machine, which generates the same traces for the set of all visible events, equivalent to this one, is derived.

Note that this procedure can first lead to a nondeterministic state machine. If so, the nondeterministic state machine has to be translated into an equivalent deterministic machine, as there always exists a deterministic machine which is equivalent to a nondeterministic one [33].

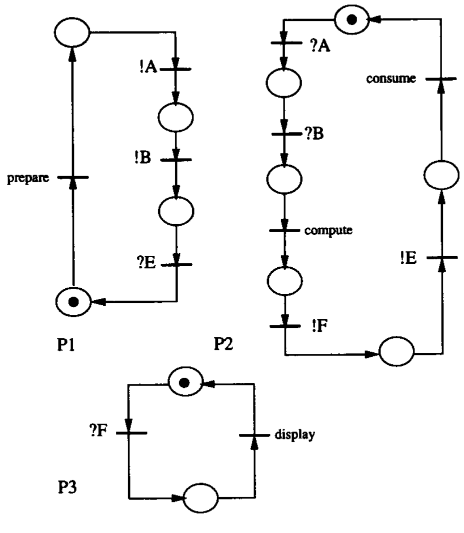
As an example, in Fig. 6(a), 3 processes are defined; they communicate by exchanging messages A, B, F and E. Let us assume that they use a very simple service, a datagram without loss, where every sent message is received. Fig. 6(b) is a Petri net model of such an ideal medium. The interconnection of the 3 processes by the selected medium leads to the Petri net of Fig. 6(c). This global model can be analyzed by adequate techniques such as reachability graph and invariants [14]. The model of Fig. 6(c) can be proved correct.

From Fig. 6(c), applying the simplification procedure leads to the observational state machine given in Fig. 6(d). This machine makes clear that, in this simple example, the observer has to control the sequential sending of messages A, B, F and E. Any other order between these four messages reveals an incorrect behavior of the system.

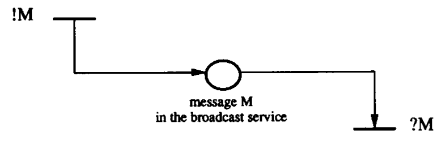
As the observable events are the exchanged messages, i.e., the Protocol Data Units, the global corresponding methodological approach for checking the behavior of protocols is given in Fig. 5. Note there that the same basic approach could be used to derive the model of a service, if a service could be defined by projecting on the set of the Service primitives, that are observable events.

4) Handling Complex Models: It follows from the previous arguments that the observer needs such an observable model for each observed protocol.

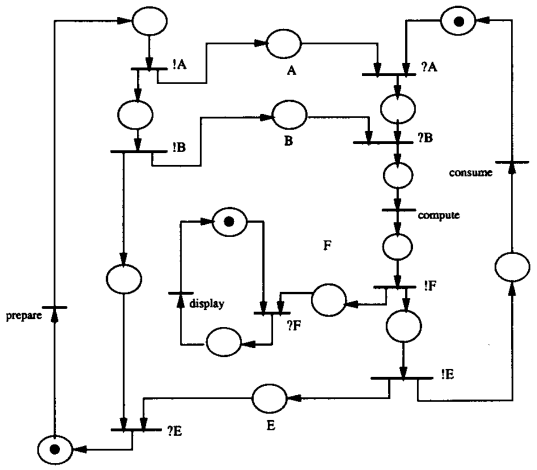
Theoretically, one protocol of one OSI layer can be represented by one model. Nevertheless, in practice, it can be difficult to describe many mechanisms inside one layer by a single model. Furthermore, complex and complete protocol behaviors can hardly be represented by only one Petri net. In such a case, the protocol description can be handled by parts, as in [25]. For instance, in [25], the connection-establishment, connection-release and data transfer phases of the Transport



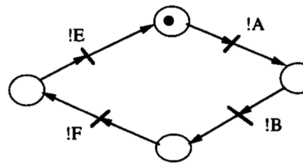
(a)



(b)



(c)



(d)

Fig. 6. An example of the design steps. (a) The communicating processes. (b) The service model. (c) The global model to be verified. (d) The projected observational model.

protocol have been modelled separately. Also, the Session protocol has been partitioned into its functional units.

