ELSEVIER

Computer Networks 31 (1999) 1835–1872

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Test development for communication protocols: towards automation

R. Dssouli [a,*], K. Saleh [b,1], E. Aboulhamid [a,2], A. En-Nouaary [a,3], C. Bourhfir [a,4]

[a] *Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Québec, P.Q. H3C 3J7, Canada*

[b] *Kuwait University, Department of Electrical and Computer Engineering, P.Q. Box 5969, 13060 Safat, Kuwait*

Abstract

In this paper we give an introduction to methods and tools for testing communication protocols and distributed systems. In this context, we try to answer the following questions: Why are we testing? What are we testing? Against what are we testing?... We present the different approaches of test automation and explain the industrial point of view (automatic test execution) and the research point of view (automatic test generation). The complete automation of the testing process requires the use of formal methods for providing a model of the required system behavior. We show the importance of modelling the aspects to be tested (the right model for the right problem!) and point out the different aspects of interest (control, data, time and communication). We present the problem of testing based on models, in the form of finite state machines (FSMs), extended FSMs, timed FSMs and communicating FSMs, and give an overview of the proposed solutions and their limitations. Finally, we present our own experience in automatic test generation based on SDL specifications, and discuss some related work and existing tools. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Distributed testing; Finite state machine; ISO standard test architectures; Protocol conformance testing; Protocol engineering; Software testing; Test generation

1. Introduction and motivations

As the data communication technology is progressing at a rapid pace and becoming more and more complex, the standardization of communication protocols and interfaces has been playing a key role in the development of computer communication systems.

The Open Systems Interconnection (OSI) Reference Model has been useful in placing existing protocols in an overall communication architecture and the development of new protocol standards. The term open systems means that if a system conforms to a standard, it is open to all other systems conforming to the same standard for communication.

In order to assure successful communication between computer systems from different manufacturers, it is not sufficient to develop and standardize communication protocols. It must also be possible to ascertain that the implemented protocols really conform to these standard protocol specifications. One way to do this is by testing the protocol implementa-

[*] Corresponding author. E-mail: dssouli@iro.umontreal.ca

[1] E-mail: ksaleh@cairo.eng.kuniv.edu.kw

[2] E-mail: aboulham@iro.umontreal.ca

[3] E-mail: ennouaar@iro.umontreal.ca

[4] E-mail: bourhfir@iro.umontreal.ca

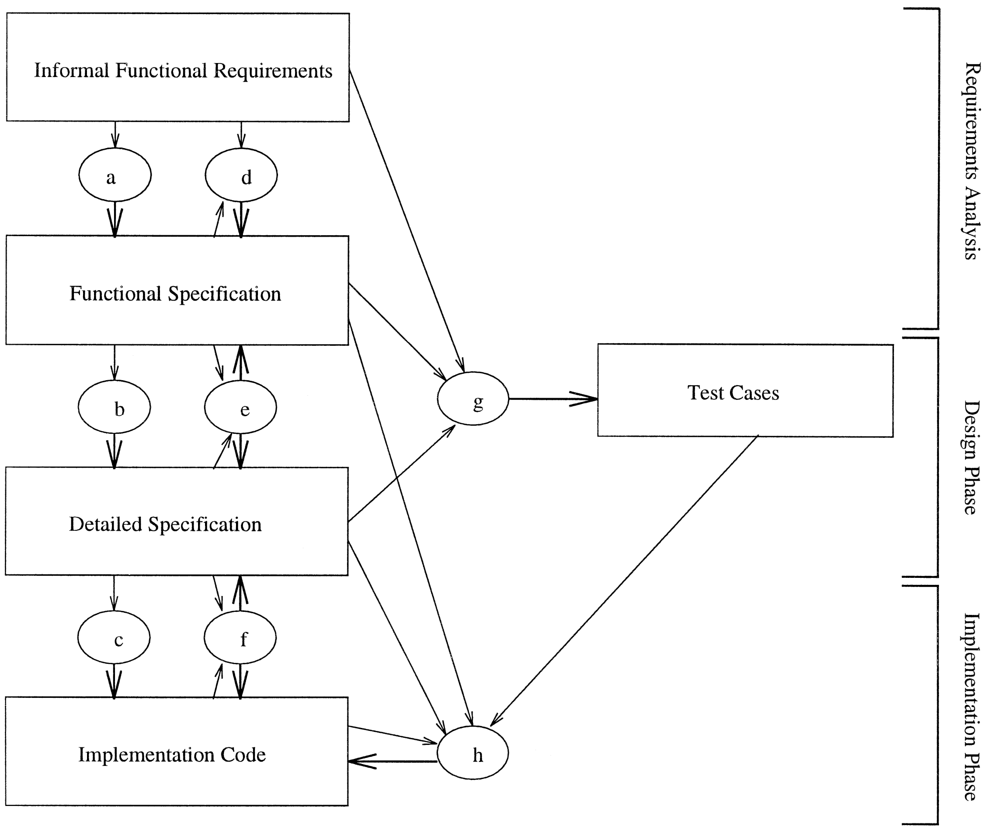
1389-1286/99/$ - see front matter © 1999 Elsevier Science B.V. All rights reserved.
PII: S1389-1286(99)00063-8

tions. This activity is known as protocol conformance testing.

Testing plays a major role in the development of communication protocols. Protocol incompatibility between or among two or more systems can have various causes. First each protocol usually provides a range of options that may result in mutual incompatibility between or among two or more systems. Second, different implementations of the same protocol might result from various interpretations of the protocol specification. Third, due to the complexity of protocols, developers may introduce errors. Finally, incompatibilities may result from incompletely specified protocols and procedures that cover, for example, system administration, system management, or maintenance of individual systems.

A communication protocol is a set of rules that govern an orderly exchange of messages among communicating entities. Communication protocols are in fact a subset of software; they are characterized by complex features such as distribution, communication and synchronization.

The construction of a communication software is based on a disciplined approach known as protocol engineering, see Fig. 1. The aim of this approach is to develop safer, performant and easy to test and maintain communications software. The key elements fulfilling this aim reside in promoting and



a : Requirement Engineering Activity	e : Design validation
b : Refinement	f : Module testing
c : Coding Activity	g : Test Cases Development Activity
d :Specification validation	h : Testing Activity

Fig. 1. Software life cycle activities.

integrating formal methods in the protocol engineering cycle, to develop support tools, and to provide procedures and standards [17].

Fig. 1 shows different activities of the software development cycle, and documents delivered in each of them [8]. First, the service concept should be defined, this is known in software engineering as requirement engineering phase. In the specification phase, a complete protocol definition will be produced for the required service. The specification must describe what the protocol should do, what it should not do, and how it should react to external stimuli. A protocol specification must be verified to ensure that it is complete, and free of logical and functional errors, and that it correctly delivers the intended service. After the development phase, conformance testing verifies whether an implementation of a protocol complies to its specification. The testing activity, for example, starts early in terms of test cases development. Different test cases have their origin in the different steps of the system refinement cycle: requirements (e.g. use cases / scenarios) specification, design, coding.

In addition to conformance testing, other types of testing have been proposed including: (i) inter-operability testing to determine whether two implementations (or more) will actually inter-operate and if not, why, (note: inter-operability testing requires $N \times N$ test campaigns between N different implementations as shown in the Fig. 2; conformance testing against the protocol specification requires only N campaigns); (ii) performance testing to measure the performance characteristics of an implementation, such as its throughput and responsiveness under various conditions; (iii) robustness testing to determine how well an implementation recovers from various error conditions and abnormal situations.

Analysis of test results and their diagnostics are activities that take place in the software life cycle. In these activities there must be some (formal or informal) reference specification for testing that defines, for any behavior observed during testing, whether it is correct or not [34,11]; this is the role of an oracle. Oracles can be easily built for deterministic specifications; the difficulty of an oracle is in the case of nondeterministic specifications. Diagnostics should pin-point the location and the type of faults introduced in the implementation [52,53].

In the software area, we distinguish between black-box and white-box testing. In many cases, for instance, in protocol conformance testing, the internal structure of the tested software product is not known; it is tested with respect to a reference specification, the structure of which is known. Since the internals of the tested system are not known, this situation is called ‘‘black-box’’ testing. In this case,

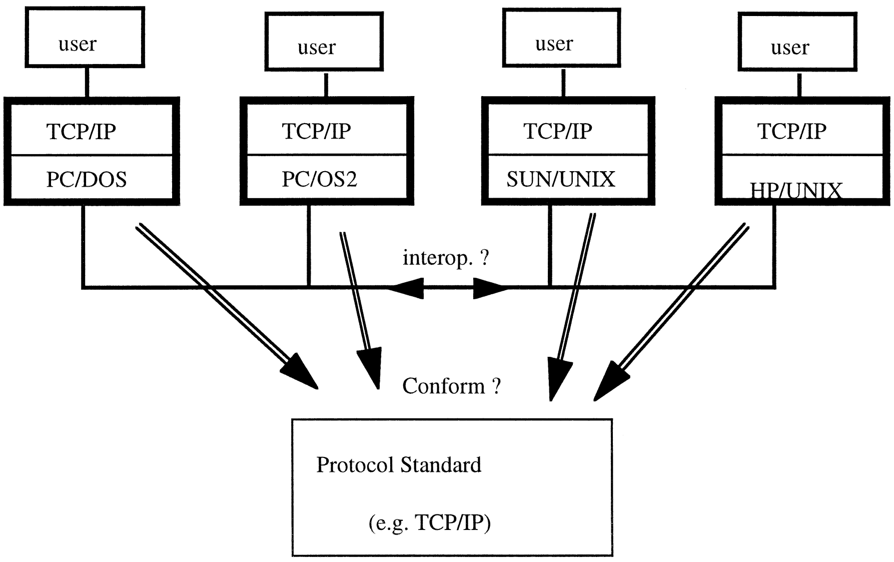


Fig. 2. Interoperability testing.

the reference specification is usually taken as the basis for test suite selection, coverage criteria, and test result analysis. The diagnostics may, for instance, indicate which part of the reference specification is not correctly implemented.

The situation where the internal structure of the tested system is known is called ''white-box'' testing. Also in this case, the tested system is to be compared with a more abstract reference specification. However, both, the knowledge about the tested system and the reference specification, may be used for test selection, coverage criteria and test result analysis. The diagnostics should usually indicate which part of the tested system is responsible for the faulty behavior. In-house testing of software is usually white-box, since the structure of the tested program is known. Sometimes the term ''grey-box'' testing is used to denote the situation where the modular structure of the tested software product is known to the testers, but not the details of the programs within each component.

Definition 1 (*Definition of a test case*). A test case defines (i) a finite sequences of (input) interactions to be applied to the implementation under test (IUT), and (ii) a finite sequence of expected output interactions generated by the IUT.

In addition, a test case may include information on how to analyse the (output) interactions received from the IUT in response to the input; this is the oracle function, and may also include diagnostics information and test preambles or preconditions. A test suite is a finite set of test cases. Each test case may be applied to the IUT independently from other test cases. Usually the order of application of the test cases is arbitrary, although some particular order may be suggested for some pragmatic considerations.

In the next section we present an overview of the test automation process and the different activities that are required. In the rest of the paper, we will present the state of the art related to each activity including comments and discussions.

2. The automation of testing activities

The great hope for software developers is to achieve a complete automation of the testing process [47] for many reasons, the cost of testing that constitutes a major part in the overall development cost, and the ''fault'' coverage guarantee offered by test automation is much higher. There exist two different understanding of the test automation process, see Fig. 3. For the industrial world, automation starts when a test suite is available (usually obtained manually). For the academic world, automation deals initially with the test suite generation. Actually the whole process can be automated for simple models, but for models that take into account the different aspects listed earlier, there is a long way to go in research.

2.1. Classification of testing activities

The automation of software engineering activities, in most cases, requires the formalization of the artifacts that are manipulated during these activities, thus allowing these activities to be automated. In the context of testing, the relevant artifacts are: the reference specification, the test cases, and the trace of observed interactions obtained during test execution, and in the case of white-box testing, the program representing the IUT. In the following, we concentrate mainly on the activity of test suite development for black-box testing, where the reference specification is of prime importance. We assume therefore that a formal model of the system specification is available, either in the form of an FSM or Specification and Description Language (SDL).

2.2. The nature of test cases

The typical structure of a test case has been defined by the OSI conformance testing methodology [56]. This structure assumes that each test case has: (i) a well defined ''test purpose'', (ii) a test preamble leading the IUT into the state in which the expected behavior corresponding to the test purpose can be observed, (iii) a test body invoking the behavior corresponding to the test purpose, (iv) a checking part observing the output in order to determine whether the expected behavior did effectively occur (note: in the case of FSM testing, this is essentially the state identification part; the expected transition output is already observed during the execution of the test body), and finally, (v) a test postamble leading the IUT into a neutral state (often the initial state from which another test case can be applied).

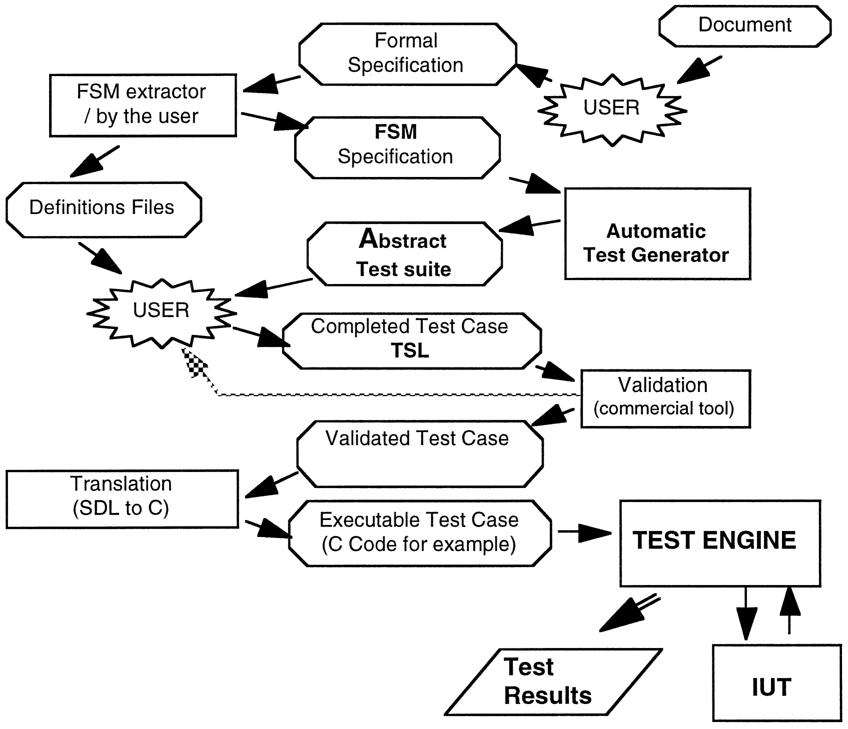


Fig. 3. Test automation.

Test case properties can be specified at different levels of abstraction.

A distinction between abstract test case and executable test case is often made. The most important aspect is the form of the interface through which the test case interacts with the IUT. An abstract definition of the interface is part of the system specification (the reference for testing). The detailed specification of the real interface may also be included in the reference specification, for example in the form of an Application Programming Interface (API), usually given in terms of a set of procedure heading definitions and associated data types. An executable test case uses a real interface (software API and/or hardware). The so-called abstract test cases used for protocol conformance testing are defined in terms of the abstract service interfaces which are associated with the protocol.

The behavioral aspects to be specified for a test case are similar to those of the specification of the correct system behavior. The specification of a test case includes the definition of the inputs to be applied to the IUT (possibly in response to particular outputs received from the IUT). It may also include the definition of the expected output interactions and the constraints on the expected outputs and their parameter values. If the observed output does not satisfy these constraints, the verdict of the test execution is FAIL. However, if the test case does not specify the expected output, a separate oracle is required to analyse the test results (trace of applied inputs and observed outputs) and provides the verdict. The possible values of a verdict include the following: FAIL indicates that the observed behavior is in contradiction with the reference specification), and PASS or INCONCLUSIVE indicating that the observed behavior conforms to the reference specification. Moreover, in the case of INCONCLUSIVE, the test purpose was not covered during the execution of the test, possibly due to an unexpected (but allowed) choice of interactions by the IUT (e.g. not accepting a request because of the lack of resources; some test purposes are difficult to test because of possible race conditions).

Usually, the test designer defines a test suite in such a way that each test case has a particular ''purpose'', i.e. a particular set of faults for which the test case is designed to detect.

The OSI conformance testing methodology [56] gives several examples of test purposes for protocol testing. Most standardized conformance test suites for communication protocols consist of a hierarchically organized set of test cases, each having a specific test purpose. Often these purposes are related to a given transition of the protocol specification. Moreover, an important subset of the test purposes to be supported by a given test suite are obtained by considering the state transition table of the protocol specification (a form of FSM model of the protocol) and defining a test purpose for each transition that is considered important. In contrast, certain test derivation methods foresee the concatenation of several test cases into a single sequential test case which combines many purposes. The execution of such a combined test case is more efficient that the sequential execution of all the original test cases that were combined. Such optimisation has the following drawbacks: (i) the intuitive notion of ''test purpose'' is largely lost, and (ii) the diagnostic power of the tests is reduced.

