An Approach to Testing Specifications

Claude Jard

CNET Lannion, France

Gregor v. Bochmann

University of Montreal

An approach to testing the consistency of specifications is explored, which is applicable to the design validation of communication protocols and other cases of step-wise refinement. In this approach, a testing module compares a trace of interactions obtained from an execution of the refined specification (e.g., the protocol specification) with the reference specification (e.g., the communication service specification). Nondeterminism in reference specifications presents certain problems. Using an extended finite state transition model for the specifications, a strategy for limiting the amount of nondeterminacy is presented. An automated method for constructing a testing module for a given reference specification is discussed. Experience with the application of this testing approach to the design of a transport protocol and a distributed mutual exclusion algorithm is described.

1. INTRODUCTION

During the design of a computer system or some application software, the validation of the design is an important activity. The purpose of validation is to make sure that the different specifications of the computer or application system are consistent. Usually, specifications are given at different levels of abstraction. The overall system specifications represent the highest level of description. During the design and implementation of the system, different specifications with more details are subsequently elaborated, such as a functional decomposition of the system into several system parts, or the steps that lead to the implementation of these parts in a programming language or in hardware. The validation activity should detect any errors in these different descriptions, and check that they are consistent with one another. In particular, the system implementation should be consistent with the original system specification.

There seem to be essentially two approaches to validation:

1. *Verification,* which analyses the specifications by logical means. This may take the form of analytical performance prediction, program "correctness" proofs, symbolic execution, or the proof of system properties based on the given specifications.
2. *Testing,* which explores a large number of the possible execution histories of the system, and compares the observed behavior with the given specification. This may take the form of simulation studies, traditional testing procedures, or the execution of specifications [10] to test their consistency.

This paper explores a validation approach which uses high-level specifications as a reference in respect to which lower-level specifications, or implementations may be tested for consistency. While the method described is generally applicable to any system design which employs formal specifications, the main objective of our research was the use of such an approach for the design of communication protocols and services.

This paper describes the use of what we call a "trace checker." This is a module which observes the execution of a system under test (which may be a system implementation or an artificial execution of some refined system specifications) and compares its behavior with the formal specifications given for that system.

In Sec. 2, we describe the main application scenarios for the use of such a trace checker, and in Sec. 3, we explain the interaction model that underlies the method used for writing abstract or refined system specifica-

Address correspondence to C. Jard, Departement Evaluation de Protocoles, CNET Lannion, BP40, 22301 Lannion, France.

The Journal of Systems and Software 3, 315–323 (1983)

0164-1212/83/$3.00

tions. We think that this model is sufficiently general to allow the use of this approach in different application areas. Section 4 presents the design and implementation principles of a trace checker for a given reference specification, and discusses its limitations. In Sec. 5, the application of this testing approach to the design of communication protocols is discussed. The experience gained with using a trace tester is described.

2. SCENARIOS FOR USE OF A TRACE CHECKER

A. Testing a System Implementation in Respect to the Specifications

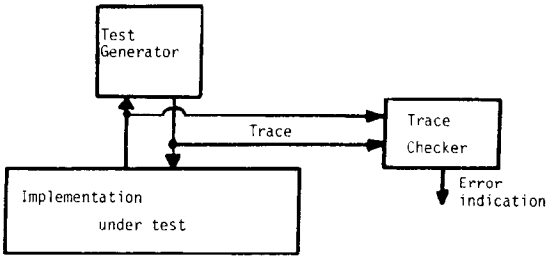
One way of using a trace checker is for the testing of an implementation. We assume that a formal specification is given for the behavior of the implemented system. The testing configuration is shown in Figure 1. The implementation under test is stimulated by some test input sequence and this input, as well as the output produced by the implementation, is observed by the trace checker. The latter checks that the observed sequence of events is a possible sequence according to the given specification.

Two modes of operation may be distinguished: on-line and off-line. The on-line mode has the advantage that in the case of a detected error, the execution of further test input can be halted and the state of the implementation under test can be investigated. It is to be noted, however, that the cause for the detected error may have occured much earlier in the observed execution history (error latency). This approach is similar to the worker–observer principle in [7]. A more detailed discussion of error detection is given in Section 4.