In OSI systems, the behavior of one layer will be considered as a set of connections between protocol entities. Let one N-connection be called a dialogue within layer N. All dialogues run the same protocol and only one Petri net based model, representing one dialogue, needs to be used to represent the layer. Nevertheless, as there are many concurrent dialogues, the observer includes dynamic process management to concurrently run as many dialogues as needed using the same Petri net model: one process, simulating one Petri net, represents one dialogue. There will be as many concurrently simulated Petri nets as there are concurrent dialogues.

The creation and release of the dialogues are detected on line by the observer by analyzing the exchanged PDU's. For instance:

—at the Data Link layer a dialogue may be created only once and never deleted if a connectionless protocol is used;
—at the Transport layer, each dialogue corresponds to one transport connection: the beginning and the end of a dialogue in this layer are deduced by the observer from the reception of the connection-request and disconnection-request TPDU's.

C. Functions of the Observer Tool

Basically, an observer tool is able to support three main functions: audit-trail, run-time checking and performance analysis.

Audit-trail function simply delivers all PDU's exchanged- between user selected entities at a selected communication layer- as obtained after protocol filtering. Some additional facilities as pattern matching, triggering, time stamping and mass storing of results are possible. This is a rather classical protocol analysis function.

Run-time checking is realized on the basis of the observer principle with respect to some selected models representing a set of given layers. The observer is simply adapted to new protocols by modifying the observable model of the new protocols.

Performance analysis facility may be added as it consists in measuring time instants, i.e., times elapsed between typical protocol PDU's. Of course, these PDU events are the ones which appears in the Petri net model when they are able to characterize and provide significant performance figures, such as the transfer throughput of the layer. Each PDU received by the observer is associated with some time value. The difference between the reception times of adequate PDU's gives the actual duration of protocol phases. Throughput is for instance computed from the ratio of the number of data transfered PDU's to the duration of the data transfer phase.

The original feature of this performance measurement technique is that the beginning and the end of each measurement is performed on a Petri net driven scheme. Then performance analysis becomes a formally defined facility and furthermore uses the same model (and tool) than the checking facility. Moreover the Petri net based model used for performance analysis is only slightly extended with respect to the validation Petri net model: to each transition of the Petri net is associated a simple performance command. When a performance related transition is fired by the observer, the performance command is executed. Performance related commands contain one of the following actions:

—store the reception time of the current message,
—calculate the difference between a previously stored time and the reception time of the current message,
—increment the number of transfered data PDU's,
—etc.

The execution of these actions directly follows from the Petri net model, and protocol performance can be formally defined by defining the events that are labels of the Petri net.

III. Examples of Application

This section gives three applications of the observer for on-line debugging, behavioral testing and performance measurements. The first tool was used in REBUS, an experimental LAN for real time control, for debugging and on-line checking a token bus protocol. The second example presents a prototype related to an open LAN for industrial applications [16]; it allowed the testing and the performance evaluation figures of a Link and a Transport layers. The last tool has been applied to a multilayered system, including the layers 2 to 6 of the ISO protocols.

A. REBUS

The observer concept has been developed to support the implementation of REBUS [18]. Some features of the worker, of the observer, and some experimental results are given below.

1) The Worker: The worker is the local area network implementation. REBUS was an experimental system developed for designing fault tolerant bus access software. The hardware consisted of two (duplicated) serial busses as the physical media and of Serial Bus Interface (SBI) processors where the communication software resided.

The mechanism which has been selected and actually checked by the observer, because of its importance, is the MAC (Media Access Control) protocol, a fault tolerant virtual ring protocol. This protocol is based on a token passing scheme, so token passing was on-line tested.

2) The Observed Protocol: The Fault-Tolerant Virtual Ring-for Bus Allocation: The token passing protocol is based on a (programmable) "privilege" circulating on a virtual ring. One unit at a time becomes primary, i.e., is able to access the bus, whereas the other units are secondary and are allowed to send only response messages to the primary unit. At the end of its privilege, the primary unit sends the token (the primary status) to its successor on the virtual ring. The virtual ring is a logical organization, ordered in a circular fashion, of the interface units (SBI). This ordering is virtual as the token must follow the ring; functional messages sent by the units may be transmitted to any unit and no particular organization of the communication media is implied.