One may ask in which language should the test cases be defined? Many languages might be used to define test cases. (1) Programming language, e.g. C same as implementation language, that is ease of use for in-house testing. It is at low level of abstraction, often leads to lengthy programs and it is implementation-dependent. This type of language is not so interesting for standardized conformance tests. (2) TTCN: standardized by ISO and IUT-T (CCITT) for describing protocol conformance tests. It is difficult to understand by people not familiar with language, and there a need for editing and translation tools. (3) SDL: standardized by IUT-T has an intuitive graphical representation for many aspects also used for protocol specifications. In this case too, there is a need for editing and translation tools.

2.3. Test execution

Automated test execution is important, especially for debugging (where the same tests are executed on different versions of implementations) and for regression testing. In most cases, abstract test cases are translated into programs which are compiled and executed, after being linked directly or through some inter-task communication facilities with the IUT (test engine for example). Various tools exist for the management of a test campaign. Such tools control the execution of many test cases in sequence, including conditional execution depending on the verdicts of previous test cases. The test results are usually automatically recorded in the form of so-called test traces. No standard exists for the description of these traces. Certain tools for the automatic analysis of these traces also exist. Moreover, there are possibilities for partially automating the test result analysis and diagnostics processes, and obtaining automatically an oracle starting from a given specification. In the case of a deterministic specification which is written in an executable specification language, an execution environment for this language provides the means for obtaining an oracle, since it is sufficient to execute the specification with the same inputs as applied to the IUT: the output obtained from the specification is the output to be expected from the IUT. An Oracle for deterministic specifications is shown in Fig. 4.

In the case of nondeterministic specifications, its execution will provide only one of all the possible outputs (or output sequences) which are allowed. Their direct comparison with the outputs obtained from the IUT is not suitable for determining whether this output is correct. Simulated execution with back-tracking has been explored for the automatic generation of oracles and diagnostics from a given executable reference specification (e.g. TETRA for LOTOS specifications, and TANGO for Estelle specifications, developed at University of Montreal). Unfortunately, in many cases there exist an explosion of possibilities for unsuccessful backtracking, and these methods become very inefficient, if not unfeasible, in these cases. However, in other cases, useful results have been obtained.

2.4. Automatic FSM diagnostics

Using the FSM fault model of output and transfer faults, certain methods allow the automatic generation of diagnostic information when a fault is detected. The diagnostics are based on the knowledge of the reference specification (an FSM model) and the obtained output from the previous execution of other test cases. Various algorithms have been developed for single and multiple fault diagnosis, includ-

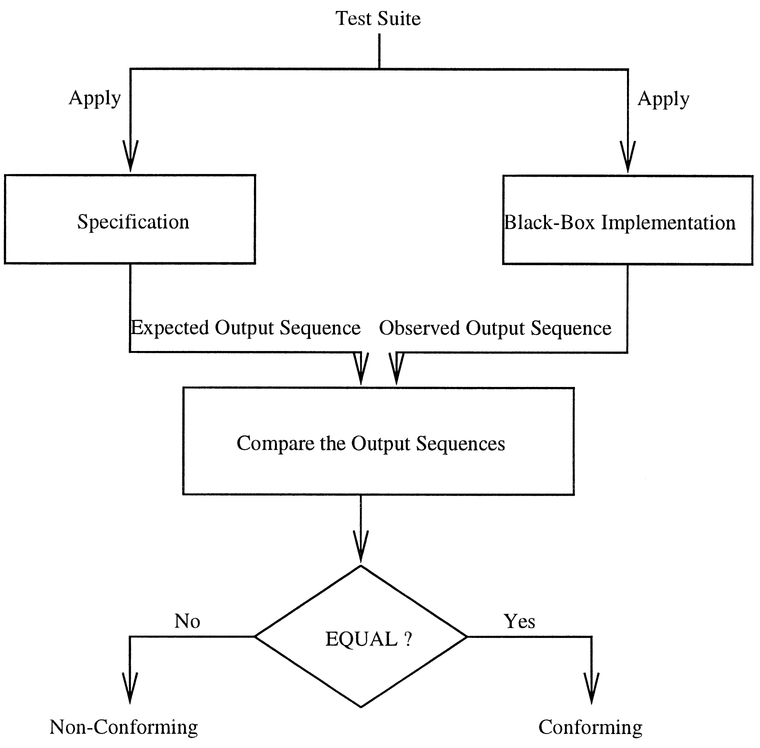


Fig. 4. Oracle.

ing the case of several communicating FSMs. Some of these algorithms have been implemented [53,43]

3. Testing models and architectures

One may ask why do we test? A simple answer might be: for detecting errors in implementations. But testing is seen as a process for demonstrating the conformance to a reference specification, e.g. protocol conformance testing. One may also consider testing as a way for the assessment of the correctness of an implementation.

In fact, testing is a method of software verification that deduces from execution results or traces that the software under test possesses certain ‘‘good’’ properties. The intuitive meaning of test is the following, ‘‘a good test provides convincing evidence that an implementation is correct’’ [74]. The problem is how to obtain this ‘‘good test’’.

Howden [50] defined a reliable test to be a test whose success implies the implementation correctness. The problem with achieving the correctness via testing is that a reliable test is not attainable in general. To achieve implementation correctness in protocol testing means to apply exhaustive testing, which implies in general to apply an infinite input test suite to the implementation. Thus, it is important to define clear criteria for the description and the evaluation of the quality of a test. The notion of fault coverage should play this role when test is based on a fault model. Testing is a major concern in the computer industry; in practice a test campaign is allowed a limited time and budget. For these reasons, compromises should be made between the cost of testing, the feasibility of exhaustive testing and the ‘‘target’’ quality of the end product one would like to obtain via testing.

Testing is in some sense how to obtain a ‘‘finite test suite’’ from an infinite behavior.

For most systems, not all their behaviors can be tested. Examples:

- For boolean function of two arguments, only 4 possible input patterns are possible (easy to test).

- For a hardware multiplication unit for fixed-point integers with a precision of 32 bits, there are 2**32 × 2*32 possible input patterns (practically impossible to test all).
- For a sequential machine with two states (in its specification), the length of input sequences is unbounded; therefore an infinite number of tests are possible (impossible to test all)

We can show that a finite number of finite test cases cannot detect all possible faults in the implementation of a sequential machine, by constructing an implementation which is correct for any input sequence not longer than the given test cases, but which is wrong for longer sequences.

The correctness via testing is also a problem of conformance relation verification.

Definition 2 (*Conformance relations*). What is a correct implementation? [61,14] There are many possible definitions of a conformance relation. Here are some examples:

- For sequential program specified through input and output assertions: for any input which satisfies the input assertion, the output satisfies the output assertion.
- For sequential machines with deterministic (partial) specification: for any input sequence for which the specification is defined, the implementation provides the same output as the specification. (This relation is called quasi-equivalence.)
- The coverage problem and the fault models. There are usually an infinite number of possible faulty implementations, not all of them can be detected by the tests to be applied. How can one characterize the types of faults that will be detected (covered)?
- Distinguishing the non-conforming implementations. One usually introduces (often implicitly) a model of the possible implementations and/or of the possible faults. For instance, in the mutation testing approach, one considers various ‘‘mutations’’ of a correct program (or of the reference specification) and checks whether such mutations would be detected by the tests to be applied.

The question of test coverage becomes: Among all the faulty implementations considered within the given fault model, which implementations (or what percentage of these implementations, or which single faults in implementations, or which sets of multiple faults) will be detected by the tests to be applied?

Definition of complete fault coverage [99]: A test suite TS has complete fault coverage with respect to a given fault model if for each implementation considered by the fault model, it either satisfies the conformance relation (i.e. it is not faulty) or there is a test case in TS which results in the verdict FAIL when executed against the implementation (see Fig. 5).

3.1. Cost models for the testing process

The cost of the testing phase within the development cycle includes the following aspects:

- Cost of test development.

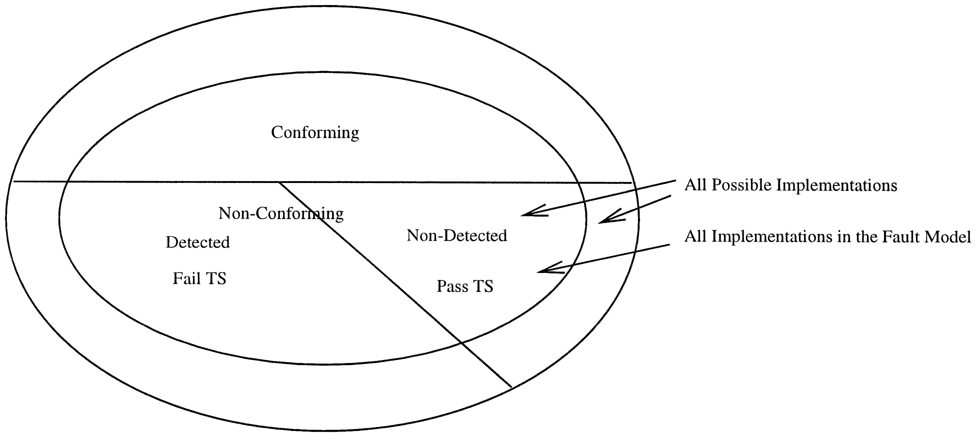

Conforming
Non-Conforming
Detected
Fail TS
Non-Detected
Pass TS
All Possible Implementations
All Implementations in the Fault Model

Fig. 5. The implementation univers.

- Cost of test execution.
- Cost of test result analysis, diagnostics, etc.
- Cost of modifying the implementation under test in order to eliminate the detected faults.

The cost of test execution is often based on the number of test cases to be executed, or the total number of input/output interactions included in all test cases. The optimisation methods mentioned above address this point. In order to determine at what point a system has been tested thoroughly (a question difficult to answer), it is necessary to consider not only the cost required to detect and eliminate the remaining undetected faults in the system under test, but also the costs which occur if these undetected faults remain in the system. The latter cost is difficult to estimate; it is clearly higher for systems with high reliability requirements.

Optimizing the testing process means organizing this process in such a manner that the overall cost (testing cost plus cost of remaining faults) is minimized.

3.2. Models and their representation

Most systems are too complicated to be understood in detail and to be tested. Very often, we have to build simplified models that allow us to reason about them. In our context, a model is a particular mathematical model that represents and simplifies our concept of a system. In general, the use of a model, makes the study of some complex problems easier [6,78]. In order to have a complete view, we have to build a model of a system specification, a model of a real implementation to be tested and a model of faults that we try to capture during the testing process.

Often simplified models of the reference specification are built to approximate the precise specification. Often one assumes simplified models for the behavior of the IUT (corresponding to the assumed fault model) in order to justify the method for test suite development. In Fig. 6, the conformance relation that may be verified by testing is a conformance relation that takes place between two abstract models. The implementation abstract model and the specification abstract model consider behavior aspects that are important to test. The temptation to deduce from this modelling process and testing, is that a corresponding conformance relation is verified between an entire real specification and its corresponding entire real implementation is a misunderstanding of the modelling step. In the case where a modelling of a specification and its corresponding implementation is adequate, and assumptions are verified, the conformance relation that holds between a real specification and its implementation concerns only the modeled aspects.

How is the correct behavior (i.e. the reference specification) specified?

What aspects of behavior are important?

To answer these questions, one may consider the various aspects of the behaviour to be specified and to be tested:

(a) Temporal ordering of interactions.

(b) Range of possible interaction parameters [not to be tested].

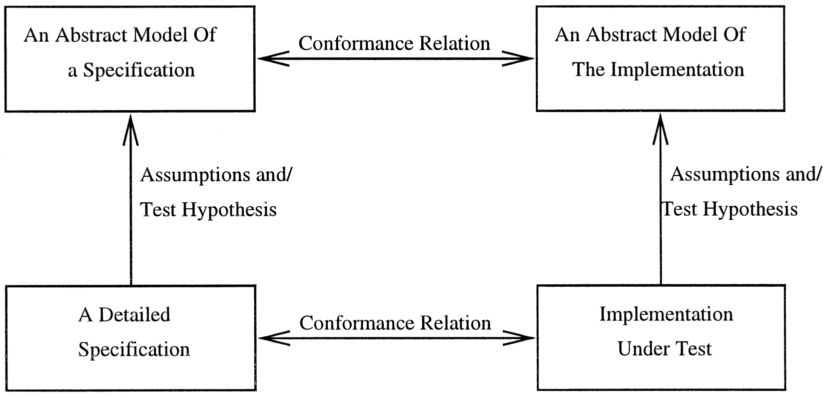

An Abstract Model Of a Specification
Conformance Relation
An Abstract Model Of The Implementation
Assumptions and/ Test Hypothesis
Assumptions and/ Test Hypothesis
A Detailed Specification
Conformance Relation
Implementation Under Test

Fig. 6. The modelling.

(c) Rules concerning actual values of parameters.
(d) Coding of interactions (PDU's).
Corresponding specification languages are:
(a): FSMs, Petri nets, grammars, LTS.
(b,c): Abstract data types.
(a,b,c): Programming and specification languages.
(b,d): ASN.1 (used for defining protocol messages).

Formal Description Techniques FDTs developed by ISO and ITU (CCITT) for OSI communication protocols and services are:

- SDL (CCITT): version 1980: interconnected FSM's.
- version 1988: extended FSM, module interconnections.
- version 1992: extended with object-oriented features.
- Estelle (ISO): extended FSM model, module interconnections.
- LOTOS (ISO): process algebra and abstract data types.

Certain formal methods (used in Europe):

- Z (first order predicate calculus + sets).
- VDM (similar concepts).

A legitimate question arises,which models are the most useful? The answer is to use the right model for the right system! In the following, we list the suitable models for each aspect to test:

- Control flow, Finite State Machine (FSM).
- Data flow, high-level programming language.
- Data and control aspects, Extended FSM (e.g. SDL or Estelle).
- Communicating Components, Communicating FSMs, Communicating EFSMs, (e.g. SDL or Estelle).

3.3. ISO standards test architectures for protocol conformance testing

ISO has defined four types of test architectures for protocol conformance testing. These are the local, distributed, coordinated and remote test architectures [56,82]. Local test architecture corresponds to traditional software testing, where PCO and IUT refer to Points of Control and Observation, and the implementation under test, respectively. In this architecture, the two PCOs of the IUT can be viewed as a single port since the IUT and the tester are located in the same place. In the distributed test architecture, the system is divided into so-called upper and lower testers which access the PCO1 and PCO2 of the IUT, respectively, as shown in Fig. 7. The lower interface PCO2 is accessed over distance and indirectly through the underlying communication service. The coordinated test architecture is similar to the distributed test architecture, and the only difference between the two is that the former has some kind of coordination between upper and lower testers, established using the so-called test coordination protocol through a (possibly separate) communication channel between upper and lower testers. The remote test corresponds to the distributed test architecture where only a lower tester is used, the IUT may include a stack of several protocol layers above the layer being tested.

Test architectures differ in their observability and fault detection capabilities [34,35]. For a system that receives inputs and produces outputs, the observability refers to the ease of determining if specified inputs affect the outputs; fault detectability refers to the ease of detecting specified faults. Observability and fault detectability of an IUT vary in different test architectures. Consider the ISO test architectures: an IUT has the highest observability and fault detectability in the local test architecture, and has a lowest observability in the remote test architecture. The ISO test architectures only deal with an individual single protocol entity. To test the overall properties of distributed application and communication networks, one may face a general distributed test architecture where the IUT has several ports (PCOs)

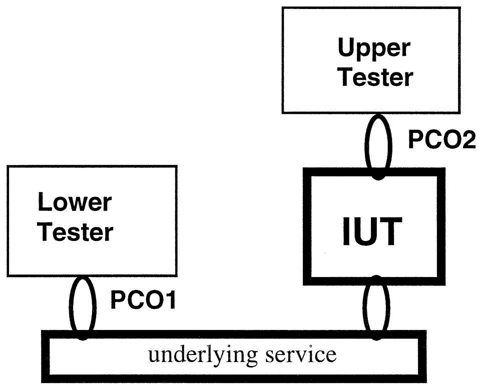


Fig. 7. The distributed test architecture.

and corresponding testers cannot communicate and synchronize with one another unless they communicate through the IUT, and no global clock is available [66].

In the next section we give an overall view of test hypothesis and assumptions that are often made to reduce the set of implementations to consider for testing.

4. Hypothesis and assumptions

Test hypothesis have been introduced to simplify testing. Some of these hypothesis were made implicitly such as the capability of the programmer or the testing system is correct. The purposes of the use of test hypothesis and assumptions, when a specification of a system is determined, is the reduction of the set of implementations to consider for testing. This can be achieved in the following two possibilities [92]:

- The use of test purposes which may have the effect of partially covering a given specification, or enlarge specification which imply to test less properties.
- The verification of a weaker conformance relation. It is well known that the verification of weaker conformance relation between the implementation and a specification requires less tests

4.1. General test hypothesis

Test hypothesis have been used for a long time in testing. The explicit definition of test hypothesis concept was given for the first time in Refs. [7,41]. They applied the concept to algebraic specifications. Tretmans, Phalippou and Charles [92,78,23] based a large part of their thesis on these concepts. The usual test hypothesis are the following: the regularity assumption, the uniformity assumption, the independency assumption and the fairness assumption.