In the case of off-line operation, the observed trace of input and output interactions of the implementation under test is temporarily stored on some medium and later checked by the trace checker. This may be easier to realize than on-line testing, in certain cases.

In the area of communication protocols, this testing scenario could be used for the testing and certification of protocol implementation [1]. A given protocol implementation may be checked for conformance with the protocol specification, which is given by a formal description.

Figure 1. Testing a system implementation.



B. Step-Wise Refinement of Specifications

In the case that a design phase consists of a step-wise refinement of a more abstract specification into a more detailed one, a trace checker may be used to test the consistency of the refined specification in respect to the abstract one. It is assumed that it is possible to "execute" the refined specification. (See, e.g., [10] and the approach in [12]). Care must be taken to cover, as far as possible, all possible execution sequences that are allowed by the detailed specifications.

The testing configuration is similar to Figure 1. Instead of an "implementation under test," we have a "detailed specification under test," which consists presumably of several interacting submodules, the behaviors of which are defined by the specification. In the case of communication protocol design, this scenario would be used for protocol design validation [2], i.e., to check that the protocol specification of a given protocol layer N is consistent with the service specifications of that layer and the layer below. As shown in Figure 2, the behavior of the protocol will be observed by "executing" two protocol entities interacting through an entity representing the communication service of the layer below.

Test inputs are generated by appropriate user entities which use the communication service provided by the protocol. The observed trace of interactions with the user entities (inputs generated by the users and outputs generated by the protocol entities) are compared by the trace checker with the possible execution histories that are defined by the service specification for the same layer N.

It is important to note that the observed trace characterizes the global properties of the system layer, since interactions at the different service access points are considered.

3. THE CONTEXT OF A TRACE CHECKER

This section explains the context in which the trace checker described in this paper has been designed. This includes in particular the descriptive model used for the writing of formal specifications.

A. The Interaction Model

The descriptive model used by the trace checker is described in more detail in [3–6]. A "process" (also called entity, module, etc.) is characterized by a number of "ports" (also called interaction points, access points, interfaces, etc.) through which it interacts with other pro-

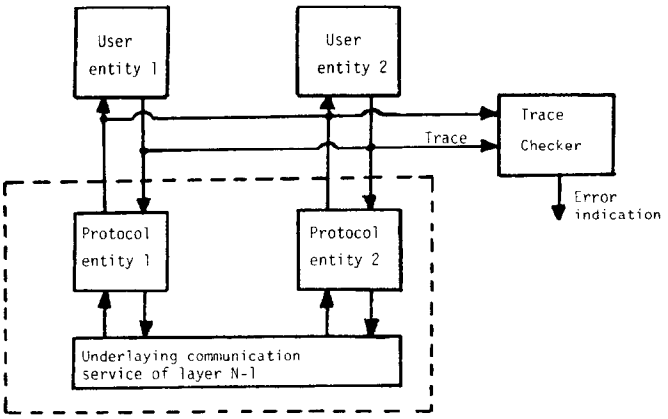


Figure 2. Testing a detailed specification.

cesses in its environment. At each port, certain types of interactions are possible. An interaction type is characterized by its name and a number of parameters. For each occurrence of an interaction, each parameter takes on a definite value within the range of possible values defined by a data type definition associated with the declaration of the interaction type. In addition, it is assumed that each interaction transfers information only in one direction; therefore the distinction between "input" and "output" interactions is made.

A "trace" (also called execution history, etc.) of a process is obtained by observing all interactions in which the process participates. Assuming that at a given port only one interaction can take place at any given time, the order of the interactions at a given port is defined for each trace. For also checking certain properties of the process the relative order of interactions occurring at different ports must be determined. In a distributed system, this may in general be a difficult problem. However, different practical methods can be chosen to obtain a total order of all observed interactions. For example, the interactions may be ordered in respect to their beginning in real time, where a global clock, or several synchronized clocks may be taken as time reference [14]. In the case of execution by simulation, the virtual simulation time may be chosen as a reference. In the case that simultaneous interactions are considered, several different traces are obtained depending on the arbitrary order into which the simultaneous interactions are put in the trace. We assume in the following that a satisfactory method is given for obtaining a trace of interactions from an observed execution of an implementation, or detailed specification.