Let I be the primary unit. It keeps the privilege for a limited amount of time. When time elapses or when it has no more

TABLE I
FIRST EXPERIMENT

TYPE OF EVENT	PERCENTAGE
message primary status	99.6
message become primary	0.2
successful recovery	0.173
incorrect recovery	0.003
unsuccessful recovery	0.024
duplication of master	0

TABLE II
SECOND EXPERIMENT

TYPE OF EVENT	PERCENTAGE	
	PHASE A	PHASE B
message primary status	99.9992	99.995
message become primary	4.10-4	8.10-4
successful recovery	4.10-4	6.10-4
incorrect recovery	0	28.10-4
unsuccessful recovery	0	1.10-4
duplication of master	0	1.10-4

message to send, it relinquishes the primary status by sending the broadcast message "primary_status_(S(*I*))" -where $S(I)$ means Successor of unit I on the virtual ring-. It then loses its privilege and becomes secondary. By receiving this message the unit successor of I, $S(I)$, becomes primary.

If the unit successor of $S(I)$, $S(S(I))$, detects that the primary status has not been normally transmitted from unit I to unit $S(I)$ after a given time limit—for instance if unit I is dumb-then unit $S(S(I))$ sends a recovery message "become primary" to unit $S(I)$. When receiving this message, unit $S(I)$ becomes the new primary, if it has not previously received the message "primary status ($S(I)$)" or it simply ignores the message "become primary", if it has already received the primary status-note that in this latter case, $S(S(I))$ is faulty.

The model and the validation of this protocol appears in [18]. It has been formally shown that the protocol satisfies the following properties:

—mutual exclusion: at any time, only one of the SBI's has the possibility of getting the access to the bus, i.e., of being primary,

—robustness: this primary status can be temporarily lost but will be recovered within a finite time,

—fairness: every nonfaulty SBI processor will in turn receive the primary status.

3) The Observer: The observer has been developed on an INTEL processor connected to an SBI board as the other functional processors (see Fig. 3).

The kernel of the observer software is a simple Petri net simulator. The tool contains general purpose functions such as:

—man-machine dialogue for on-line diagnosis and control display,

—statistical processing and storage of the received messages.

4) Experimental Results: Two sets of results are provided, the first one collected during the debugging phase, early in the implementation process, and the second one collected much later, during normal behavior, when the system was released.

a) First Results: The first results (Table I) showed some frequent existing malfunctioning, while the corresponding faults were recovered and not visible for the users, as the service provided to the users was correct. The on-line detections of the observer were very useful because if they were not detected, the accumulation of these not visible faults could have later induced errors on the external behavior, i.e., on the provided service.

Unsuccessful recoveries resulted from the losses or the nonacceptation of the recovery messages (the "become primary" messages) which have been always followed by successful recoveries and found to be nonmalignant.

Incorrect recoveries revealed a violation due to an erroneous recovery attempt by the $S(I)$ after a token passing which was undetected by $S(I)$. The examination of the observed traces showed an intermittent malfunctioning of one SBI.

b) Second Results: For the second set of experiments, the protocol behavior has been correct until the bus primary status became once duplicated. The results are given in Table II: phase A consists of the results collected from the beginning of the second set till the detection of this error, and phase B gives the results related to the error.

Checking the observer outputs showed that the origin of the error was a particular unit. The corresponding analysis revealed an incorrect memory operation: the unit that was the current primary was stopped after executing a "Halt" instruction, which did not exist in the actual program, and resulted into an incorrect read operation. When the execution on this unit was suspended due to the Halt instruction, the primary status was recovered by the other units; then a hardware interrupt resumed execution of the faulty unit. Duplication appeared between this unit which continued to act as a primary while the other recovered unit had become primary as a consequence of the recovery mechanism. This example should clearly show how the observer helped to detect this nontrivial hardware error.