- The regularity assumption: This type of assumption allows to limit testing to a finite set of behaviors for systems that exhibit an infinite behaviors. Examples are programs (or specifications) with loops and integer input and output parameters, finite state machines, and reactive systems, in general.

  Principle: assume that the implementation has a ‘‘regular’’ behavior, which means that the number of control states of the implementation is limited.

  If the number of states is not higher than the corresponding number of states of the specification, then all loops (of the specification) have to be tested only once. This corresponds to the idea behind the FSM fault model where the number of implementation states is limited to n, or to some number $m > n$, where n is the number of states in the specification and m a number of states in the implementation. This is also the idea behind certain approaches for testing program loops and for testing with respect to specifications in the form of abstract data types.
- The uniformity assumption or congruence: The origin of this assumption is in Partition Testing ‘‘There exist similar behaviors, if grouped under an equivalence relation, then it is sufficient to test one behavior of each equivalence class for conformance testing.’’

  Principle of partition testing: Apply test for at least one representative for each partition of the input domain (equivalent actions for EFSM, equivalent states for FSM).
- The independency assumption: The different submodules of the system under test are independent, and faults in one module do not affect the possibility of detecting the faults in the other modules. This means that the testing of a system is equivalent to testing each of its components separately. This is a controversial assumption: In most complex systems, modules or components are interdependent. They may share resources (e.g. memory), and have explicit interactions.

  The use of the independency assumption permit to break down the complexity of testing a large system. It is largely used and very often misused. This assumption avoids the consideration of a class of faults that can be captured by considering the interleaving of actions between submodules.

  Example: The case where several connections supported by a protocol entity, see Fig. 8. One may test only one connection in detail (it is in some sense independent of the others). The others need not be tested, since they are all equal (uniformity assumption, see above). The indepen-

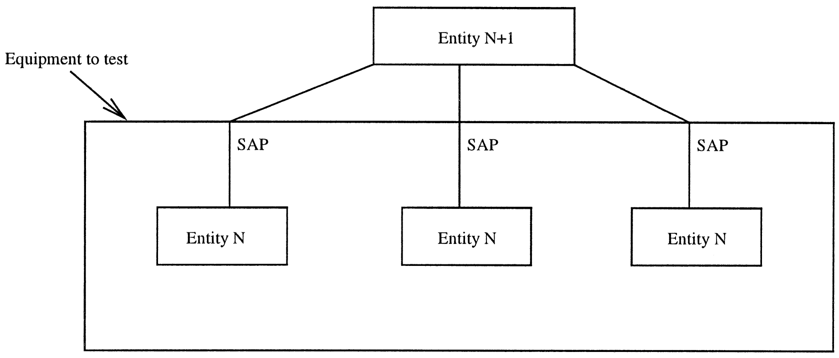


Fig. 8. Independency assumption.

dency relation is a reasonable assumption in certain cases.

- Fairness assumption with respect to nondeterminism: Many systems have a nondeterministic nature. In particular, the parallelism of distributed systems introduces many possible interleavings of individual actions within the different system components. The assumption is that all the execution paths effectively realized during testing cover all paths that are relevant for detecting the possible implementation faults.

The above defined test hypothesis have shed some light on the form of the fault models that can be used for analysing the fault coverage of a given test suite, or for deriving test suites with a given fault coverage. Often fault models are based on a mutation approach: Each faulty implementation (mutant) is obtained from the specification by introducing a localized mutation. The fault model is a kind of add-on to the specification. Examples:

- Output and transfer faults in the FSM model. Often it is useful to introduce additional test cases to check whether the testing assumptions are satisfied.
- Checking the independency assumption for the implementation of a protocol entity by testing a large number of connections in parallel. Note: In the case of overload, independency is often lost; therefore stress testing is very useful.
- Using extreme values in the context of parameter variation testing is a means of checking whether the uniformity assumption is satisfied.
- Running a very long test case may check the regularity assumption.

5. The fault model

The large number and the complexity of physical and software failures dictates that a practical approach to testing should avoid working directly with those physical and software failures. In fact, in most cases, we are mostly concerned with the detection of the presence or absence of any failure. Many failures may very well cause the same error for a given test or set of tests. One method of resolving this problem is by using a fault model to describe the effects of failures at some higher level of abstraction (logic, register transfer, functional blocks, etc.). This implies various tradeoffs between accuracy and ease of modeling and analysis. If the fault model describes the faults accurately, then one needs only to derive tests to detect all the faults in the fault model. This approach has several possible advantages. A higher level fault describes many physical and software faults, thus reducing the number of possibilities to be considered in the generation of tests [10].

In the following, we define a FSM, a fault model for FSM, software fault model and hierarchical fault model.

5.1. The Finite State Machine (FSM) model

Definition 3 (*Finite State Machine*). A Finite State Machine (FSM) M is defined as a tuple $(S, S_0, X, Y, D_s, \delta, \lambda)$, where

- S is a finite set of states,
- $S_0 \in S$ is the initial state,
- X is a finite set of inputs,
- Y is a finite set of outputs,
- $D_s \subseteq S \times X$ is the specification domain,

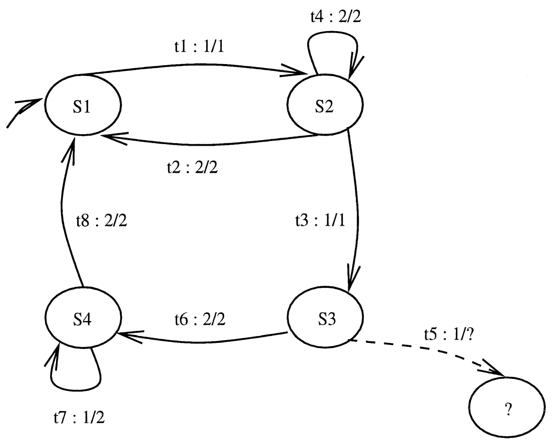


Fig. 9. A partially specified, deterministic and initialized FS.

- $\delta: D_s \to S$ is the transfer function, and
- $\lambda: D_s \to Y$ is the output function.

An example of FSM is depicted in Fig. 9.

5.2. FSM-based fault model

In our FSM model, we assume that the machine is completely specified, that is, the output and next-state functions are defined for all state and input values. The type of faults considered in this model are the following:

- Output fault: We say that a transition has an output fault if, for the corresponding state and input received, the machine provides an output different from the one specified by the output function. In Fig. 10, transition t2 of the implementation I has an output fault e, while transition t2 of the specification S has an output f.
- Transfer fault: We say that a transition has a transfer fault if, for the corresponding state and input received, the machine enters a different state than the one specified by the transfer function. In Fig. 10, transition t1 of the implementation I has a transfer fault to state s0, while the next state of transition t1 of the specification S is s1.
- Transfer faults with additional states: In most cases, one assumes that the number of states of the system is not increased by the presence of faults. (Note that a smaller number of states could be explained by normal transfer faults making a subset of the states unreachable.) Certain types of errors can only be modelled by additional states, together with transfer faults which lead to these additional states. In Fig. 10, transition t6 of the implementation I has a transfer fault to the new state s3, while the next state of Transition t6 of the specification S is state s1.
- Additional or missing transitions: In many cases, it is assumed that the finite state machine is deterministic and completely specified, that is, for each pair of present state and input, there is exactly one specified transition. In the case of incompletely specified machines, no transition may be specified for a given pair, while in the case of non-deterministic machines, more than one transition may be defined. In these cases, the fault model could include additional and/or missing transitions.

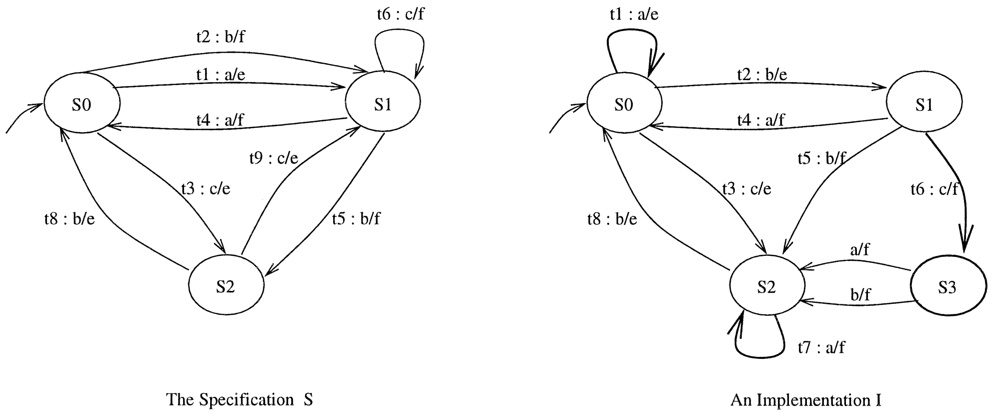


Fig. 10. Example of FSM's.

Furthermore, missing or additional spontaneous transitions (transitions without input) may be considered for non-deterministic machines. In comparison to the specification S, Fig. 10 shows transition t7 of the implementation I as an additional transition, while t9 is missing from I but exists in S.

5.3. Software fault model

There is a very large number of different types of software faults that may be considered. In order to handle this complexity, a high level abstract fault model is desirable. The following list identifies the most important types of software faults. Usually, the software faults are classified into process faults ((a)–(c) below) and data faults ((d)–(g) below). Within this paper, we also use the classification into sequencing faults ((a) below), data manipulation faults ((b)–(d) below) and data flow faults ((e)–(g) below). We note that this list does not include the system errors considered in Ref. [6].

(a) Sequencing faults, such as missing alternative path, improper nesting of loops, WHILE instead of REPEAT, wrong logic expression to control a conditional or looping statement, etc.

(b) Arithmetic and manipulative errors, such as ignoring overflow, wrong operator (e.g. GT instead of GE or + instead of *), etc.

(c) Calling wrong function.

(d) Wrong specification of data type and/or wrong format of data representation.

(e) Wrong initial values, or wrong number of data elements.

(f) Referencing an undefined variable, or the wrong variable.

(g) Defining a variable without subsequent use.

While in the past, control faults and data flow faults were usually considered separately, there have been some recent proposals to combine these two aspects within a single model [85,94]. It is to be noted that the FSM fault model described in subsection above is a special case of a control flow model.

5.4. Hierarchical fault models

Often the specification of a system is hierarchically structured. At the highest level of abstraction, the system is composed of a certain number of components. Each component is then described at a more detailed level possibly again in terms of a composition of subcomponents, and so on. Sometimes, several different descriptions of the same component exist, one in terms of the composition of subcomponents, and one which describes the behavior of the component directly in terms of its interactions with the environment.

Given such a hierarchical system description, the corresponding fault models may be established using these different levels of abstraction. In the simplest case, the following fault model based on the system decomposition into components may be considered. Each component may either be faulty or operating correctly. The interconnection structure specified for the components within the system determines which externally visible interactions of the system may exhibit the erroneous behavior of a given faulty component. This information may be used for the selection of test cases covering all components or for fault location [30].

A more detailed fault model for the specified system may be obtained by considering the specifications of each of the components at the next level of details. If the component is specified in terms of a composition of subcomponents, the same fault model (of faulty and correct behavior) may be considered at the level of the interconnected subcomponents. For instance, the assumption of a single fault of a subcomponent will restrict the type of malfunctions which would be observable at the component level. If the behavior of the component is specified directly in terms of its interaction sequences, one of the fault models described above may be applicable for the component and provide a collection of faults which could be expected to explain a malfunction of the component.

In the following, we address test based on different models.

6. Testing based on finite state models

The research contributions in test derivation and selection are mostly based on the FSM specification of the control aspects of a protocol [19]. In this section, we first list some assumptions. Then, we present some known methods such as, Transition tour (TT-Method) [75], W-method [24], Distinguishing Sequence method (DS method) [45], and Unique-Input-Output method (UIO method) [83]. Many variations of these methods are also given.

6.1. Assumptions

Assumptions that should be made for FSM-based testing can be classified into two classes. The first class of assumptions is about the desirable properties of the specification. The second class of assumptions is about the types of faults (i.e., the fault model [9,74]) that can be present in an implementation. Without the second class of assumptions, any FSM can be considered as an implementation of a given specification, the number of implementation machines will be infinite. This makes the problem intractable. Therefore, assumptions of the second class are introduced to limit the number of implementation machines to be considered [44,59]. An interesting work has been made by researchers to define, explain and show the importance of test hypothesis and assumption for test and coverage estimation, see [42,78,99,23]

For the specification machine, assumptions made are basically about the following structural properties:

- Deterministic or non-deterministic;
- Completeness: if a specification is completely or partially specified;
- Connectedness: if a specification is strongly or initially connected;
- Reducibility: if a specification is reduced or non-reduced.

Assumptions about implementations:

- Deterministic or non-deterministic;
- Completely defined which means that the implementation will react to any input;
- It has a limited number of extra states;
- It has a reliable reset (that is not necessary in some cases).

Many of these assumptions can be avoided (see [99]) and methods have been developed for partially specified and nondeterministic behaviors (see below).

6.2. Test derivation methods

6.2.1. Transition tour (TT-method)

The TT-method [75] generates a test sequence called ‘‘transition tour’’. For a given FSM, a transition tour is a sequence which takes the FSM from an initial state, traverses every transition at least once, and returns to the initial state.

The T-method allows the detection of all output errors but there is no guarantee that all transfer errors can be detected. This method has a limited error detection power compared to other methods since it does not consider state checking. However, an advantage of this method is that the test sequences obtained are usually shorter than test sequences generated by the other methods.

To further optimize a Transition Tour, we find the shortest path through the automaton which covers all transitions (variation of the so-called ‘‘Chinese Postman algorithm’’).

In the example of Fig. 9, no transition needs to be traversed twice. A possible transition tour is formed by the input sequence 1.2.1.2.1.2.2.

Clearly, the final state of the last transition is not checked. And if the previous transition, t8, leads to state s2, this fault would not be detected either.

In order to systematically detect transfer faults, one has to identify the state into which the implementation goes after the execution of the tested transition.

6.2.2. The UIO method

The UIO method [83] is quite simple and produces a variable-length state identification sequence. This method can be applied if for each state of the specification, there is an input sequence such that the output produced by the machine, when it is initially in the given state, is different than that of all other states. If a transition is supposed to lead to state s1, it suffices to apply the UIO input sequence of state s1 and check that the output is as expected.

As an example, an FSM with its UIO sequences is shown in Fig. 11 and Fig. 12, respectively. Certain methods propose the concatenation and overlapping of the sequences belonging to different test cases. If a given state possess several UIO sequences, one may choose the one which leads to optimized concatenation possibilities. However, fault detection guarantees are usually lost by such approaches.

The UIO method does not guarantee full fault coverage (with respect to the fault model of implementations having at most *n* states). However, Vuong found a counter-example [95].

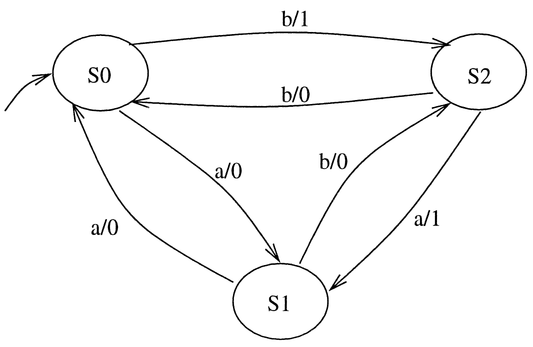


Fig. 11. An example of FSM specification.

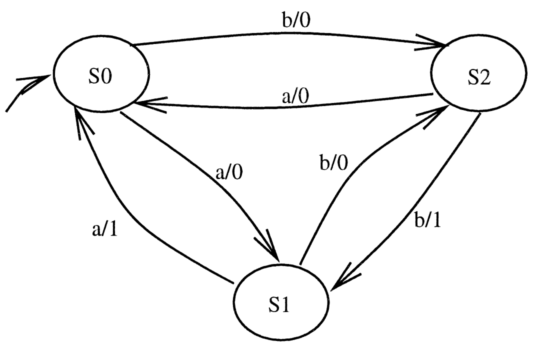


Fig. 13. An example of FSM specification.

The reason for such an undetected fault is that the UIO input sequence leads to unique output for the specification, but not for the faulty implementation (certain multiple faults compensate themselves partly). The solution is to check the uniqueness of the applied identification sequences on the implementation, meaning that each identification sequence must be applied on each state of the implementation and the outputs are compared with those expected from the specification.

Checking the identification sequences is realized by various methods, such as the following: UIOv-method, DS-method, W-method, Wp-method and HSI-method.