B. An Extended State Transition Specification Model

The trace checker described in this paper is based on an extended state transition model for the description of the reference specification in respect to which the trace is compared. The model is an extension of a finite state machine. In addition to the major state, which defines the finite state part of the machine, a process description contains, in general, some additional state variables. Together with the major state and the parameters of an input interactions, they determine the transition to be performed by the machine. Each transition defined by a given specification is characterized by a condition that must be satisfied if the transition is to be executed, and a transition action which updates the state variables and possibly generates some output interactions. (For more detail, see [3, 11]).

For the purpose of the following exposition, we distinguish the following classes of transitions:

1. *Input transitions.* They are initiated by some input interaction, and may produce one or several output interactions.
2. *Spontaneous output transitions.* They are spontaneous transitions (i.e., they may be executed anytime provided that the machine is in an appropriate state) which produce one or several output interactions.
3. *Internal transitions.* They are spontaneous, and do not produce output.

C. Nondeterminism in Specifications

A specification is nondeterministic, as observed by its interaction sequences, if, for a particular state, there is a spontaneous transition, or for a given input interaction, there are more than one possible input transitions. In practice, nondeterminism is mostly due to spontaneous transitions which are usually introduced for one of the following reasons:

1. The description of failures of system components which may occur any time.
2. The description of a time-out mechanism, which activates a time-out transition after a given time period. Since most specifications are time independent,

the concept of a time period is not naturally described. Usually, a time-out transition is specified as a spontaneous transition that may execute any time (as long as the machine is in the "timer running" state, which may be changed by other transitions that occur).

3. The description of incompletely specified conditions, which may occur at certain times, such as "network congestion."
4. The description of parallelism, where the order in which two or more spontaneous transitions occur is not important.

It is clear that in the case of a nondeterministic reference specification, in general, several different traces are valid for a given sequence of input interactions provided to the system under test (because the same inputs can produce different sets of outputs). How the trace checker takes care of these different possibilities is discussed in Sects. 4A and 4B.

An example of a nondeterministic specification is given in Figure 3. It is part of a distributed mutual exclusion algorithm studied by the CNET [8]. This example will be used as an illustration throughout the paper. We consider first the state diagram for one site only.

1. Initially, the given site is in an inactive (FAILED) state, and becomes IDLE after a reset (re) interaction by the user.
2. The user may demand the resource (de) entering the CANDIDATE state; if the state was not IDLE, the service is refused (rs).
3. When the resource is free, the user may receive a reservation interaction (er) and enter the ELECT state; if he is not elected after a finite amount of time, an exclusion failure is indicated (ee).
4. After the use of the resource in the elected state, the user may release the resource (fe).
5. At any time, the site may crash and enter the FAILED state.

All the types of transitions appear in Figure 3: the failure transitions are internal and no indication is sent to the user, two transitions from the CANDIDATE state are spontaneous output transitions and indicate to the user the mutual exclusion reservation (er) or failure (ee), the others are input transitions.

D. Incomplete Specifications

It is often convenient to allow for incomplete specifications. A specification is incomplete if, for a particular state and a particular input interaction (with particular parameter values) no input transition is specified (i.e., the conditions of all input transitions are false). In the case of a service specification, e.g., usually only the "normal" input interaction sequences are considered by the specification, since it is assumed that the user of the service follows certain rules about the order in which the input interactions are invoked.

If the situation of an incompleteness of the reference specification is encountered by the trace checker during the analysis of a trace, the analysis must be stopped, since at this point the specification is considered to