B. Application to Layered Systems

This section describes the design of a debugging and testing prototype tool for an industrial LAN and gives the observer which has been developed for checking a complete layered architecture of an office automation open system based on Ethernet, including layers 1 to 6 of the OSI Reference model.

1) Industrial LAN: The LAN is a layered open system architecture, developed for real time control, and includes the Data Link and Transport layer protocols. The observer has been build to separately check the two main protocol layers, Data Link and Transport, and to offer performance analysis facility derived from the Petri net observational model [16].

The LAN has a double coaxial cable bus with identical cable interface units (CIU). The communication software is implemented on the CIU and up to four users can be connected to each CIU.

At the MAC sublayer of Data Link layer, a CSMA/CD-like priority based mechanism manages the medium access. The LLC (logical link control) sublayer of data link layer consists of a connectionless protocol. At the transport layer a subset of the ISO transport protocol and service were implemented.

a) Observable Model of the Transport Layer: The observable model of the transport layer consists of two basic observable models. Resulting from the modelling technique described before, they represent the connection establishment phase, the release phase and the data transfer phase.

Sending each PDU of the connection establishment and release phases (CR, CC, DR, DC) was represented by a distinct transition; sending all data transfer PDU's (DT, AK, FREE, AK_AND_FREE) was represented by one transition labelled by "Initiator ! DT_PDU" for the initiator (resp. "Responder ! DT_PDU" for the responder).

Finally the model of the connection establishment-release phases and the model of the data transfer phase were connected together by merging each transition of the data transfer phase model which are labelled by "Initiator ! PDU$_i$" (resp. "Responder ! PDUi") with the transition of the connection establishment-release phases model labelled by "Initiator ! DT_PDU" (resp. "Responder ! DT_PDU").

b) The Observer: For multilevel communication systems, it appeared difficult to directly implement the observer concept as stated. It was decided to implement both observations (Data link and Transport) but only one at a time: observing the system means observing alternatively one of the two layers.

As seen before, the observer required protocol filtering in order to restore the Transport PDU's from the information directly available on the physical medium. The layered behavior of the system is considered as a set of dialogues between protocol entities. Each N-dialogue corresponds to one N-connection and all dialogues follow the same protocol. Hence, only one formal model representing one dialogue represents the whole layer.

c) Protocol Filtering: For observing the LLC sublayer, LPDU's are restored by executing the same delimiting, synchronisation and error detection functions as in the worker, because the observer supports the same sublayers as other CIU's.

At the Transport layer, interactions exchanged in the system, i.e., TPDU's, were restored from LPDU's by:

—recombining each Transport connection splitted on two Data Link connections by setting up the mapping between these two layers,

—detecting and discarding the duplications inherent to the Data Link layer; thus, when a data transfer LPDU is retransmitted by the Data Link layer, only the data (TPDU) conveyed by the first LPDU must be considered and the data carried by retransmitted LPDU's must be discarded.

The Petri net representing all dialogues at one layer was a re-entrant program, where several identical processes corresponded to the several actual dialogues taking place in the system.

At the Data Link layer each dialogue is created only once and never deleted, because a connectionless protocol is used. At the Transport layer each dialogue corresponds to one Transport connection.

The creation and the deletion of dialogues are detected through the corresponding exchanged PDU's.

d) Performance Analysis Facility: This function measured the time elapsed between typical protocol events and computed the data transfer throughput for each layer.

Each Link or Transport PDU received by the observer was associated with a value of a timer. The difference between the reception times gave the duration of the phases and the throughput was derived from the ratio of the number of transferred data PDU's to the duration of the data transfer phase.

The original feature of this technique is that the detection of the beginning and the end of each phase is performed on a Petri net driven scheme. Performance analysis facility uses the same structure including protocol filtering, dynamic process management and Petri net simulation than the checking function. Hence, the execution of these actions in the Petri net model provides the formal performance analysis facility.

e) The Implementation of the Observer: The hardware support of the observer is a standard CIU connected to the shared communication media. The tasks performing checking and performance analysis functions are implemented in the CIU and supported by a multitasking executive. Other functions are human interface and message storage.