6.2.3. DS-method [45]

A distinguishing sequence (DS) is used as a state identification sequence. An input sequence is a DS for an FSM, if the output sequence produced by the FSM is different when the input sequence is applied to each different state. The test sequences obtained by the DS method guarantee to identify a particular FSM from all other FSMs. It has a full fault coverage (detecting both transfer and output errors). However, the disadvantage of this method is that a DS may not be found for a given FSM (as one single sequence is the UIO for all states). Also applying a fixed-length sequence may not lead to the shortest state identification sequence.

As an example, Fig. 13 and Fig. 14 show an FSM that has a distinguishing sequence DS = a.a and the test cases generated by the DS-Method, respectively.

6.2.4. W-method [24]

The W-set is a set of sequences which allows to distinguish all states. Since all sequences must be applied to the state to be identified, it is necessary to execute the same transition to be tested several times, one for each sequence to be applied. The W-method involves two sets of input sequences: one is the W-set, the other is the P-set. The W-set is a characteristic set of the minimal FSM, and consists of input sequences that can distinguish between the behaviors of every pair of states. This set is sometimes represented by W, hence the W-method. A method for constructing minimal W-sets can be found in Ref. [44]. We write P for any set of input sequences (including the empty sequence) such that for each transition from state A to state B on input x, there are input sequences p and p.x in P such that p takes the machine from the initial state into state A. This

State S_i	UIO(S_i)	Preamble(S_i)
S_0	b/1	Null
S_1	a/0.b/1	a/0
S_2	a/1	b/1

Fig. 12. An UIO sequence for the FSM of Fig. 11.

State S_i	Output For DS = a.a	Preamble(S_i)
S_0	0.1	Null
S_1	1.0	a/0
S_2	0.0	b/0

Fig. 14. A DS sequence for the FSM of Fig. 13.

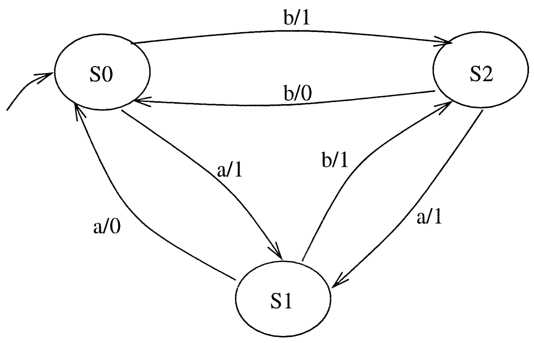


Fig. 15. An example of FSM specification.

means, the set P (or P-set) consists of all partial paths.

The W-method gives a set of test sequences formed by the concatenation of the W-set and P-set. Each test sequence starts with the initial state and returns to it again afterwards. It is also guaranteed to detect any misbehavior of the machine.

The W-set is constructed using a special method, and the P-set can be formed from a testing tree, which shows every transition from state i to state j on each input.

The W-method applies the W-set, which consists of a number of input sequences such that the last output symbols observed by applying the strings in W in the same order are different for each state of the FSM [24,87].

The assumptions about the machine under test are that the machine is minimal, may start in a fixed initial state, and is strongly connected. Under these assumptions a W-set exists, and the W-method gives a set of sequences that are guaranteed to detect any misbehavior of the machine. However, the limitation of using this method is that it is not certain that every FSM will have a W-set sequence, especially if it is an incompletely specified machine. Therefore, before using this method, one should first make sure that the defined machine has a W-set sequence. When a machine does not have a W-set sequence, a procedure can take place to form a completely specified machine which has a W-set, for example, adding an 'error' state and declaring all unspecified transitions to lead to this state.

Fig. 15 and Fig. 16 show an FSM and the resulting test cases generated by the application of W-Method, respectively.

6.2.5. *Wp-method [40]*

This is a generalization of UIOv method which is always applicable. It is at the same time an optimization of the W-method. The main advantage of the Wp method over the W method is to reduce the length of the test suite. Indeed, instead of using the set W to check each reached state S_i, only a subset of W is used in certain cases. This subset W_i depends on the reached state S_i and is called *an identification set* of S_i. An identification set of a state S_i is a *minimal* input sequence W_i such that for each state $S_j \in S$ with $(i \neq j)$ there exists an input sequence of W_i for which S_i and S_j produce two *different* output sequences. The Wp method consists of two phases with the following purposes:

- The first phase checks that all states defined by the specification are identifiable in the implementation. Moreover, the transitions leading from the initial state to these states are checked for correct output.
- The second phase checks that the remaining transitions are correctly implemented.

As an example, let us consider again the FSM given in Fig. 15. The application of the Wp-Method to this FSM leads to the sets P, Q and W_i $(i = 0,1,2)$ and their corresponding outputs as shown in Fig. 17. Instead of using the set $W = \{a,b\}$ to check the reached states S_1 an S_2, we use only the subsets

State S_i	Output For W ={a, b}	Preamble(S_i)
S_0	{1, 1}	Null
S_1	{0, 1}	a/1
S_2	{1, 0}	b/1

Fig. 16. A W-set for the FSM of Fig. 15.

State S_i	Output For W ={a, b}	Preamble(S_i)	Set W_i	Output for W_i
S_0	{1, 1}	Null	{a, b}	{1, 1}
S_1	{0, 1}	a/1	{a}	{0}
S_2	{1, 0}	b/1	{b}	{0}

Fig. 17. W and Wi Sets for the FSM of Fig. 15

$W_1=\{a\}$ and $W_2=\{b\}$ to verify these states. However, we use the entire set $W=\{a,b\}$ to check the reached state S_0. Note that most of these methods have been generalized to be able to provide full coverage guarantee also for a fault model of implementations with less than m states, where $m\geq n$. However, the length of the required test suite increases strongly as m increases [99].

6.3. Optimization techniques

This group of testing techniques is based on the UIO sequences [1,86,98,100]. These methods have a better fault coverage than the TT method. A single test case generated by such a method will not only traverse all the transitions (to detect all output faults), but also somehow checks the ending state of each transition (and therefore can detect some transfer faults). The general principles underlying these methods are to

1. construct a test subsequence for each transition specified in the specification. A test subsequence is formed by the input symbol of the transition under test followed by the input sequence of the UIO for the ending state of that transition; and
2. find a single optimal test case which traverses each of the test subsequences at least once, and if possible at most once by using the Rural Chinese Postman (RCP) tour algorithm.

This general approach can be enhanced by using multiple UIOs and overlapping test subsequences [86,98,100] to obtain an even shorter test case. However, these optimization techniques cannot guarantee complete fault coverage for the implementation class with n states, i.e., a single test case generated in such a way can sometimes fail to detect a non-conforming implementation. As an example, let us consider the FSM specification given in Fig. 18a. We use the UIO sequences x/1, x/0.x/1 and y/1.x/1 for the three states S1, S2 and S3, respectively, to form the transition test subsequences nt1, nt2, nt3, nt4, nt5 and nt6 given in Fig. 19. By using the above general approach, we generate a single test case which is also given in Fig. 19. It is easy to verify that this test case traverses each of the six transition test subsequences once. However, a faulty implementation modeled by the FSM given in Fig. 18b can still pass this test case. (The same problem also exists for the multiple testing approach). The specification FSM in Fig. 18a and the implementation FSM in Fig. 18b were used in Ref. [95] to show that the UIO-method cannot guarantee complete fault coverage.)

The reason that this sort of problems may happen is that a UIO sequence derived from a given specification may no longer be a UIO sequence in a faulty implementation [95]. As in the above example, the UIO ‘‘y/1.x/1’’ for state S3 in Fig. 18a is no longer a UIO sequence for the corresponding state I3

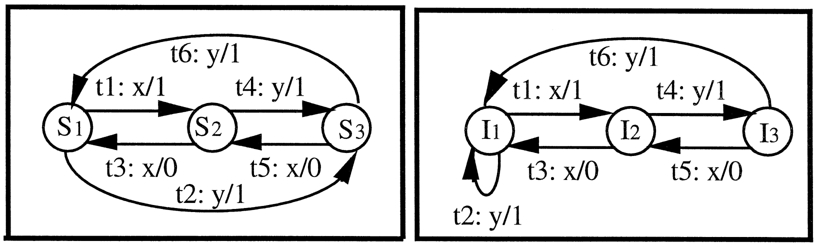


Fig. 18. (a) FSM specification, (b) an implementation of the FSM.

n t1: [1] x [2] x [1] x [2]   n t2: [1] y [3] y [1] x [2]   n t3: [2] x [1] x [2]

n t4: [2] y [3] y [1] x [2]   n t5: [3] x [2] x [1] x [2]   n t6: [3] y [1] x [2]

TS = [1] x [2] x [1] x [2] x [1] x [2] y [3] y [1] x [2] x [1] y [3] y [1] x
[2] y [3] x [2] x [1] x [2] y [3] y [1] x [2]

Fig. 19. Optimized test case.

in Fig. 18b, since both states I1 and I3 possess this I/O sequence. Note that the integer in a pair of square brackets in an input sequence indicates a state number. For instance, [1]x[2]x[1] means that the input sequence starts from state S1, passes state S2 and ends at state S1. Individual test cases may be concatenated into a single, large test case, thus reducing the total number of test inputs required. Such approaches are often called optimization techniques.

7. Test development for extended FSM specifications (e.g. SDL)

7.1. The Extended FSM model

The basic Extended FSM (EFSM) model describes a module (process) as a basic FSM extended by the following:

- Interactions have certain parameters, which are typed.
- The module has a certain number of local variables, which are typed.
- Each transition is associated with an enabling predicate, which depends on the effective parameters of the received input and the current values of the local variable, with an action, which is executed when (and if) the transition is fired and which may update the local variables, and for each output generated, there is an expression for each associated parameter which determines the parameter value as a function of the local variables and the input parameters.

Formally, an EFSM is described by a tuple $M = (S, s_0, I, O, T, V)$ where

- S is an nonempty set of states the process can be in,
- s_0 is the initial state ($s_0 \in S$),
- I is an nonempty set of input interactions,
- O is an nonempty set of output interactions,
- T is an nonempty set of transitions,
- $\delta: S \times (I \cup O) \rightarrow S$ is a transition relation.

Each element of T is a tuple $t = (s_i, s_j, i, p, b)$. Here, s_i and s_j are the states of S representing the starting state and the tail state of t, respectively. i is either an input interaction from I or empty. p is a predicate expressed in terms of the variables in V, the parameters of the input interaction and some constants. b is a set of assignment and output statements.

This basic model underlies many specification languages, including SDL, Estelle, StateCharts, and many state-oriented notations for describing the behavior of object-oriented systems.

The basic problem that the test generation encounters for this model is the executability problem. This is equivalent to finding a path for which all transitions are executable, meaning that all predicates are satisfied along the path. This problem is known to be undecidable in general.

Since the EFSM specification model combines the FSM and programming language approaches to specification, the corresponding testing methods can be combined for testing with respect to EFSM reference specifications. Typically, one of the FSM methods is combined with data flow criteria and parameter variation methods. Sometimes, the control flow of the action associated with a transition is ''normalized'' in order to be closer to the FSM flow control model.

The main problems are that the control flow is not independent of the data. The question of what input sequence to apply for leading the IUT to the execution of a given transition is undecidable (that is, it is impossible to find an algorithm which returns such an input sequence or returns the message ‘‘this transition is not executable’’). Therefore, heuristics are needed. (Note: The same problem is true for software testing, in general).

7.2. Data flow analysis

This technique originated from attempts for checking the effects of testing data objects in software engineering. It is usually based on a data flow graph which is a directed graph whose nodes representing the functional units of a program and the edges representing the flow of data objects. The functional unit could be a statement, a transition, a procedure or a program. In the context of EFSM modeled communications protocols, data flow analysis analyzes the data part of the EFSM in order to find data dependencies among the transitions. It usually uses a data-flow graph where the vertices represent transitions and the edges represent data and control dependencies. The objective is to test the dependencies between each definition of a variable and its subsequent use(s).

Definition 4.
A transition T has an assignment-use or A-Use of variable x if x appears at the left-hand side of an assignment statement in T. When a variable x appears in the input list of T, T is said to have an input-use or I-Use of variable x. If a variable x appears in the predicate expression of T, T has a predicate-use or P-Use of variable x. T is said to have a computational-use or C-Use of variable x if x occurs in an output primitive or an assignment statement (at the right-hand side). A variable x has a definition-use (referred to as def-use) if x has an A-Use or I-Use.

We now define some sets needed for the construction of the path selection criteria.

Definition 5.
def(i) is the set of variables for which node i contains a definition, C-Use(i) is the set of variables for which node i contains a C-use and P-Use(i,j) is the set of variables for which edge (i,j) contains a P-use. A path $(t_1,t_2,\ldots,t_k,t_n)$ is a def-clear-path with respect to (w.r.t.) a variable x if the path $t_2,\ldots,t_k$ do not contain definitions of x. A path $(t_1,\ldots,t_k)$ is a du-path w.r.t. a variable x if and either or, and $(t_1,\ldots,t_k)$ is a def-clear path w.r.t. x from t_1 to t_k.

Obviously, when selecting a criterion, there is a trade-off. The stronger the selected criterion, the more closely the program is scrutinized in an attempt to detect program faults. However, a weaker criterion can be fulfilled, in general, using fewer test cases. As the strongest criterion all-paths can be very costly, we will use the second strongest criterion all-du-paths. P satisfies the all-du-paths criterion if for every node i and every x, P includes every du-path w.r.t x. For a complete list of the selection criteria, refer to [96].

The main difference between the ‘‘all definition-use’’ or ‘‘all du’’ criterion and a fault model such as the FSM fault model is the following: in the case of the ‘‘all du’’, the objective is to satisfy the criterion by generating test cases that exercise the paths corresponding to it. Exercising the paths does not guarantee the detection of existing faults because of variable values that should be selected. If the right values are selected then certain ‘‘du’’ criteria are comparable to fault models.

7.3. Handling the executability of the test cases

The executability problem is in general undecidable. However, in most cases, it can be solved.

7.3.1. Control and data flow testing

In the EFSM model, the traditional methods for testing FSMs such as UIO sequences, distinguishing sequences (DS), or W-Method are no longer adequate. The extended data portion has to be tested also to determine the behaviors of the implementation. Recently, there has been some work on data flow testing of protocols [85,94,97]. However, they have focused on data flow analysis and control flow has been ignored or considered separately, and they don’t consider the executability problems. As to control flow test, applying the FSM-based test generation methods to EFSM-based protocols may result in non-executable test sequences. The main reason is the existence of predicates and conditional statements.

In Ref. [94], the authors presented a method for the automated selection of test sequences for testing both control flow and data flow aspects of a protocol. As mentioned before, the selection of test sequences is based on the identification and subsequent coverage of every association between an output and those input that influence that output. The method requires that each such association is examined at least once during testing. The authors stated that the application of the IO-df-Chain criterion results in a test sequence which covers every transition at least once. However, we tried this method on an EFSM, and the results showed that one transition was not covered. The problems of building test sequences which cover the IO-df-Chains and checking their executability are left to the user.

In Ref. [54], the authors presented an executable data flow and control flow protocol test sequence generation techniques for EFSM-specified protocols. In the data flow part, the transition paths that contain definition uses and output uses of variables in the protocol specifications are detected and tested. An executable test sequence (ETS) contains three parts: (1) the executable switching sequence EDSS (or ECSS) which can reach DO-paths, is generated by expanding Transition Executability Analysis (TEA) trees rooted from the EFSM's initial state configuration, (2) the executable DO-path (EDO-path) or the executable control path (EC-path) which is generated by expanding TEA trees rooted from the state configurations of the tail states of EDSSs, and (3) the executable back path (EBP-path) which is derived by expanding a TEA tree rooted from the tail state of the EDO path. The DO-path is a definition-output path (the same as in Ref. [94]), the EDSS (or ECSS) is the preamble and the EBP-path is the postamble. In this technique, all derived sequences are executable, but this technique is a kind of reachability analysis for EFSMs and hence has the same disadvantages (i.e., state explosion). Also, to derive the executable test sequences, one must instantiate the input parameters. The generated test sequences vary according to the values given to the input parameters.

Ref. [80] presented a unified test case generation method for EFSM-specified protocols using the context independent unique sequences. This method considers the feasibility of the test cases while they are being generated. A new type of state identification sequence, namely, the context independent unique sequence (CIUS) is defined. The trans-CIUS-set criterion used in the control flow test case generation is superior to the existing control flow coverage criteria for EFSMs. In order to provide observability, the ''all-uses'' data flow coverage criterion is extended to what is called the ''def-use-ob'' criterion. Finally, a two-phase breadth-first algorithm is designed for generating a set of executable test tours for covering the selected criteria. An interesting method, described in Ref. [21], uses data flow analysis techniques, all ''definition-use'' paths ([96]), to find data and control dependencies among the transitions of the EFSM. The testing task is then to test these dependencies. For the purpose of data flow testing, it suffices to determine and test the relations among the variables in the protocol specification. The algorithm presented generates all control and data dependencies between transitions, the corresponding def-clear paths and the linking variables. Then, subsequences which cover all du-paths (definition-use) and all transitions (and states) are generated.