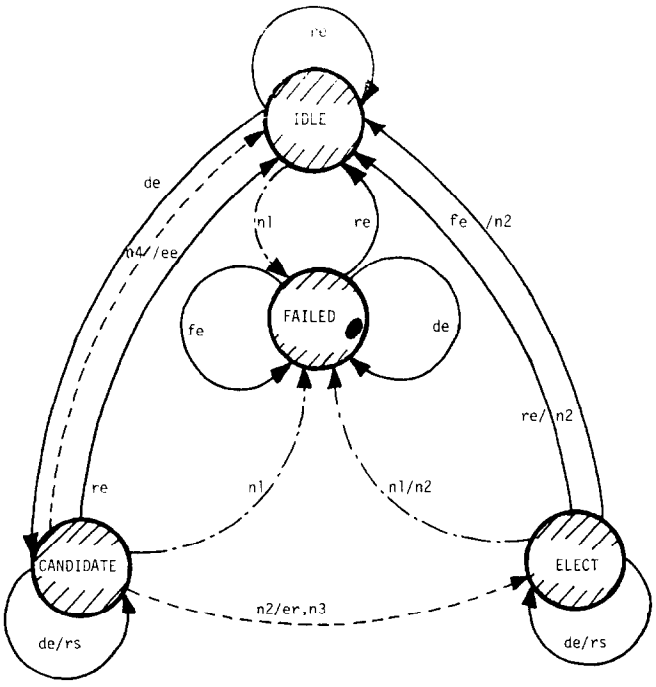


Figure 3. Mutual exclusion specification for one site. *Interactions:* re: reset (input); de: demand of exclusion access to resource (input); rs: refusal of resource access (output); ee: exclusion failure (output); er: reservation confirmed (output); fe: release of resource access (input). *Internal conditions:* n1: crash of the site; n2: resource is or becomes free; n3: resource not free; n4: time out elapsed; *Notations:* → :input; → :spontaneous output; →: internal.

allow an arbitrary behavior for the specified system. The following two possibilities should be considered:

1. The specification is not correct, i.e., should be completed to take care of the encountered input.
2. The test input provided to the system under test does not conform to the assumptions that are made, according to the specification, about the behavior of the environment of the system under test.

Let us consider again the specification given in Figure 3. This specification is not complete because no transition is specified for release request (fe) in the states IDLE and CANDIDATE. It is assumed that the user may only release the resource after he was elected to use it.

4. DESIGN AND IMPLEMENTATION OF A TRACE CHECKER

This section describes the design and implementation principles of a trace checker. Experience with such a trace checker is described in Sect. 5.

A. Principle of Operation

The task of a trace checker is to check whether a trace of observed interactions is an execution history that is a possible one according to the specification of the system under test. Its principle of operation is not very difficult, as explained in the following paragraphs.

We assume that the trace checker reads the trace sequentially, and records after each observed interaction the possible states in which the reference specification may possibly be.

We note that after a given prefix of the whole trace, the reference specification may be in one of several different states depending on the possible nondeterministic transitions of the specification. In fact, for each input interaction, the trace checker considers for each of the possible states the transitions that are possible for the given input. Any output interaction to be generated is recorded. Internal transitions must also be considered.

For each output interaction observed in the trace, the checker verifies for each of the possible states, whether the output was already generated by a previous input transition, or whether it may be generated by a spontaneous output transition. If for a given state, this is not possible (or only outputs with different parameter values can be generated) then the state in question is not in agreement with the observed trace, and must be eliminated from further consideration in the trace analysis.

If at some time during the analysis of the observed trace, no state of the reference specification remains in agreement with the grace, an error has been found.

B. A Checking Algorithm

We give in this section a more detailed description of the checking algorithm used in the trace checker we implemented. We also discuss the application of this algorithm to the example given in Sec. 3C.

As mentioned above, the trace checker makes a sequential analysis of the observed trace and maintains at all stages of the analysis a list of all states in which the reference specification may possibly be after the interactions already analysed. Such a "state" has two components:

1. The values of the state variables declared in the reference specification, and
2. The output interactions already generated by the considered transitions of the reference specification, but not yet observed in the trace.

The analysis of the next interaction in the observed trace builds up a list of possible next states, as follows: For each of the states in the list mentioned above, the following action will be executed. If the interaction is an input, then all possible input transitions will be considered, each leading to a next state which is placed on the list of possible next states. Any output interaction generated by the transition is recorded in the next state. Then all internal transitions are considered from the new next states found, leading in turn to next states. Some of these next states may already be in the list of next states. This process is continued until no new next states can be found.