2) Office Automation Open System: This second connection oriented system consisted of an Ethernet LAN linking together terminals and servers connected to the outside world through a PABX interface or a packet switching interface. This network included two server stations (messages and archives) and allowed to access the X25 TRANSPAC network through a gateway.

The architecture was organized according to the OSI model. The implemented protocols in the different layers were:

Session:	ISO Session protocol
Transport:	ISO Transport protocol (Class 0 and 2)
Network:	X25 protocol
Data link:	IEEE 802.2 protocol (type 1 and 2)

For example, two simplified observable models derived from a Petri net model representing the Connection Establishment and Release phases of X25 and IEEE 802.2 type 2 (for the perfect case, i.e., without loosing PDU's) are given in Fig. 7(a) and (b).

The following comments can be given. Only left hand parts of the figures are considered as right parts are symmetric (changing A and B roles).

a) IEEE 802.2 type 2 [Fig. 7(a)]

— Path 1-3-5 represents the connection establishment initiated by A (A!SABM) and accepted by B (B!UA); note that the transition from state 3 to 4 expresses the refusal of B (B!DM);

— Path 1-3-6-8-5 gives the scenario related to the connection establishment crossing;

— Path 5-9-4 and path 5-11-12-4 represent the disconnection scenarios initiated by A, respectively in the case of acceptance by B and in the case of disconnection crossing.

b) X25 protocol [Fig. 7(b)]

- Path P1-P3-P5 gives the scenario of the connection establishment initiated by A (by A!Connect Request) and accepted by B (B!Connect Confirm);
- Path P1-P3-P7-P5 represents the collision problem where A has initiated a connection establishment and receives a connection establishment request from B; according to the highest priority (A or B), the transition from P7 to P5 can be either B!Connect Confirm or A!Connect Confirm;
- Path P1-P3-P4 shows an unsuccessful connection establishment initiated by A where B refuses the connection by B!Clear Request and A accepts this refusal by A!Clear Confirm ; note that the transition from P4 to P1, A!Clear Request, corresponds to the case where the disconnection is initiated by A in parallel with receiving the refusal from B;
- Transition from P5 to P12 starts, after the connection establishment, the normal disconnection phase requested by A; the scenarios starting from States P3 and P7, leading to States P6 and P13 model the disconnection requested by A, whereas A has not yet received the answer to its connection establishment request.

Note that in addition to communications internal to the LAN, communications with entities external to the LAN using X25 through the local network can be observed for layers from Network to Session. So, the general case of protocol filtering has been implemented.

The observer has been run on a UNIX processor [24] consisting of one Ethernet interface unit, one processing unit and one graphic interface unit. It included all the models [25] of the 4 layers, from Data Link to Session, given before. It allowed audit trial, checking and performance measurements for all layers.

IV. Conclusion

From these experiments, two important aspects can be emphasized. First, the observer proved to be quite useful for debugging communication software during the development phase; second, even when debugging was considered complete, the observer still had been quite helpful for detecting malfunctioning coming from hardware faults or software errors that occurred at run time. It also appeared that the observer concept is fully consistent with the formal methodologies presently defined for formally designing distributed systems. The observer is also of interest for measurements and maintenance during the system life cycle.

Also some subtle aspects have to be developed further, such as the use of more complex and more complete observable models. Also, the observer recovery, (when the observer itself loses messages), as well as the definition of sets of multiple cooperating observers, for no broadcast services, still need to be investigated.