7.3.2. Test data selection approaches

One may observe that in EFSM test sequence generation, some subsequences may not be executable because the transition enabling predicates (also called constraints) along the path cannot be satisfied for any inputs. Test data selection is the critical step in testing. Test data sets must contain not only input to exercise the implementation, but also provide the corresponding correct output responses to the test data inputs. Thus the development of test data sets involves two aspects: the selection of data input and the determination of the expected response. Often the second aspect is most difficult.

To our knowledge, the two techniques that are used for test data selection are CLP techniques (which use symbolic evaluation) and mutation analysis. But first, lets define symbolic execution because it is the technique used to generate the constraints along paths to be tested.

7.3.2.1. Symbolic execution. Symbolic execution [27,51,58] is a program analysis method that represents a program's computations and domain by symbolic expressions. It derives an algebraic representation over the input values of the computations and their applicable domain. Thus symbolic evaluation describes the relationship between the input data and the resulting values, whereas normal execution computes numeric values but loses information about the way in which these numeric values were derived. There are three basic methods of symbolic evaluation which differ primarily in their techniques for selecting the paths to be analyzed: path-dependent evaluation, dynamic symbolic evaluation, and global symbolic evaluation.

Symbolically executing a program is closely related to the normal notion of program execution. It offers the advantage that one symbolic execution may present a large, usually infinite, class of normal executions. Formal verification methods use symbolic evaluation techniques to formulate the verification conditions that must be proven. Typically, input, output, and loop invariant assertions are supplied. Verification conditions are then created by symbolically evaluating the code between two adjacent assertions.

Support of the path selection process is a natural application of symbolic evaluation (data flow techniques). The symbolic representation created by symbolic evaluation can be quite useful in determining what test data should be selected in order to have confidence in a path's reliability. Symbolic evaluation can also be used in program debugging, program optimization and software development (program requirements can be expressed in terms of symbolic representations).

7.3.2.2. Constraint satisfaction problem and constraint logic programming. Techniques for solving the constraint satisfaction problems (CSP) [31] have been an active research area in the AI community for many years. Its application has extended to many other areas such as operations research and hardware design. A CSP involves a set of n variables $X_1, \ldots, X_n$, each represented by its domain values $R_1, \ldots, R_n$ and a set of constraints. A constraint $C_i(X_{i1}, \ldots, X_{ij})$ is a subset of the Cartesian product $R_{i1} \times \cdots \times R_{ij}$ which specifies which values of the variables are compatible with each other. A solution is an assignment of values to all the variables which satisfy all the constraints and the task is to find one or all solutions. CSPs are in general NP-complete problems. However, by exploiting domain knowledge and manipulating constraints cleverly, researchers [39,73,68] have shown that CSPs can be solved efficiently.

CSPs can be solved using CLPs (Constraint Logic Programming) [79]. As the name suggests, CLP is a descendent of logic programming, which was famous for the Prolog language as a consequence of the Japanese ‘‘5th Generation’’ project and the expert systems boom of the mid-80's. Now CLP languages make logic programs execute very efficiently by focussing on a particular problem domain.

In a CLP language, the purely abstract logical framework is supplemented by objects that have meaning in an application domain – for example the integers or the real numbers, along with their associated algebraic operations (e.g. addition and multiplication) and predicates (e.g. $=$, $<$, and $>$). Hence there isn't a single CLP language, but a whole family of them defined for different application domains. A CLP programmer introduces arithmetic expressions called ‘‘constraints’’ (e.g. $X > 0$ or $Y + Z < 15$) into programs, which have to be satisfied for successful execution of the program.

CLP does mathematics with uninstantiated variables, so that in the absence of complete information the answer might be a symbolic expression like $10 - X$, or even a constraint like $X > 23$. A CLP program still needs to search a database of facts, but it can use constraints to rule out many possible outcomes (i.e. to prune away large parts of the search tree) resulting in enormously improved efficiency, comparable to custom solutions written in C.

The founding work on the CLP scheme was done at Monash university in Melbourne, Australia, by Jaffar and Lassez [57]. They created CLP(R) system which works on the domain of real linear arithmetic. In Europe, a language called CHIP was developed at the ECRC (European Computer-Industry Research Centre). CHIP provides constraint solvers over finite arithmetic, linear rational, and boolean domains. In 1990, Alain Colmerauer created Prolog III, a CLP language over the domains of linear rational arithmetic, booleans, and finite strings or lists. Interesting non-Prolog-based CLP languages include ‘‘Trilogy’’, from the Vancouver-based Complete Logic Systems, and ‘‘Oz’’, an object-oriented concurrent CLP being developed at DFKI (German Research Center for Artificial Intelligence) in Kaiserlautern.

7.3.2.3. Mutation analysis. Mutation analysis [32,70] is a fault-based method that measures the adequacy of a set of externally created test cases. In practice, a tester interacts with an automated mutation system to determine and improve the adequacy of a test data set. This is done by forcing the tester to test for specific types of faults. These faults are represented as simple syntactic changes to the test program that produces mutant programs. The goal of the tester during mutation analysis is to create test cases that differentiate each mutant program from the original program by causing the mutant to produce different output. In other words, the tester attempts to select inputs that cause each mutant to fail. When the output of a mutant program differs from the original program on some input, that mutant is considered dead and is not executed against subsequent test cases. A test case that kills all mutants is adequate relative to those mutants.

After all mutants have been executed, the tester is left with two kinds of information. The proportion of the mutants that die indicates how well the program has been tested. The live mutants indicate inadequacies in the current test (and potential faults in the program). The tester must add additional test cases to kill the remaining live mutants.

It is generally impossible to kill all mutants of a program because some changes have no effect on the functional behavior of the program.

7.3.3. Test data selection methods

Ref. [94] addressed the test data selection problem, but didn’t completely solve it. The authors stated that a particular selection of values for parameter fields of input interactions in a given test subsequence is subject to two major constraints requiring human intervention. The first constraint is that only some representative values from a relatively large ranges of values for interaction parameter fields should be selected. The second constraint for a particular selection of values for parameter fields of input interactions in a given test subsequence is that the values must satisfy the predicates associated with the state transitions covered by the test sequence.

In Ref. [48], only integer and boolean parameter types are considered and operators are restricted to ‘‘+’’ and ‘‘-’’. Hence, integer linear programming techniques are employed to find solutions for the constraints.

Ref. [25] used CLP techniques to solve the executability problem. To generate only executable test cases, a sequence of transitions, such as UIO as in FSM-based techniques, should be constructed while keeping conditions satisfiable. However, since the test case generation is based on the specification, not on the implementation, the concrete values of a context cannot be determined during test case generation. Symbolic evaluation of each variable in a specification is used instead of the concrete value. In this case, the condition cannot be checked against current values of variables. Thus, the constraint concept is introduced. A constraint is a set of relations which define valid ranges of arguments of input interactions. Initially, the constraint is true. As a transition is selected and appended to a sequence being generated, a constraint dependent on the transition is added.

To generate only executable sequences, the CSP under a given context and arguments must be solved. With finite domains, checking the feasibility of a sequence can be solved by using CLP [57]. Although the problem is NP-complete, it can be used in the context of communication protocols due to their simplicity.

A tool for generating test cases from TOF (Test Oriented Form), an intermediate form whose modeling power lies between that of a pure FSM and that of Estelle, has been implemented.

In Ref. [70], after test sequences are generated, test data for each sequence is chosen using a weak mutation technique to guarantee detection of specific kinds of faults in the data flow. Weak mutation technique was adopted because of its advantages regarding the choice of a test data set. Also, in this method, as mentioned previously, all possible def-ob (definition-observation) paths and conditional paths are traversed, which guarantees a better fault coverage than traversing a subset of such paths. This approach generates an executable test sequence and also guarantees a high degree of fault coverage.

Ref. [32] presents a technique for automatically generating test data. The technique is based on mutation analysis and creates test data that approximates relative adequacy. The technique is a fault-based technique that uses algebraic constraints to describe test cases designed to find particular types of faults. A set of tools, collectively called Godzilla, has been implemented to automatically generate constraints and solves them to create test cases for unit and module testing. This set of tools has been used as an effective way to generate test data that kill program mutants.

Ref. [21] used transition loop analysis ([27]) for influencing self-loops and the CSP method to solve the executability problem. Test data selection is not mentioned. However, in Ref. [22], an automatic protocol test case generator that generates both test sequence and test data is presented. First, test sequences are generated using the method described in Ref. [21]. A set of path conditions associated with each test sequence is obtained using symbolic execution techniques. By solving the path conditions as a group of constraints (input constraints), test data are then automatically generated.

A heuristic to find a solution for a constraint system is proposed. The procedure repeatedly assigns values, randomly chosen from the variable's domain, to each variable until a solution is found. When a dead-end is encountered (i.e., a given value of the variable x cannot be used to satisfy all occurrences of x), other values are tried. This procedure is repeated until all constraints are satisfied or a maximum number of trials is performed. In that case, the constraint system is assumed to be unsolvable.

Ref. [81] used the same technique for test data selection as the one described in Ref. [22] to generate test data for Fortran programs.

8. Testing real-time properties

8.1. Introduction and formal model

The last two decades have witnessed a lot of research activity in the area of untimed black box conformance testing for communication protocols. However, nowadays, protocol communications are increasingly involved in safety-critical and real-time systems, such as patient monitoring systems, plant control systems, air traffic control systems, etc. Moreover, we witness the rapid development and deployment of new time dependent applications such as multimedia communications. All these systems are commonly specified with time constraints to control their behaviors. Since the functional misbehavior of these systems is usually due to the unsatisfiability of time constraints, testing such systems becomes an unavoidable issue.

There are two types of real-time properties:
- Hard real-time properties, which state that certain actions must occur within certain time bounds; usually, there is a minimum and maximum delay associated with the action, and
- Performance properties, which usually state properties of a statistical nature, such as average and standard deviation of the response time of the system or maximum throughput obtainable.

The performance properties are usually not difficult to test, except for high-performance systems for which the testing equipment must also support such high performances. We do not consider these issues here.

For the systematic testing of hard real-time properties, one has to select an appropriate fault model which would typically be associated with the specification formalism used to define the real-time properties.

There are different specification formalisms in use:
- Introducing a real-time variable, sometimes called NOW, the value of which always represents the real time.
- Timers: these are independent processes which may be started and stopped, and which generate a timeout interaction after a given time has elapsed after being started. (SDL [20] combines the NOW variable with timers.)
- Various versions of timed automata and Petri nets exists, where minimum and maximum times may be associated with states and/or transitions.
- Extension of classical temporal logic to incorporate timing aspects [69].

In the following, we focus only on timed automata model since it is very suitable for describing real-time systems, and therefore is very popular among researchers for testing and verifying timed systems.

The timed automata model gives rise to different models depending on the domain of time variables, the semantic of time, and the form of time constraints. If we try to make a distinction based only on the time domain, the time models can be grouped into three major categories [33], *discrete time model*, *fictitious clock model*, and *continuous time model*.

In the discrete time model, time is isomorphic to natural numbers. This model has the advantage of being easily transformable onto an FSM model by extending the alphabet with a silent event, *Next Time*, to indicate the passage of time from t to $t+1$. So, we can apply all existing FSM's techniques to this model. However, for asynchronous systems, we have to choose an adequate time quantum to avoid any error.

In the fictitious clock model, the time is continuous, but timing assertions are made by comparing with a fictitious global clock that ticks at some known, fixed rate. The limitation of this model is that the timing information is not exact.

The continuous time model is more general than the above ones in the sense that it can describe any timed system that can not be specified in the other models. In this model, time is dense with no restriction. The density of time gives rise to an infinite state space and, as a consequence, testing such model becomes very difficult.

In the following, we address the issue of testing real-time systems described in a continuous time model. First, we begin by defining the *Timed Input Output Automata* (TIOA for short) model. Secondly, we introduce the fault model in the timed setting. Thirdly, we present the timed test cases generation methods.

Definition 6 (*Timed Input Output Automata*). A Timed Input Output Automaton (TIOA) A is defined as a tuple $(\Sigma_A, L_A, l_A^0, C_A, T_A)$, where

- Σ_A is a finite alphabet composed of input actions beginning with "?" and denoted by I_A, and output actions beginning with "!" and denoted by O_A,
- L_A is a finite set of locations,
- $l_A^0 \in L_A$ is the initial location,
- C_A is a finite set of clocks all initialized to zero in l_A^0,
- $T_A \subseteq L_A \times L_A \times \Sigma_A \times 2^{C_A} \times \Phi(C_A)$ is the set of transitions.

A tuple $(l, l', \{?,!\}a, \lambda, G) \in T_A$, denoted in the rest of paper with $l \rightarrow^{\{?,!\}a,\lambda,G}{}_A l'$, represents a transition from location l to location l' on action $\{?,!\}a$. The subset $\lambda \subseteq C_A$ gives the clocks to be reset with this transition, and $G \in \Phi(C_A)$ is a clock guard (time constraint) for the execution of the transition. The term $\Phi(C_A)$ denotes the set of all guards over C_A, built using boolean conjunction over atomic formulas of the form $x < m$, $x > m$, $x = m$, and $x \le m$, where $x \in C_A$ and $m \in N$. The operator $\le$ is particularly used in output action constraints. The choice of naturals as bounds in constraints helps us, later, to discretize the set of reals into integer intervals reducing thereby the state space of timed systems.

As an example, we consider the 1-clock Timed Input Output Automaton given in Fig. 20. The automaton has two locations named l_0 and l_1, one clock, x, and two transitions. The transition from l_1 to l_0 has *Off* as an output action, and a guard condition $x \le 1$, while the transition from l_0 to l_1 can execute on input *On* at any time, and resets to zero clock x.

Informally, the system starts at location l_0 with its clock initialized to zero. The value of the clock increases continuously and measures the amount of time elapsed since its last initialization or reset. At any time, the automaton can change its current location l to l' by making a transition (l, l', a, λ, G) provided the current value of the clock satisfies G.

To give the semantic of a TIOA, we define a state of a TIOA as a pair (l, υ) consisting of a location $l \in L_A$ and a configuration of the clock values υ that assigns to each clock x a real value $\upsilon(x) \ge 0$. The set of all states is denoted by S_A. Furthermore, we

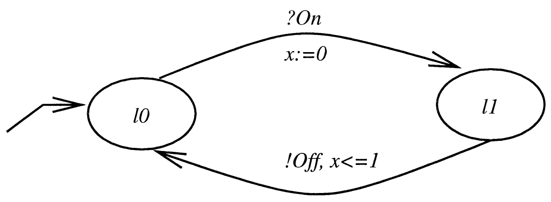


Fig. 20. An example of TIOA.

distinguish between two types of transitions in a TIOA:

- Elapse of time: being in a state, when time progresses, the automaton changes automatically its state. These transitions are referred to as delay transitions.
- Explicit transitions: starting from a state, as time progresses, the automaton makes many transitions of the first type, and when it reaches a state where the guard condition of an outgoing transition is satisfied, the automaton may execute the transition.

The semantic model of a TIOA A is given by a timed labeled transition system $S_t(A)$ that consists of the state set S_A, the label set $R^{\geq 0} \cup \Sigma_A$, both input/output actions and time increments, and the transition relation $\rightarrow^a$, for $a \in R^{\geq 0} \cup \Sigma_A$.

Since the timed labeled transition system $S_t(A)$ is infinite, due to the infinity delay transitions, we can not deal with it to generate test cases. The challenge is therefore to reduce the number of states in the system. To achieve this, an equivalence relation is defined on the set S_t in order to cluster equivalent states into equivalent classes. The resulting timed labeled transition system is called region graph [4,5]. For example, the TIOA of Fig. 20 gives rise to the region graph shown in Fig. 21.

In addition to the TIOA model presented above there are many like models [...refs...] that are used as basis for timed test cases generation methods and that differ from the TIOA on the semantic of time and the form of time constraints.

Once the basic timed models are presented, we point out now the fault model in the case of timed systems.

8.2. Fault model for timed systems

The faults that may arise in an implementation of timed systems can be classified into two categories:

- Timing faults: these faults are due to the non respect of the time constraints under which the timed system must make its transitions.
- Input/Output action faults: this category concerns the faults related to both input and output actions, and is similar to the well known fault model of the FSM and LTS models (see Section 5).

Under the timing faults fall the restriction and the enlargement of the time interval in which an input (respectively an output) action must be performed. This leads to the modification of the interval bounds. With the enlargement of the time interval, we mean that the specification says that an input (respectively an output) action must occur (respectively, be produced) after time B_i and before time B_j, with $B_i \leq B_j$, and the implementation accepts the input (respectively answers with the output) action outside the specified bounds (i.e. before the time B_i or after the time B_j). In such situation, the implementation is considered as a faulty implementation. As an example, let us consider again the specification given in Fig. 20. So, any implementation of this specification that produces the output action *Off* more than 1 unit

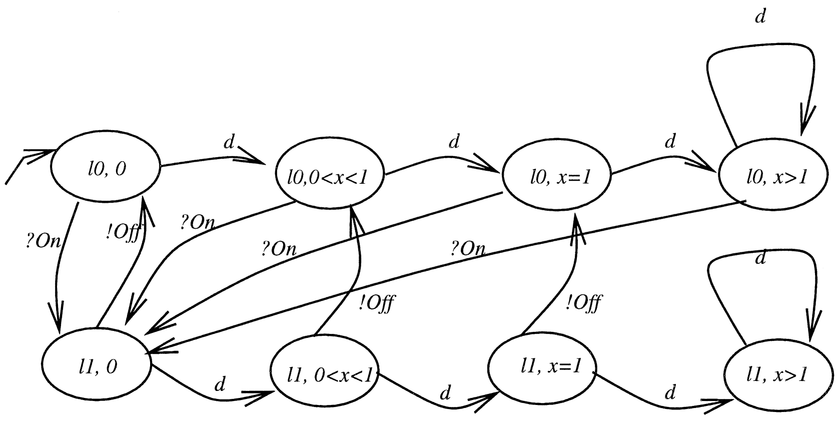


Fig. 21. The region graph of TIOA.