As an example, we consider the specification of Figure 3 and a trace of the form ⟨re,de,er,fe⟩. This trace represents successively, the activation (re), the request (de), and confirmation (er) of the access to the resource followed by its release (fe). Figure 4 shows the checking tree for this trace. The (only) initial state of the specification is FAILED. Given the first interaction in the trace (re), only one input transition may be fired from this state. The next state is IDLE. From this state an internal transition is possible and produces the next state FAILED. Therefore the possible states after the (re) interaction are IDLE and FAILED. In the same way, the next input interaction (de) applied to these possible states give rise to the two possible next states CANDIDATE and FAILED.

While an input interaction in the trace usually increases the number of possible states to be considered, the output observed interactions usually reduce the number of possible states. If the next interaction in the analysed trace is an output, the following action is executed for each of the possible states: if the state contains an already generated output interaction which is identical to the output interaction of the trace, then this interaction is deleted from the state, and the state is

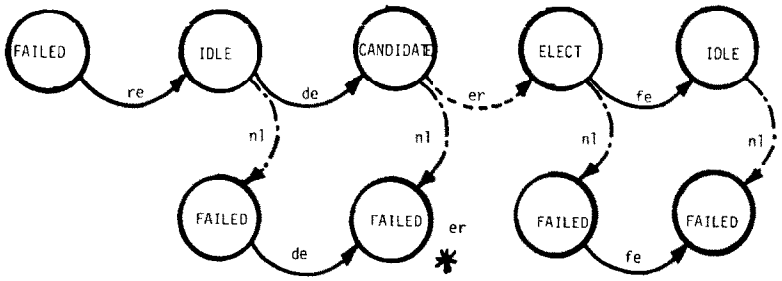


Figure 4. Checking for the trace ⟨re, de, er, fe⟩ (for notations, see Figure 3; a "*" indicates that a state has no possible next state for the given trace).

kept as one of the possible next states. Otherwise all possible spontaneous output transitions are considered to check whether they generate the observed output. Any different output generated is kept as state information, and any subsequent internal transitions are considered. After the exploration of all spontaneous output and internal transitions, only those states are kept that have generated the output observed in the trace. All other states are discarded from the further analysis.

Let us consider the next output interaction (er) in the trace of the exemple discussed above. The state FAILED does not give rise to any next states, since there is no sequence of transitions that leads from this state to the generation of the output interaction (er). From the state CANDIDATE, a spontaneous output transition may be fired to the state ELECT producing this output. In this state, the site may then crash. The system remains in one of the two possible states (ELECT and FAILED) and the checker is allowed to continue the analysis of the trace.

We note that different strategies for the analysis of a trace could be considered. For instance, it would be possible to check during the analysis of an input interaction whether an output interaction generated by the input transition can be found in the trace. It is important to note that, in general, it is not necessary that this interaction is the next interaction in the trace. Indeed, a certain degree of uncertainty may be allowed in the ordering of the interactions of the trace (see Sec. 3A), which means that interactions occuring at different interaction points may appear in the trace between the input and output interactions of a single transition of the specification. The algorithm described above takes care of this possibility.

C. Error Detection

The following three kinds of errors may be detected:

1. *Violation of safeness constraints (unexpected outputs).* If at some point during the analysis of the trace, no possible next state exists then the trace is in contradiction with the specification, i.e., the output interactions generated by the system under test do not conform with the specification.
2. *Inconclusive test result due to incompleteness.* If at some point during the analysis of the trace, no input transition is specified for the next input interaction of the trace for one of the possible states of the reference specification, then an incompleteness of the specification is encountered, and one of the possibilities mentioned in Sec. 3D must be considered.
3. *Termination errors.* Different situations may characterize the end of an observed trace. If the end of the trace is at an arbitrary point in the execution history of the system under test, not much can be said. If, however, the end of the trace corresponds to a stable state of the system under test, i.e., a state that does not produce any further output, then the following can be said about the termination: (a) If all possible final states of the reference specification include generated output interactions, then there is a termination error; the system under test should have produced one of the possible outputs. (There may be a deadlock.) (b) If all possible states either include generated output interactions or allow further output through spontaneous output transitions, then there *may* be a termination error, as in (a).

In the case of a detected error, it is, in general, not easy to locate the reason for the erroneous behavior of the system under test. In fact, the reason for such behavior may in some cases not be immediately visible in the observed trace, but only lead to a change of the internal system state, which may become apparent only in the later part of the trace.