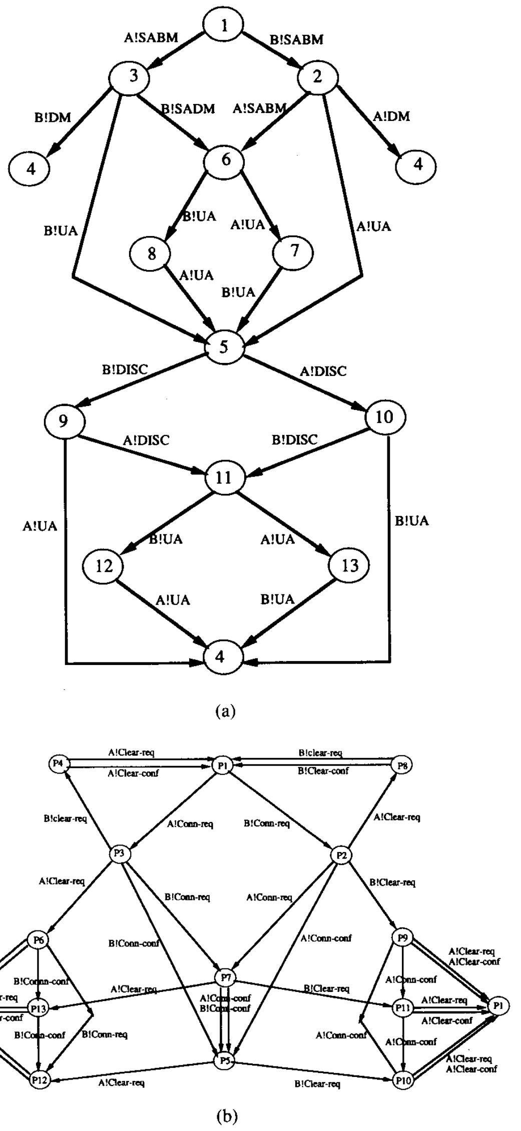


Fig. 7. Observers of the IEEE 802.2 and X25 Connection Establishment and Release phases.

Acknowledgment

The authors would like to thank J. M. Ayache and R. Molva who were greatly involved in the developments and implementations presented in this paper, as well as all other people who were involved in this work.

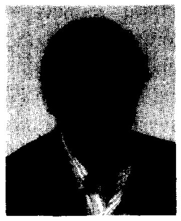
References

[1] J. M. Ayache, P. Azema, and M. Diaz, "Observer, a concept for on line detection for control errors in concurrent systems," in *9th Int. Symp. FTC*, Madison, June 1979.

[2] J. M. Ayache *et al.*, "Software redundancy for error detection in distributed systems," *Congrés Fiabilité et Maintenabilité*, Toulouse, Sep. 1982.

[3] G. Lamarche and P. Tallibert, "IDA: Software test language and associated tools," in *1st Colloque Génie Logiciel*, June 1982.