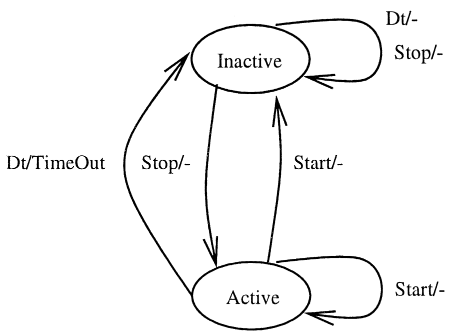


Fig. 22. The timer FSM.

of time after it received an input action *On* is considered faulty.

On the other hand, the restriction means that the specification requires that an input (respectively, an output) action must occur (respectively be produced) after time B_i and before time B_j, with $B_i \leq B_j$, and the implementation accepts the input (respectively answers with the output) action not in all the points between B_i and B_j (i.e. after a time B_k and before a time B_l, with $B_i \leq B_k, B_l \leq B_j, B_k \leq B_l$, and $(B_i, B_j) \neq (B_k, B_l)$). In this case, we distinguish between input and output actions. While the restriction is considered as a fault for input action, it is not considered so for output actions, but it is seen as a valid reduction. For example, any implementation of the specification given in Fig. 20 that produces the output action *Off* no more than $1/2$ unit of time after it received an input action *On* is considered a valid reduction and is therefore a correct implementation. But, any implementation that accepts the input action *On* only before 2 units of time after it was in its initial location is considered faulty.

8.3. Timed test cases generation methods

In the following, we present a survey of the existing timed test cases generation methods.

8.3.1. Test cases generation for FSM with timers and counters

In this work [64], the author adapted the Wp-method (see Section 6.2.5) to generate test cases from a specification given as an FSM with timers and counters. For this purpose, he first draws the FSMs to represent the behavior of the timer and the counter. Then, the three FSMs are combined to obtain the FSM product. Finally the Wp-method is applied on the resulting FSM. Despite the good coverage of the Wp-method, this approach does not deal with a general case of timed specifications.

As an example, let us consider the INRES protocol [49]. The Timer and Counter FSMs are given in Fig. 22 and Fig. 23 respectively. The timer's FSM has two states *{active, inactive}*, three inputs *{start, stop, Dt}*, and two outputs *{null, timeout}* (*null* means no output is produced). A timer is *active* if it is running; otherwise it is *inactive*. *Dt* is an external event used to represent a time interval of *Dt* time units, during it no input (*start* or *stop*) occurs. When the timer is *set* (the input event *start*), it enters the state *active* and waits an interval *Dt* for an input event *start* or *stop*. If any of these two events does not occur in the interval *Dt*, the timer *expires* and produces the output *timeout*. At any time, a timer can be switched off by the input event *stop*. As consequence, the timer goes to the state *inactive*.

The counter's FSM has $N+1$ states representing the values $0 \leq i \leq N$ of the counter; N is the greatest integer the counter is compared to. Initially, the counter takes the value 0. Then, it is either incremented by 1 ($C := C + 1$) or reset to zero ($C := 0$). Thus, the set of inputs consists of two input events $C := C + 1$ and $C := 0$. However, the set of outputs consists of two output events $C < N$ and $C \geq N$ to indicate whether or not the value of the counter is less than N. From a state i and on input $C := C + 1$ (respectively $C := 0$), the counter goes to the state $(i + 1)$ and produces either the output $C < N$, or

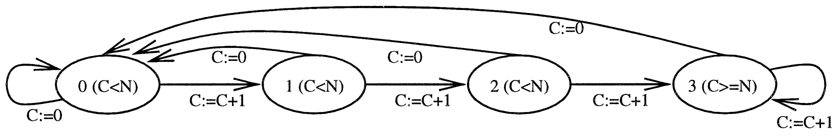


Fig. 23. The counter FSM.

$C \geq N$ (respectively goes to the state 0 and produces the output $C < N$).

8.3.2. Test cases generation for TIOA

As we early mentioned, the TIOA is the most suitable model to describe real-time systems. It represents an extension of the w-automata with clock variables and time constraints to control the execution of the transitions. From a testing viewpoint, the TIOA can be seen as an Extended Finite State Machine (EFSM for short) model in which the variables are clocks. In the EFSM model, the variables are manipulated by the user. Howeever, in the TIOA model, the only operation that a user can make on clocks is the reset to zero; otherwise, the clocks change continuously when time progresses. Moreover, since the TIOA remains an abstract model, testing such model assumes that the well known path executability problem of the EFSM model is solved. For this reason, we must consider the relation between clocks by using the region graph as a semantic model of TIOA.

Based on the region graph, many systematic testing methods are developed [36,88,37,38]. They are the adapted versions of the FSM's based techniques [88]. constitutes the first theoretical framework that has been published in this domain. In fact, the authors use the W-method (see Section 6.2.4) and the bisimulation relation [71,72] to derive timed test cases with a complete test coverage. However, the model used, as basis for this work, is not general. Indeed, in order to control the responses of the IUT, the outputs are assumed to occur only on the integer values of clocks. Moreover, the number of timed test cases generated by this approach is very large even for a simple example of specification. All subsequent work has to derive practical testers or test cases with less fault coverage guarantee but acceptable fault coverage.

In Ref. [36], we applied the TT-method on the entire region graph. We identified a state only by one empirical value for each clock and we have shown how to execute the obtained timed test cases by using a specific test architecture. In Ref. [38], we re-used the test architecture of [36] and we used the LTS technique to generate timed test cases based on the conformance relation *ioconf* [93]. We first minimize the region graph using the algorithm presented in Ref. [3]. Then, we translate the resulting minimal region graph into an LTS in which the label of each transition consists of a couple (l,z), where l is an action of the initial TIOA and z is a clock zone. Finally, we applied Tretmans' method [93] to generate test cases from the resulting LTS.

In Ref. [37], we presented a method to generate timed test cases from a TIOA with a good fault coverage. This method is based on the state characterization technique and constitutes an extension of the Wp-Method to the timed setting. Thus, many transformations are required. First, the region graph is sampled with a particular granularity to cover all clock regions of the TIOA. For a n-clock TIOA, we use a granularity of $1/(n+2)$ if the number of clocks is greater or equal to 2; otherwise, we use a granularity of $1/2$ (see [60] for more details on sampling). The sampling operation leads to an explicit extension of the TIOA's alphabet (actions) with the granularity delay action; the resulting automata is called a *Grid Automata*. This grid automata is then transformed into a Nondeterministic Timed Finite State Machine (NTFSM). Finally, an adapted version of the Generalized Wp-Method [65] is applied to the resulting NTFSM.

As an example, let us consider again the TIOA of Fig. 20. The corresponding Grid automata is shown in Fig. 24. Since the TIOA of Fig. 20 has one clock, we use a granularity of $1/2$ to sample the region graph. So, each state (l_i,v), in the grid automata, has an outgoing delay transition labelled with $1/2$ and whose target state is $(l_i,v+1/2)$. For example, the set of states reachable from the initial state $(l_0,0)$ by consecutive $1/2$-delay actions is $\{(l_0,1/2),(l_0,1),(l_0,\infty)\}$. Notice that the time constraint of the transition $l_0 \xrightarrow{?On,\lambda} l_1$ is empty. Therefore, from any of the states $\{(l_0,0),(l_0,1/2),(l_0,1),(l_0,\infty)\}$, there is an outgoing transition labelled with the input action ?*On* and whose target state is $(l_1,0)$. For a complete description of the sampling algorithm, we refer the reader to [37].

The NTFSM resulting from the transformation of the Grid automata of Fig. 24 is shown in Fig. 26. In this Figure, the reset to zero of clock x is represented explicitly by a signal *Resetx* following the timed test model of [36,38]. Furthermore, an output (possibly *Null*) is associated to each input to indicate what will be the response of the IUT on this input.

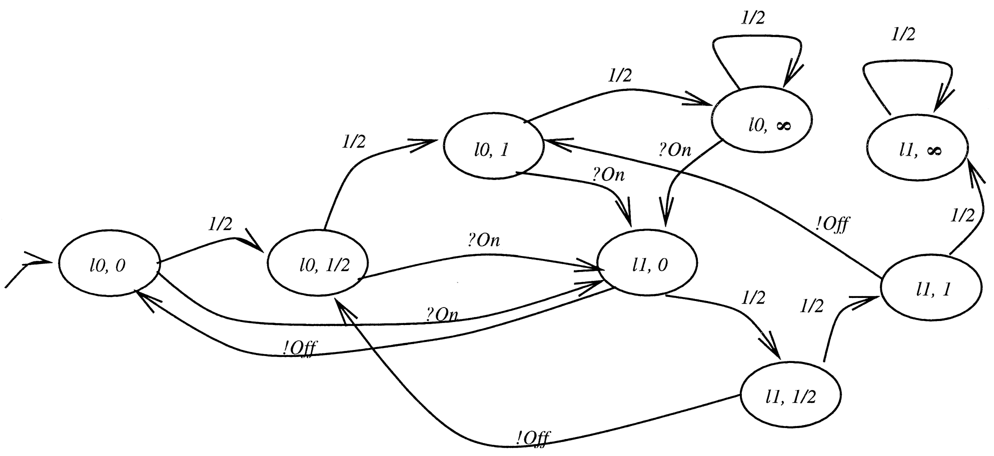


Fig. 24. A grid automata of the TIOA of Fig. 20.

The translation of the grid automata into the NTFSM is made according to the two basic schemes of Fig. 25.

8.3.3. *Other timed test cases generation methods*

In addition to the timed test cases generation methods presented above, there exists other approaches [26,69] that are based on other formal models. In Ref. [26], the authors present an approach to generate timed test cases from a *Constraint Graph (CG)*. A CG is a directed acyclic graph $G(V,E)$, where V is the set of nodes representing input and output actions, and E is the set of edges modelling in some sense the transitions of the system. Each edge in the graph is labelled with a time interval that represents the time constraint between the occurrence of the edge's source action and that of the edge's destination action. The test cases are generated based on the satisfaction of some criteria on the set of actions and time constraints. In Ref. [69], an extension to the classical temporal logic is presented to deal with timing aspects, and timed test cases are generated from formulas written in that logic. In this study, the time domain is discretized into integer values and timed test cases are generated on the basis of what is called *Histories*.

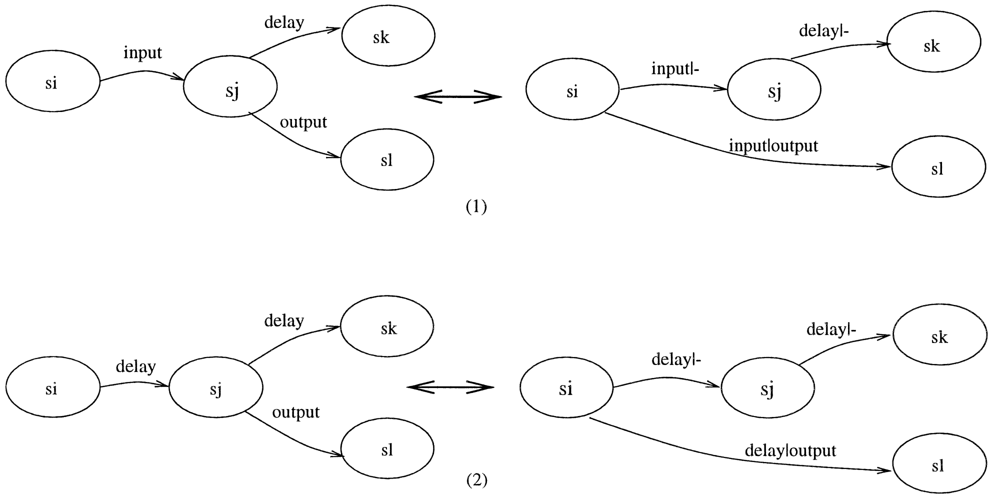


Fig. 25. The basic transformation schemes.

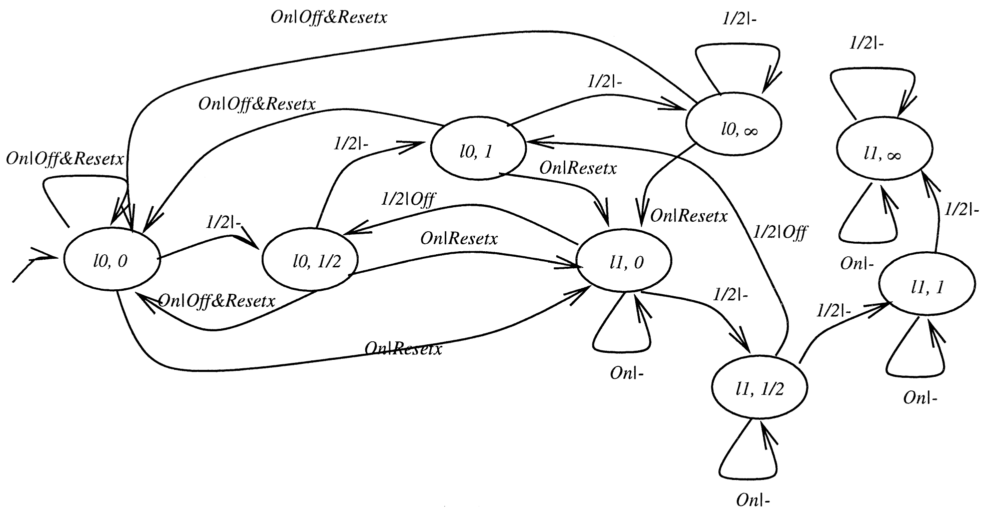


Fig. 26. The NTFSM of the grid automata of Fig. 24.

However, the formal models used as a basis to these results have limitations with respect to their description of real-time systems. In this sense, the CG model is restricted to only describing minimum and maximum allowable delays between input/output events in the execution of a system following Taylor's and Dassarathy's [89,29] classification of timing requirements. The extended temporal logic of [69] allows only simple formulas with one clock.

9. Testing complex systems

9.1. Introduction

So far, we have only surveyed and assessed testing methods for sequential programs, typically assuming that a single program module would be tested in isolation. Moreover, testing methods based on finite state models, which are particularly suitable for reactive systems for which inputs and outputs, present themselves as sequences of interlaced interactions with the environment. In the following, we will address issues related to testing complex systems, usually having composite structures. The following are the main issues in testing such systems:

- Composite structure of the system under test.
- Modeling techniques combining different behavior aspects and related testing methods, for example so-called extended FSM models combining FSM models and data (in terms of state variables, interaction parameters, data types, etc., usually modeled using high-level programming languages), or real-time properties.
- Methodological issues related to the assumptions about the structure of implementations and associated fault models, and about coverage criteria and the cost of testing.

9.2. Problems of distributed testing

Several software programs calling one another (e.g. several related objects, or frameworks), several communicating FSM or EFSM modules. The following is a list of issues related to distributed testing of communications software:

- Composition issue.
- Testability in terms of observability and controllability.
- Synchronization of different testers.

The testing approaches can be classified according to whether they assume that the possible faults in one given module and their detection is independent of the possible faults in the other modules (independence assumption). With the independency assumption, coverage criteria for each module may be simply combined. Here are examples for the case of FSM testing: If the transition tour coverage is used for each given module, then the desired global test suite should simply cover all transitions of all modules. If the ‘‘complete fault coverage for output and transfer faults'' (see Section 5) is used for each module, then the desired global test suite should satisfy the corresponding sufficient conditions for each given module, assuming that the other modules have no faults. (Note: In practice, these conditions are not so easy to satisfy in general, because certain wrong outputs of an internal module may not be directly visible by the testing environment.) Without this assumption, one may consider the reachability graph of the system of composed modules (i.e. consider all possible interactions among the modules for all possible input sequences) and apply some test derivation method based on this graph. However, this graph represents a kind of product of the individual modules and is thus much more complex that the ''sum'' of the modules, accordingly, the derived tests would be very complex in general.