D. Implementation

Trace checkers for the two reference specifications discussed in Sec. 5 have been implemented in PASCAL. This experience clearly demonstrates that two parts of a trace checker may be distinguished: the first one that is specification dependent, and the second one that is specification independent. The independent part consists of the checking algorithm explained in Sec. 4B. The specification-dependent part consists of two subparts:

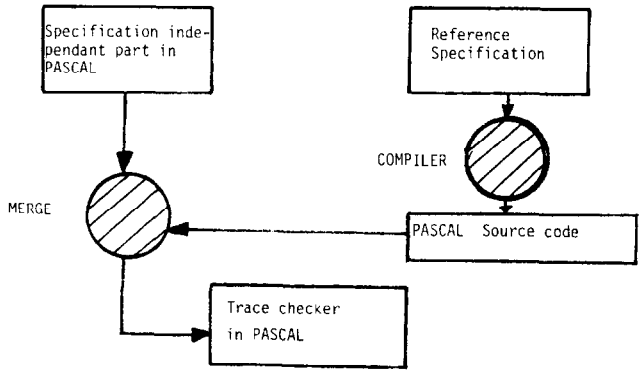


Figure 5. Automated generation of a trace checking module.

1. The declarations of the interactions, and
2. The program realizing the transitions of the specification.

It is noted that the specification-dependent parts are very similar to the PASCAL source code that is generated by a compiler that translates a formal specification written in the extended state transition model [11] into a PASCAL program [9]. It would not be difficult to adapt that compiler to generate directly the code needed for the specification dependent part of the trace checker. Following this approach, the implementation of a trace checker could be automated, as indicated in Figure 5.

5. APPLICATION EXPERIENCES

Two different protocols have been used for application experiences. The first has been briefly described (in Sec. 3C) to illustrate the principle of the checking algorithm: a distributed mutual exclusion protocol. Specifications of this protocol exist in [8] and the construction of an executable model for these specifications is described in [13]. This model was used for the validation of the defined protocol, using the trace checker described in Sec. 5A. Several errors in the protocol specification were found during this process, as explained in

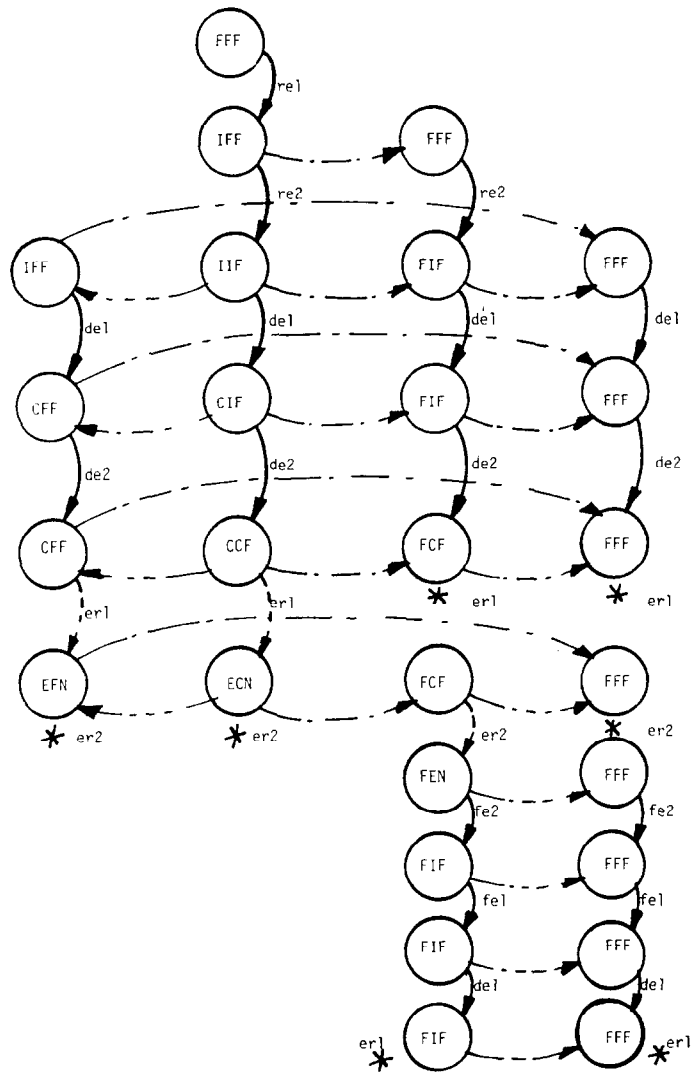


Figure 6. Checking tree for the trace ⟨rel, re2, del, de2, erl, er2, fe2, fel, del, erl, fel⟩ (for notation see Figure 3).