[4] D. L. Parnas, "The use of precise specifications in the development of software," *Proc. IFIP*, 1977, pp. 861, 867.
[5] D. Rayner, "Towards standardised OSI conformance tests," *Protocol Specification, Testing and Validation*, V. Diaz, Ed. Amsterdam, Netherlands: North Holland, 1986.
[6] B. Pradin *et al.*, "OGIVE: un outil graphique interactif de vérification des systèmes parallèles décrits par des réseaux de Petri," *Revue MICADO* Sep. 1980.
[7] H. Zimmermann, "OSI reference model—The ISO model of architecture for open systems interconnection," *IEEE Trans. Commun.*, vol. COM-28, pp. 651–660, Apr. 1980.
[8] M. A. Fischler, O. Firchtein, and D. L. Drew, "Distinct software: an approach to reliable computing," in *2nd USA-Japan Comp. Conf.*, pp. 27-4-1, 27-4-7.
[9] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, I:2, pp. 220–232, June 1975.
[10] J. R. Kane and S. S. Yau, "Concurrent software fault detection," *IEEE Trans. Software Eng.*, I:1, pp. 87–99, Mar. 1975.
[11] M. Diaz, "Specification and validation of communication and cooperation protocols using Petri net based models," *Computer Networks*, Dec. 1982.
[12] G. V. Bochmann and C. A. Sunshine, "Formal methods in communication protocol design," *IEEE Trans. Commun.*, vol. COM-28, no. 4, pp. 624–631, Apr. 1980.
[13] A. S. Danthine, "Protocol representation with finite-state models," *IEEE Trans. Commun.*, vol. COM-28, no. 4, pp. 632–643.
[14] H. J. Genrich and K. Lautenbach, "The analysis of distributed systems by means of predicate/transition nets," in *Semantics of Concurrent Computation*, G. Kahn Ed. New York: Springer-Verlag, 1979, pp. 123–146; also in Lect. Notes in Computer Sciences, vol. 70.
[15] G. V. Bochmann, "A general transition model for protocols and communication services," *IEEE Trans. Commun.*, vol. COM-28, no. 4, pp. 643–650, Apr. 1980.
[16] Manuel de présentation de FACTOR, APTOR (1984).
[17] J. M. Ayache, J. P. Courtiat, and M. Diaz, "Self-checking software in distributed systems," *3rd Int. Conf. Distrib. Comput. Syst. IEEE*, Oct. 1983.
[18] J. M. Ayache, J. P. Courtiat, and M. Diaz, "REBUS, a fault-tolerant distributed system for industrial real-time control," *IEEE Trans. Comput., Special Issue on Fault Tolerant Computing*, vol. 31, no. 7, Jul. 1982.
[19] P. Azema *et al.*, "Specification and validation of distributed systems using PROLOG interpreted Petri nets—PIPN," *7th Int. Conf. Software Eng.*, Orlando, FL, Mar. 1984.
[20] ISO, "Estelle, a formal description technique based on an extended state transition model," *ISO-TC97-SC21-WG1, DIS 9074.*
[21] ISO, "LOTOS, a formal description technique based on an extended state transition model," *ISO-TC97-SC21-WG1, DIS 8807.*
[22] G. Juanole and B. Algayres, "Protocol design and modeling," in *4th European Workshop on Application and Theory of Petri Nets*, Toulouse, Sept. 1983.
[23] M. Diaz, "Petri net based models in the specification and verification of protocols," LNCS, no. 255, *Advances in Petri Nets*, E. Brauer *et al.*, Eds. New York: Springer-Verlag, 1987.
[24] R. Molva, "Conception et réalisation d'un observateur d'architectures multicouches," Thèse de Doctorat, Univ. Paul Sabatier, Toulouse, Oct. 1986.
[25] J. M. Novali, "Modèles d'observation pour les architectures multicouches," Thèse de Docteur-Ingénieur, Toulouse, INSA, Nov. 1986.
[26] D. A. Anderson and G. Metze, "Design of totally self-checking circuits for m-out-of-n codes," *IEEE Trans. Comput.*, vol. 22, Mar. 1973.
[27] M. Diaz, P. Azema, and J. M. Ayache, "Unified design of self-checking and fail safe combinational circuits and sequential machines," *IEEE Trans. Comput.*, vol. 28, no. 3, pp. 276–281, Mar. 1979.
[28] N. G. Levenson, "Safety analysis using Petri nets," *IEEE Trans. Software. Eng.*, vol. SE–13, pp. 386–397, Mar. 1987.
[29] F. Biairdi, N. D. Francesco, and G. Vaglini, "Development of a debugger for a concurrent language," *IEEE Trans. Software Eng.*, vol. SE–12, pp. 547–553, Apr. 1986.
[30] M. Diaz and C. Vissers, "SEDOS, Estelle and LOTOS environments for the design of open distributed systems," *IEEE Software Eng.*, vol. 15, Nov. 1989.
[31] J. C. Lloret and P. Azema, "Incremental verification of token ring protocol," in *6th Int. Workshop on Protocol Specification, Testing and Validation*, Montreal, June 1986.
[32] R. J. van Glabbeek, "The linear-branching time spectrum," in *Proc. CONCUR89*, LNCS 458, J. Baeten & J. Klop Eds. New York: Springer-Verlag, 1990, pp. 278–287.
[33] Z. Kohavi, "Switching and finite automata theory," *Computer Science series*. New York: McGraw-Hill, 1970.
[34] M. Hennessy and R. Milner, "Algebraic laws for nondeterminism and concurrency," *J. ACM*, vol. 32, no. 1, pp. 137–161, Jan. 1985.
[35] R. Milner, "Communication and Concurrency," *Int. Series in Computer Science*. Englewood Cliffs, NJ: Prentice-Hall, 1989.