9.3. Testing module with multiple interfaces

A general distributed test architecture where the IUT (implementation under test) contains several distributed ports has been studied for testing distributed systems. It is based on the Open Distributing Processing (ODP) Basic Reference Model (BRM), see Fig. 27. In this architecture, the IUT contains several distributed interfaces, called ports or PCOs (i.e., Points of Control and Observation). Also, the testers cannot communicate and synchronize with one another unless they communicate through IUT, and no global clock is available in the system. This could be a test architecture of a communication network with n accessing nodes, where the testers reside in these nodes. When $n = 2$, this general distributed test architecture is reduced to the ISO distributed test architecture [55] for communication protocol testing.

Usually, in the so-called local test architecture developed by ISO, the specifications of communication protocols are first abstracted into state machines [62], then test cases are generated from the resulting machines. A number of methods have been developed to generate test cases for finite state machines (FSMs) [40,76,24,83,75] or for nondeterministic FSMs [67], however, they are not directly applicable to the distributed test architecture, because of the synchronization problem between distributed testers. In distributed test architectures, testing is relatively difficult because certain problems of synchronization between the testers may arise during the application of test sequences. To solve this problem, with respect to the ISO distributed test architecture where there are only two ports, an approach for test generation has been developed in Ref. [84] by modifying the existing test generation methods for FSMs such as the transition tour [75], the DS-method [59], and

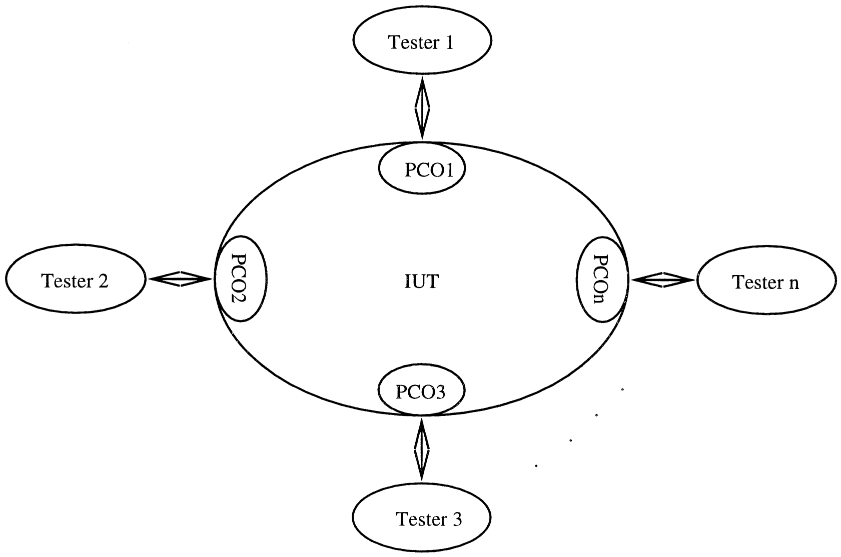


Fig. 27. Distributed test architecture.

the W-method [24] such that the resulting sequences are so-called synchronizable test sequences.

9.4. Testing embedded components

Testing an isolated component is usually less complex than testing the same component in its environment. The relationship of the component to be tested with others components in the environment should be taken into account for test cases generation. For this purpose, the structure of the environment has to be known. This type of testing is known as embedded testing or testing in context [56,77, 63,13]. An example of the application of such type of testing is the testing of telecommunication services in the switching systems. Let us consider a general case of a system with two components ''IUT'' and ''Test Context''. The ''IUT'' and ''Test Context'' communicate with each other through hidden interfaces, and the ''Test Context'' component communicates with the environment using PCOs in an observable way. Testing embedded ''IUT'' component raises the following problems, – partial controllability and observability of the component under test. The solution should foresee in order to determine the partial product of system components that provide the maximum controllability and observability of the component under test. In most cases and depending on the behavior of the environment, the partial product will offer less fault detection possibility (less control and observability) compared to the testing of the component in isolation (see Fig. 28).

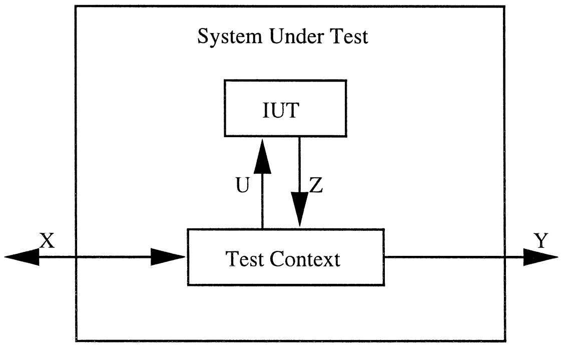


Fig. 28. Embedded system.

10. A chain of tools for test development from formal specification

10.1. Tools for test suite development

We mention specifically a tool for derivation of tests from deterministic, partially defined FSM models which was developed at the University of Montreal [12]. Testgen which was developed at INT Evry Tools for test suite development from SDL or Estelle specifications. Since SDL specifications have an underlying FSM model (which is a simplification of the SDL specification) [16,18], one can apply the FSM test derivation methods to this simplified model. The tests derived by such an approach could check for output and transfer faults. The obtained FSM test cases must be augmented manually to include the necessary processing of interaction parameters. (Note: The outputs could be checked not only for the correct output primitive, but also for the correct output parameter values.)

Different approaches to the automation of test development from EFSM models have been pursued: (i) combination of FSM testing methods and data flow criteria (e.g. Sarikaya, Tripathy at Concordia Univ., Kim at UBC), (ii) automated test case generation for user-guided test selection based on test purposes given in terms of interaction scenarios (Hogrefe's group at Bern), (iii) partial unfolding approach (the FEX tool developed at the University of Montreal), (iv) a pragmatic approach developed at CNET, France: The tool TVEDA, finally, (v) commercial test tool TestMaster (Paradyne Inc.) which uses an ad-hoc EFSM model and automatic exhaustive test development which can be constrained by the user. Various tools exist to support the Tree and Tabular Combined Notation (TTCN) test definition language, such as table-oriented editors, compilers, and partial translators from SDL to TTCN. (Note: many of the test generation tools mentioned above support TTCN as a possible output language.)

10.2. An example of a chain of tools

Fig. 29 shows a set of automated tools that has been developed at the University of Montreal. A complete description of this chain tool is given in Ref. [12]. The methodology followed in this work is based on partial unfolding of SDl specification of a given system to be tested. The test derivation relies on FSM-based techniques. To derive test suite from SDL specification, we have to extract automatically the FSM form specification. The FEX tool (FSM Extraction)functions are:

- Permit to obtains FSM model from a given SDL specification.
- It uses partial unfolding branch coverage of SDL specification (similar to normal form transitions for Estelle).
- Preserves constraints on the values of input parameters.
- Creates separate files to be integrated in the test cases generated by the TAG tool.
- The test cases generated by TAG must be completed by hand provide values for output parameters: check input signal parameters in some situations, add iteration on some behavior to make test case executable.

A TAG tool for test suite derivation from partial FSM Specifications has been developed by Tan (PhD student, UdeM). it implements transition identification approach, that is similar to the approach used in TVEDA. TAG provides the following functions:

- Compiles an FSM specification and provides related information: nondeterminism, state distinguishability, etc;
- Displays the FSM state table and intermediate results: preambles, state identifiers and postambles;
- Generates a test suite with complete fault coverage;
- Derives test case for a given test purpose;
- A simple and flexible input format facilitates the definition of states, input/output events and transitions;
- The generated tests are presented in the form of readable I/O sequences, or as SDL or TTCN skeletons.

A branch of commercial tools can be used for validation and translation to implementation lan-

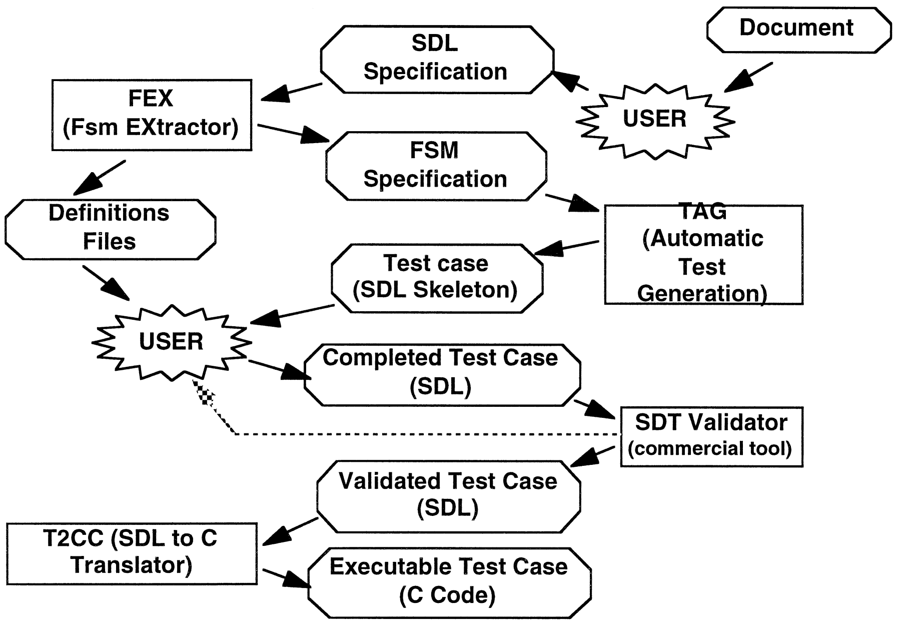

SDL Specification
Document
FEX (Fsm EXtractor)
USER
FSM Specification
Definitions Files
TAG (Automatic Test Generation)
Test case (SDL Skeleton)
USER
Completed Test Case (SDL)
SDT Validator (commercial tool)
Validated Test Case (SDL)
T2CC (SDL to C Translator)
Executable Test Case (C Code)

Fig. 29. An example of test automation.

guages. This cahin of tools has been used for the development of test suite for the ATM PNNI signalling protocol.

11. Some test generation tools from SDL specifications

Over the past ten years, tools have become available that seek to automate the software testing process. These tools can help to improve the efficiency of the test execution process by replacing personnel with test scripts that playback application behavior. However, it is the up-front process of deciding what to test and how to test it that has the dominant impact on product quality. Likewise, the cost and time to develop tests is an order of magnitude greater than that required for test execution. Today, manual methods are still the primary tools for this critical stage, however, tools exist which automate some parts of the testing process. In the following, some existing tools for test case generation or tools that help the test suite developer in the test generation process are presented.

TESDL [15] is a prototype tool for the automatic generation of test cases from SDL specifications in the context of the OSI Conformance Testing Methodology and Framework. TESDL implements a heuristic algorithm to derive the global behavior of a protocol as a tree, called Asynchronous Communication Tree (ACT), which is based on a restricted set of SDL diagrams (one process per block, no two processes are able to receive the same kind of signal, etc.). The ACT is the global system description as obtained by reachability analysis by perturbation. In the ACT, the nodes represent global states. A global state contains information about the states of all processes in the specification. Tests are derived from the ACT of a specification by a software tool, called TESDL. Input for the tool is a SDL specification (SDL-PR), the output are the test cases in TTCN-Notation.

TTCN Link (LINK for short) [90] is an environment for efficient development of TTCN test suites based on SDL specifications in SDT3.0 (SDL Description Tool) [91]. LINK assures consistency between the SDL specification and the TTCN test suite. It increases the productivity in the development by automatic generation of the static parts of the test suite and specification-based support for the test case design. The intended user of the LINK tool is a TTCN test suite developer. His inputs are an SDL specification and a test suite structure with test purposes and his task is to develop an abstract TTCN test suite, based on this input. This tool is semi-automatic.

SAMSTAG [46] is developed within the research and development project ‘‘Conformance Testing a Tool for the Generation of Test Cases’’ which is funded by the Swiss PTT. The allowed behavior of the protocol which should be tested is defined by an SDL specification and the purpose of a test case is given by an MSC which is a widespread mean for the graphical visualization of selected system runs of communication systems. The SaMsTaG method formalizes test purposes and defines the relation between test purposes, protocol specifications and test cases. Furthermore, it includes the algorithms for the test case generation.

TOPIC V2 [2] (prototype of TTC GEN) works by co-simulating the SDL specification and an observer representing the test purpose. This co-simulation enables to explore a constrained graph, i.e., a part of the reachability graph of the specification, which enables to use this method for infinite graphs. The observer is described in a language called GOAL (Geode Observation Automata Language). In order to facilitate the use of TOPIC, it is possible to generate the observers from MSC’s. From the constrained graph, some procedures are executed in order to generate the TTCN test.

Tveda V3 [28] is a tool for automatic test case generation which incorporates several features: A modular architecture, that makes it possible to choose between the specification languages (Estelle or SDL), test description languages (TTCN or Menuet) and test selection strategy (single transition, extended transition tour, etc.) A semantic module, which can be called from the strategy modules to compute feasible paths. Functional extensions, such as hypertext links between tests and specification, test coverage analysis, etc. To compute execution paths, two techniques can be used, symbolic computation techniques or reachability analysis. Symbolic computation techniques put strong restrictions on which constructs are accepted and the path computation requires an exponential computation with respect to the length of the path to be computed. On the contrary, reachability analysis puts almost no restriction on the Estelle or SDL constructs which are accepted, and it is implemented by interfacing Tveda with a powerful commercial tool for reachability analysis, Véda.

12. Conclusion

From all this, we draw the following remarks:

FSM based testing methods are well developed for deterministic and completely defined specifications.

A huge amount of work has been completed on LTS [14] (testing based on LTS is not given in this paper).

All this work constitutes for industry a little step towards a complete test automation. Realistic implementation such as Switches, multimedia applications and emerging protocols are of composed type systems with multiple test interfaces and m to n communication model. Further work on test suite development is required for

non-deterministic specifications,

embedded testing,

testing based on extended FSM models, such as SDL,

real-time testing.

Furthermore, practical testing tools require simple interfaces with the IUT and flexible test architectures, easily usable test script language (SDL, TTCN, C), and better integration with specification and implementation languages.

Acknowledgements

This paper has been partially based on a tutorial which was prepared in collaboration with G. v. Bochmann for the SDL Forum 1997. The authors would like to acknowledge the contributions from other members of the research group at the University of Montreal, in particular Gregor v. Bochmann, Omar Bellal, Daniel Ouimet, Alexandre Petrenko and Mingyu Yao, Gang Luo, Aziz Saleh. Many examples and ideas in this tutorial have been taken from their work. This work was partly supported by the Hewlett-Packard - NSERC - CITI Industrial Research Chair on Communication protocols, NSERC Individual Grant and an NSERC Strategic Grant.

References

[1] A. Aho et al., An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours, in: S. Aggarwal, K.K. Sabnani (Eds.), Protocol Specification, Testing and Verification VIII, North-Holland, Amsterdam, 1988.

[2] B. Algayres, Y. Lejeune, F. Hugonnet, GOAL: observing SDL behaviors with object code, in: Proc. 7th SDL Forum, Oslo, Norway, September 1995.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, H. Wong-Toi, Minimization of timed transition systems, in: Third International Conference on Concurrency Theory, 1992, pp. 340–354.

[4] R. Alur, D. Dill, Automata for modeling real-time systems, in: 17th ICALP (International Colloquium on Automata, Languages, and Programming), 1990, pp. 322–335.

[5] R. Alur, D. Dill, A theory of timed automata, Theoretical Computer Science 126 (1994) 183–235.

[6] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold Electrical/Computer Science and Engineering Series, 1983.

[7] G. Bernot, M.C. Gaudel, B. Marre, Software testing based on formal specifications: a theory and a tool, Software Engineering Journal (1991) 387–405.

[8] G. v. Bochmann, Protocol specification for OSL, Computer Networks and ISDN Systems 18 (1989) 167–184.

[9] G. v. Bochmann et al., Fault model in testing, in: R.J. Heijink, J. Kroon, E. Brinksma (Eds.), IFIP Transactions, Protocol Test Systems, IV (Proc. IFIP TC6 4th International Workshop on Protocol Test Systems, 1991) North-Holland, Amsterdam, 1992, pp. 17–30.

[10] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, G. Luo, Fault model in testing, in: Proc. International Workshop on Protocol Test System (IWPTS'91), Leidschendam, the Netherlands, 1991.

[11] G. v. Bochmann, R. Dssouli, J.R. Zhao, Trace analysis for conformance and arbitration testing, IEEE Transactions on Software Engineering SE-15 (1989) 1347–1356.

[12] G. v. Bochmann, A. Petrenko, O. Bellal, S. Maguiraga, Automating the process of test derivation from SDL specifications, in: Proc. Eighth SDL Forum, Evry, France, September 1997.

[13] C. Bourhfir, R. Dssouli, El M. Aboulhamid, N. Rico, A guided incremental test case generation procedure for conformance testing CEFSM specified protocols, in: A. Petrenko, N. Yevtushenko (Eds.), IFIP International Workshop on Testing Communicating Systems 98, Chapman and Hall, London, 1998.

[14] E. Brinksma, A theory of the derivation of tests, in: S. Aggarwal, K.K. Sabnani (Eds.), Protocol Specification,

Testing and Verification VIII, North-Holland, Amsterdam, 1988, pp. 63–74.

[15] L. Bromstrup, D. Hogrefe, TESDL: Experience with generating test cases from SDL specifications, in: Proc. 4th SDL Forum, 1989.