[13]. The second protocol is a Transport protocol developed by ISO and CCITT; a formal specification of the class protocol is given in [5]. An implementation of this protocol is being developed and the trace checker may be used to test that it provides the Transport service as specified in [4].

A. Distributed Mutual Exclusion Algorithm

A specification provided by the service of the protocol is given in Figure 3. There is such a state diagram for each site of the distributed system. Figure 6 shows the checking process for a more complex trace formed from the interactions of two users at sites 1 and 2. The checking algorithm of Sec. 4B is used. A state of the system is characterized by three letters [the initials of the state for user 1 and user 2 followed by the letter F or N indicating whether the resource is free (F) or not (N)].

The trace which is explored in Figure 6 includes an error (two successive exclusion confirmation er1 and er2 in the absence of failures). Naturally, this error is detected; however, we see here a good example of a later error detection: when checking the output interaction er2, a possible valid state is FEN where sitel is crashed. A violation of the service specification is detected when the next output (er1) of site 1 is checked. Figure 7 shows the output produced by the checker program for this trace: the number of points after each interaction indicates the number of possible states after the analysis of each of the interactions indicated at the beginning of the line.

B. Transport Protocol

While the mutual exclusion example, with the absence of interaction parameters, allows a quasifinite state reference specification, the Transport service specification [4] relies heavily on interaction parameters and additional state variables for expressing the service properties. It is interesting to note that for this example the finite (major) state does not induce any nondeterminism for the trace checker. In fact, there are no internal transitions. Some nondeterminism due to nonspecified values of certain additional state variables does not lead to any problem since the value adopted by the unit under test can be deduced by the next output produced.

Figure 8 shows an example of an output obtained for the Transport service: the trace checker detects an error in the ordering of data units during data transfer after the connection establishment phase. Unit 3 is received by user 2 at the access point AP2 before the unit 2.

Figure 7. Checker output for the mutual exclusion algorithm.

```
-- TRACE CHECKER __ V04/05/11/81 __

        .
OP(1).RE . .
OP(2).RE . . . .
UT(1).DE . . . .
UT(2).DE . . . .
UT(1).ER . . . .
UT(2).ER . .
UT(2).FE . .
UT(1).FE . .
UT(1).DE . .
UT(1).ER

 END OF ANALYSIS ....

 TIME USED : 0.64 SECONDS ..


..... STOP CHECKING .....
..... BECAUSE OF AN UNEXPECTED OUTPUT
```

6. CONCLUSION

The trace checker described in this paper is a module which compares the observed behavior (i.e., a trace) of a system under test with the requirements of a refer-