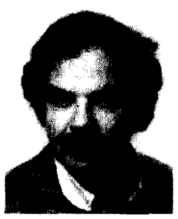
Michel Diaz (SM'92) is Directeur de Recherche at the Centre National de la Recherche Scientifique (CNRS) and leads the Research Group "Communications Softwares and Tools" at Laboratoire d'Automatique et d'Analyse des Systemes du C.N.R.S., Toulouse. He has been working on the development of formal methodologies, techniques and tools for designing distributed systems during the last ten years. In 1989 and 1990, he spent a year as visiting staff at the University of Delaware at Newark and at the University of California at Berkeley.

Dr. Diaz is a member of many Program Committees; he served as a Chairman of the Program Committee for the IFIP Congres on "Protocol Specification, Testing and Verification," the European Workshop on "Application and Theory of Petri Nets," and the International Conference on Distributed Computing Systems, in Area "Software Engineering", the IFIP congres FORTE on Formal Description Techniques. He is a Technical Editor for the journals *Reseaux et Informatique Repartie*, *Annales des Telecommunications*, and *Communications Magazine*. From 1984 to 1988, he was the prime manager of the SEDOS project (Software Environments for the Design of Open distributed Systems, in which the Formal Techniques Estelle and LOTOS have been developed) within the ESPRIT programme of the CEC. He has written one book and more than 100 technical publications. He is the editor of the North Holland volume on *Protocol Specification, Testing and Verification*, 1985, co-editor of two North Holland volumes dedicated respectively to the *Formal Description Techniques Estelle and LOTOS*, 1990, and co-editor of the North Holland volume on *Formal Description Techniques*, 1992. He has received the Silver Core of the IFIP and is listed in the *Who's Who in Science and Engineering*. He is presently Director of the French Research Coordination Group on "Parallelism, Networks and Systems" (GDR "Parallelisme, Reseaux et Systemes) and the co-head of the French CNET-CNRS project CESAME on the formal design of high speed multimedia cooperative systems.

Guy Juanole received the "Doctor es Sciences" degree from the University Paul Sabatier, Toulouse, France, in 1978.

He is, at the present time, Professor at the University Paul Sabatier (lectures in Automatic Control, Computer Communication Networks, Formal Models for the Analysis of Distributed Systems (Petri Nets, Queuing Networks, Stochastic Petri Nets) and researcher at the Laboratory LAAS (Laboratoire d'Architecture et d'Analyse des Systemes) of the CNRS (Centre National de la Recherche Scientifique) in Toulouse. His research interests are in the area of the Petri nets based models with a major emphasis on Timed and Stochastic Petri nets and their applications to communication networks and protocols (high speed networks - industrial networks).

Dr. Juanole has been Program Co-Chairman of the 5th International Workshop on Petri Nets and Performance Models.



Jean-Pierre Courtiat (M'88) graduated in computer science from ENSEEIHT in 1973. He received the Ph.D. and Doctorat d'Etat degrees in computer science from the University of Toulouse, France, in 1976 and 1986, respectively.

After having been a researcher at LAAS/CNRS from 1973 to 1976, he has been an expert of the French technical cooperation from 1976 to 1980 with an appointment at the Federal University of Rio de Janeiro, as Professor of Computer Science. In 1980, he came back to LAAS, as "charge de recherche au CNRS" (a research position of the French National Council of Scientific Research). At LAAS, he works in the OLC (Software and Communication) research group, where he leads the SFP (Formal Specification of Protocols) research team. His research interests include the design of protocols, as well as the definition and application of formal methods for the specification, verification and testing of protocols and distributed systems, areas in which he has authored or co-authored more than 70 international publications. Currently, he participates to several researches dealing with the semantics of concurrency and the expression of time-constraints in the formal description techniques, as well as the application of these techniques to the formal design of co-operative high speed multimedia distributed systems. In carrying out these researches, he has taken several responsibilities in different European research projects, and has participated to standardization activities, as an expert of AFNOR and ISO. He is currently one of the co-managers of CESAME, a CNET-CNRS collaborative project on High Speed Multimedia Systems sponsored by France-Telecom.

Dr. Courtiat is a member of ACM.