[16] A. Cavalli, B.M. Chin, K. Chon, Testing methods for SDL systems, Computer Networks and ISDN Systems 28 (1996) 1669–1683.

[17] A. Cavalli, J-P. Favreau, M. Falippou, Standardization of formal methods in conformance testing of communication protocols, Computer Networks and ISDN Systems 29 (1996) 3–14.

[18] A. Cavalli, B. Lee, T. Macavei, Test generation for the SSCOP-ATM network protocol, in: Proc. SDL FORUM'97, Evry, France, Elsevier, 1997.

[19] A.R. Cavalli, R. Anido, Verification and testing techniques based on finite state machine model, Research Report 97-09-02 INT, France, 1997.

[20] CCITT, Specification and Description Language (SDL), Recommendation Z.100, International Standard Z.100, CCITT, Genève, 1993.

[21] S.T. Chanson, J. Zhu, A unified approach to protocol test sequence generation, in: IEEE INFOCOM, San Francisco, CA, 1993.

[22] S.T. Chanson, J. Zhu, Automatic protocol test suite derivation, in: IEEE, 1994.

[23] O. Charles, Application des hypothès de test à une définition de la couverture, PhD thesis, Université Henri Poincaré-Nancy 1, 1997.

[24] T.S. Chow, Testing software design modeled by finite-state machines, IEEE Transactions on Software Engineering SE-4 (1978) 178–187.

[25] W. Chun, P.D. Amer, Test case generation for protocols specified in ESTELLE, in: FORTE'90, Madrid, Spain, 1990.

[26] D. Clarke, I. Lee, Automatic generation of tests for timing constraints from requirements, in: Proc. Third International Workshop on Object-Oriented Real-Time Dependable Systems, Newport Beach, CA, February 1997.

[27] L.A. Clarke, D.J. Richardson, Applications of symbolic evaluation, Journal of Systems and Software 5 (1985) 15–35.

[28] M. Clatin, R. Groz, M. Phalippou, R. Thummel, Two approaches linking a test generation tool with verification techniques, in: Proc. International Workshop on Protocol Test System IFIP, North-Holland, Amsterdam, September 1995.

[29] B. Dasarathy, Timing constraints of real-time systems: constructs for expressing them, IEEE Transactions on Software Engineering 11 (1985) 80–86.

[30] J. De Kleer, B.C. Williams, Diagnosing multiple faults, Artificial Intelligence 32 (1984) 97–130.

[31] R. Dechter, J. Pearl, Tree clustering for constraint networks, Artificial Intelligence 38 (1989) 353–366.

[32] R.A. DeMillo, J. Offutt, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering SE-17 (1991).

[33] D. Dill, Timing assumptions and verification of finite-state concurrent systems, in: 1st CAV (Conference on Computer-Aided Verification) 1989, pp. 197–212.

[34] R. Dssouli, Etude des méthodes de test pour les implantations de protocoles de communication basées sur les spécifications formelles, PhD thesis, Université de Montréal, 1986.

[35] R. Dssouli, Les facteurs influancant l'observation de fautes dans les logiciels de communication. in: O. Rafiq (Ed.), Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'91 (Pau, France), Paris, September 1991.

[36] A. En-Nouaary, R. Dssouli, A. Elqortobi, Génération de tests temporisés, in: Proc. 6th Colloque Francophone de l'Ingénierie des Protocoles, 1997.

[37] A. En-Nouaary, R. Dssouli, F. Khendek, A. Elqortobi, Timed test cases generation based on state characterisation technique, in: 19th IEEE Real-Time Systems Symposium (RTSS'98), Madrid, Spain, December 1998.

[38] A. En-Nouaary, H. Fouchal, A. Elqortobi, R. Dssouli, E. Petitjean, Timed testing using clock zone vertices, Technical Report, 1998.

[39] R.E. Fikes, A system for solving problems stated as procedures, Artificial Intelligence 1 (1970) 27–120.

[40] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, Test selection based on finite state models, IEEE Transactions on Communications 17 (1991) 591–603.

[41] M.C. Gaudel, Testing can be formal, too, TAPSOFT, 1995.

[42] M.C. Gaudel, Test selection based on ADT specifications, in: Proc. 5th International Workshop on Protocol Test Systems (IWPTS'92), Montreal, Canada, 1998.

[43] A. Ghedamsi, G. v. Bochmann, R. Dssouli, Diagnosing distributed systems modeled by CFSMs, Journal Réseaux et Informatique Répartie, France, 1993.

[44] A. Gill, Introduction to the Theory of Finite State Machines, McGraw-Hill, New York, 1962.

[45] G. Gonenc, A method for the design of fault detection experiments, IEEE Transactions on Computers C-19 (1970) 551–558.

[46] J. Grabowski, D. Hogrefe, R. Nahm, A method for the generation of test cases based on SDL and MSC, Technical Report Institut für Informatik, Universität Bern, April 1993.

[47] R. Groz, M. Phalippou, La génération automatique de tests est-elle possible, in: C. Jard, P. Rolin (Eds.), Colloque Francophone sur l'Ingenierie des Protocols CFIP'95, 1995.

[48] T. Higashino, G. v. Bochmann, X. Li, K. Yasumoto, K. Taniguchi, Test system for a restricted class of LOTOS expressions with data parameters, in: Proc. Fifth IFIP International Workshop On Protocol Test Systems (IWPTS'92), North-Holland, Amsterdam, 1992, pp. 205–216.

[49] D. Hogrefe, MUTEST: OSI Formal Specification Case Study: The INRES Protocol and Service, Internal Report, 1992.

[50] W.E. Howden, Reliability of the path analysis testing strategy, IEEE Transactions on Software Engineering SE-2 (3) (1976).

[51] W.E. Howden, An evaluation of the effectiveness of symbolic testing, Software Practice and Experience 8 (1978) 381–397.

[52] E. Htite, R. Dssouli, G. v. Bochmann, Sélection des tests à

partir de spécifications orientées objets, in: Proc. Third Maghrebian Conf. on Software Eng. and Art. Intelligence, Rabat, Maroc, April 1994.
[53] S. Htite, R. Dssouli, A. Ghedamsi, Diagnostique automatic avec l'outil MFDT, in: Proc. 6th bi-Annual Colloque Francophone de l'ingénierie des Protocoles, Lièges, Belgique, 1997.
[54] C.M. Huang, Y.C. Lin, M.Y. Jang, Executable data flow and control flow protocol test sequence generation for EFSM-specified protocol, in: International Workshop on Protocol Test Systems (IWPTS), Evry, France, 1995.
[55] ISO, Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, ISO/TC97/SC21/N DIS8807, 1987.
[56] ISO9646, International Standard 9646, International Organization for Standardization — Information Technology — Open Systems Interconnection, Genève 1991.
[57] J. Jaffar, J.L. Lassez, Constraint logic programming system, in: Proc. 14th ACM POPL Conference, Munich, Germany, 1987.
[58] J.C. King, T.J. Watson, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976).
[59] Z. Kohavi, Switching and Finite Automata Theory, McGraw-Hill Computer Science Series, New York, 1978.
[60] K.G. Larsen, W. Yi, Time abstracted bisimulation: implicit specifications and decidability, in: Proc. Mathematical Foundations of Programming Semantics (MFPS 9), Lecture Notes in Computer Science, vol. 802, Springer, Berlin, 1993.
[61] G. Leduc, On the role of implementation relations in the design of distributed system, PhD thesis, Publications de la Faculté des Sciences Appliquées de l'Université de Liège, vol. 130, Liège 1991.
[62] D.Y. Lee, J.Y. Lee, A well-defined ESTELLE specification for automatic test generation, IEEE Transactions on Communications 40 (1991) 526–542.
[63] L.P. Lima, A. Cavalli, A pragmatic approach to generating test sequences for embedded systems, in: Proc. IWTCS'97, Cheju Islands, Korea, 1997.
[64] F. Liu, Test generation based on an FSM model with timers and counters, Master thesis, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1993.
[65] G. Luo, G. v. Bochmann, A. Petrenko, Test selection based on communicating nondeterministic finite state machines using a generalised WP-method, IEEE Transactions on Software Engineering SE-20 (2) (1994).
[66] G. Luo, R. Dssouli, G. v. Bochmann, P. Vankataram, A. Ghedamsi, Test generation with respect to distributed interfaces, Computer Standards and Interfaces 16 (1994) 119–132.
[67] G. Luo, A. Petrenko, G. v. Bochmann, Selecting test sequences for communicating partially-specified nondeterministic finite state machines, Technical report TR-864 IRO, Université de Montréal, 1993.
[68] A.K. Mackworth, Consistency of network relations, Artificial Intelligence 8 (1977) 99–118.
[69] D. Mandrioli, S. Morasca, A. Morzenti, Generating test cases for real-time systems from logic specifications, ACM Transactions on Computer Systems 13 (1995) 365–398.
[70] R.E. Miller, S. Paul, Generating conformance test sequences for combined control and data flow of communication protocols, in: Proc. Protocol Specification, Testing, and Verification (PSTV'92), Florida, USA, 1992.
[71] R. Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92, Springer, Berlin, 1980.
[72] R. Milner, Communication and Concurrency, Prentice-Hall International, Englewood Cliffs, NJ, 1989.
[73] U. Montanari, Networks of constraints: fundamental properties and applications to picture processing, Information Science 7 (1974) 95–132.
[74] L.J. Morell, A theory of fault-based testing, IEEE Transactions on Software Engineering SE-16 (8) (1990).
[75] S. Naito, M. Tsunoyama, Fault detection for sequential machines by transition tours, in: Proc. Fault Tolerant Computer Systems, 1981, pp. 238–243.
[76] A. Petrenko, Checking experiments with protocol machines, in: J. Kroon, R. Heijink, E. Brinksma (Eds.), IFIP International Workshop on Protocol Test Systems IV, North-Holland, Amsterdam, 1991.
[77] A. Petrenko, N. Yevtushenko, G. v. Bochmann, R. Dssouli, Testing in context: A framework and test derivation, Special Issue on Protocol Engineering of Computer Communication, 1997.
[78] M. Phalippou, Relation d'implantation et hypothèses de test sur des automates à entrées et sorties, PhD thesis, Université de Bordeaux I, 1994.
[79] D. Pountain, Constraint logic programming, Byte Magazine, February 1995.
[80] T. Ramalingom, A. Das, K. Thulasiraman, A unified test case generation method for the EFSM model using context independent unique sequences, in: Proc. International Workshop on Protocol Test System (IWPTS'95), Evry, France, 1995.
[81] C.V. Ramamoorthy, S.B.F. Ho, W.T. Chan, On the automated generation of program test data, IEEE Transactions on Software Engineering SE-2 (4) (1976).
[82] D. Rayner, OSI conformance testing, Computer Networks and ISDN Systems 14 (1987) 79–98.
[83] K. Sabnani, A.T. Dahbura, A protocol testing procedure, Computer Networks and ISDN Systems 15 (1988) 285–297.
[84] B. Sarikaya, G. v. Bochmann, Synchronization and specification issues in protocol testing, IEEE Transaction on Communications 32 (1984) 389–395.
[85] B. Sarikaya, G. v. Bochmann, E. Cerny, A test methodology for protocol testing, IEEE Transactions on Software Engineering SE-13 (1987) 518–531.
[86] Y.N. Shen et al., Protocol conformance testing using multiple UIO sequences, in: Protocol Specification, Testing and Verification IX, Twente, Netherlands, 1989.
[87] D.P. Sidhu, T.K. Leung, Formal methods for protocol testing: a detailed study, IEEE Transactions on Software Engineering SE-15 (4) (1989).
[88] J. Springintveld, F. Vaandrager, P.R. d'Argenio, Testing timed automata, invited talk at TAPSOFT'97, 1997, http://www.cs.kun.nl/fvaan/publications.html

[89] B. Taylor, Introducing real-time constraints into requirements and high level design of operating systems, in: Proc. 1980 Nat. Telecommunications Conference, Houston, TX, 1980, vol. 1, pp. 18.5.1–18.5.5.
[90] Telelogic, ITEX User Manual, 1995.
[91] Telelogic, SDT User Manual, 1995.
[92] J. Tretmans, A formal approach to conformance testing, in: Proc. 6th International Workshop on Protocol Test Systems (IWPTS'93), Pau, France, 1993.
[93] J. Tretmans, Testing labelled transition systems with inputs and outputs, Technical report 95-26, University of Twente, August 1995.
[94] H. Ural, B. Yang, A test sequence selection method for protocol testing, IEEE Transactions on Communication 39 (4) (1991).
[95] S.T. Vuong, W.W.L. Chan, M.R. Ito, The UIOV method for protocol test sequence generation, in: Proc. 2nd International Workshop on Protocol Test Systems, Berlin, Germany, 1989.
[96] E.J. Weyuker, S. Rapps, Selecting software test data using data flow information, IEEE Transactions on Software Engineering SE-11 (4) (1985).
[97] J.P. Wu, S.T. Chanson, Test sequence derivation based on external behavior expression, in: Proc. 2nd International Workshop on Protocol Test Systems, 1989.
[98] B. Yang, H. Ural, Protocol conformance test generation ising multiple UIO sequences with overlapping, Computer Communication Review 4 (1997) 118–125.
[99] M. Yao, On the development of conformance test suites in view of their fault coverage, PhD thesis, Université de Montréal, Département IRO, 1995.
[100] L.D. Zhang et al., A further optimization technique for conformance testing based on multiple UIO sequences, in: G. v. Bochmann, R. Dssouli, A. Das (Eds.), Protocol Test Systems V (Montreal 1992), North-Holland, Amsterdam, 1993.

Rachida Dssouli is professor in the Département d'Informatique et de Recherche Opérationnelle (DIRO), Université de Montréal. She received the Doctorat d'université degree in computer science from the Université Paul Sabatier of Toulouse, France, in 1981, and the Ph.D. degree in computer science in 1987, from the University of Montréal, Canada. She has been a professor at the Université Mohamed 1er, Oujda, Morroco, from 1981 to 1989, and assistant professor at the Université de Sherbrooke, from 1989 to 1991. She spent a sabbatical year at NORTEL (1995–1996), Ile des Soeurs. Her research area is in Communication protocol engineering, Requirements engineering and Multimedia applications. Ongoing projects include incremental specification and analysis of reactive systems based on scenario language, multimedia applications, conformance testing, design for testability, conformance testing based on FSM/EFSM and Timed automata. She served very often as a member of committee program of many workshops and conferences (IWPTS, IWTCS, FORTE, CFIP, MMNS, MMM, FIW, NOTERE). She organized or coorganized several international workshops and conferences (IWPT'93, CFIP'93, FORTE'95, CFIP'96, NOTERE'98, 9th SDL Forum).

Kassem Saleh obtained his B.Sc., M.Sc. and Ph.D. from the University of Ottawa in Canada in 1985, 1986 and 1991, respectively. He worked for Bell Canada from 1985 to 1991 and at Concordia University for one year before joining Kuwait University in 1992. He is currently an Associate Professor in the Department of Electrical and Computer Engineering, College of Engineering and Petroleum at Kuwait University. Dr. Saleh was placed in 8th position among the top scholars in the Field of Systems and Software Engineering in an annual assessment published by the Journal of Systems and Software in October 1997 and October 1998. He was awarded the IBM telecommunications Software Scholarship in 1988, the George Franklin Prize for the best student paper in 1990 from the Canadian Interest Group on Open Systems (CIGOS), the distinguished young researcher prize in 1994 and the distinguished teacher prize in 1996 both from the College of Engineering and Petroleum at Kuwait University. His current research and teaching activities include software engineering, communications software, distributed systems and internet programming. Dr. Saleh has presented many tutorials at international conferences and universities worldwide.

El Mostpha Aboulhamid got his Engineering degree in Computer Science and Mathematics form ENSIMAG (Institut Polytechnique de Grenoble) in 1974. He obtained his Ph.D. and M.Sc. degrees from Université de Montréal in 1985 and 1979, respectively. He has been active in testing, modeling and specifications of both hardware and Software. He gave different short courses to the industry in the domain of synthesis, testing and modeling using VHDL. He has some collaborative work with NORTEL, and CAD houses like Mentor Graphics. He has also collaborative links with Concordia University and Ecole Polytechnique de Montréal. Currently his the director of GRIAO Groupe de Recherche Interuniversitaire en Architecture des Ordinateurs et VLSI). GRIAO regroups more than 20 researchers form 7 Quebec institutions. He is also member of Micronet (Canadian Centre of excellence). He has over 60 publications in journals and international conferences.

Abdeslam En-Nouaary received the Engineering degree in Computer Science from ENSIAS (Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes), Rabat, Morocco, in 1996. He is currently pursuing the Ph.D degree in Computer Science at DIRO (Département d'Informatique et de Recherche Opérationnelle), Université de Montréal. His research interests include specification, implementation and test of real-time systems.

Chourouk Bourhfir is a Ph.D. student in the Département d'Informatique et de Recherche Opérationnelle (DIRO), Université de Montréal. She received the M.Sc. degree in Computer Science in Université Laval, Canada in June 1994. Her research interests include modeling and automatic test generation of embedded systems.