Figure 8. Checker output for the Transport protocol.

```
-- TRACE CHECKER __ V04/05/11/81 __

AP1.T_CONNECT_REQ ( TCEPI  1
                    TO_T_ADDRESS  2
                    FROM_T_ADDRESS  1
                    QOTS_REQUEST
                    OPTIONS  0
                    TS_CONNECT_DATA  CONNECT )
AP2.T_CONNECT_IND ( TCEPI  1
                    TO_T_ADDRESS  2
                    FROM_T_ADDRESS  1
                    QOTS_REQUEST
                    OPTIONS  0
                    TS_CONNECT_DATA  CONNECT )
AP2.T_ACCEPT_REQ  ( TCEPI  1
                    QOTS_REQUEST
                    OPTIONS  0
                    TS_ACCEPT_DATA  ACCEPT )
AP2.T_DATA_REQ    ( TCEPI  1
                    TSDU_FRAGMENT  DATA2-1 1 )
AP2.T_DATA_REQ    ( TCEPI  1
                    TSDU_FRAGMENT  DATA2-1 2 )
AP1.T_ACCEPT_IND  ( TCEPI  1
                    QOTS_REQUEST
                    OPTIONS  0
                    TS_ACCEPT_DATA  ACCEPT )
AP1.T_DATA_REQ    ( TCEPI  1
                    TSDU_FRAGMENT  DATA1-2 1 )
AP1.T_DATA_REQ    ( TCEPI  1
                    TSDU_FRAGMENT  DATA1-2 2 )
AP1.T_DATA_IND    ( TCEPI  1
                    TSDU_FRAGMENT  DATA2-1 1 )
AP1.T_DATA_REQ    ( TCEPI  1
                    TSDU_FRAGMENT  DATA1-2 3 )
AP1.T_DATA_IND    ( TCEPI  1
                    TSDU_FRAGMENT  DATA2-1  2 )
AP2.T_DATA_IND    ( TCEPI  1
                    TSDU_FRAGMENT  DATA1-2 1 )
AP2.T_DATA_IND    ( TCEPI  1
                    TSDU_FRAGMENT  DATA1-2 3 )

 END OF ANALYSIS ....

 TIME USED : 0.63 SECONDS ..

..... STOP CHECKING .....
..... BECAUSE OF AN UNEXPECTED OUTPUT
```

ence specification. The paper discusses how such a trace checker can be obtained semiautomatically for any reference specification given in an extended state transition description technique. Such a trace checker may be used for testing an implementation of a system for which a formal specification is given, or for testing the consistency of refined specifications in respect to a more abstract reference specification if the refined specifications exist in an executable form. The selection of appropriate input test sequences is a related problem, which is not addressed in this paper.

The objective of our work was to apply this testing approach to the validation of communication protocols. Some practical results are reported in [13]. The application of this approach to a mutual exclusion and a Transport protocol shows that the nondeterminism inherent in the reference specifications of these examples does not lead to an excessive number of possible states to be considered for the analysis of typical traces. We believe that this technique is equally suitable for other areas of software design.

REFERENCES

1. D. Rayner, A System for Testing Protocol Implementations, *Comput. Networks* 6 (1982).
2. G. V. Bochmann and C. A. Sunshine, Formal Methods in Communication Protocol Design, *IEEE Trans.* COM-28, 4, 624–631 (1980).
3. G. V. Bochmann, A General Transition Model for Protocols and Communication Services, *IEEE Trans. Commun.* COM 28, 643–650 (1980).
4. G. V. Bochmann, E. Cerny, and C. Lacaille, Formal Specification of a Transport Service, Département d'IRO, Université de Montréal, also WASH-9 of ISO TC97/SC16/WG1 ad hoc group on FDT.
5. ISO TC97/SC 16/WG1 ad hoc group on FDT, Trial Specification of Transport Protocol Using Subgroup B FDT (July 1983).
6. G. V. Bochmann and M. Raynal, Structured Specification of Communicating Systems, Publ. Département d'IRO, Université de Montréal.
7. J. M. Ayache, P. Azéma, and M. Diaz, Observer: A Concept for Detecting at Run Time Control Errors in Concurrent Systems, LAAS, presented at the IEEE Fault Tolerant Computing Symposium, Madison, June 1979.
8. Modèle Externe du Protocole D'exclusion Mutuelle du Projet Galaxie, Document Interne, CNET *groupe EVP* (Juin 1981).
9. M. Gagné, Un Compilateur Pour un Langage de Spécification, Document de Travail (Dec. 1981).
10. J. Goguen, Thoughts on Specification, Design and Verification, *ACM Software Eng. Notes* 5, 29–33 (1980).
11. A FDT Based on an Extended State Transition Model, ISO TC97/SC 16 N (May 1983).
12. C. Jard, Définition d'un Modèle de Simulation Pour le Validation de Protocoles, Note technique CNET Lannion, Groupe EVP, NT/LAA/SLC/49 (Juin 1981).
13. C. Jard, Spécification et Validation d'un Algorithme Distribué D'exclusion Mutuelle—Mise en Oeuvre de la Simulation: Méthode et Résultats, Note technique NT/LAA/SLC/93, CNET Lannion, France, July 1982.
14. L. Lamport, Time, Clocks and the Ordering of Events in a Distributed System, *Commun. ACM* 21, 558–565 (1978).