Design for testability: a step-wise approach to protocol testing

Hartmut König[a], Andreas Ulrich[b], Monika Heiner[a]
[a] Department of Computer Science, BTU Cottbus, PF 101344, 03013 Cottbus, Germany, e-mail: {koenig, mh}@informatik.tu-cottbus.de
[b] Department of Computer Science, University of Magdeburg, PF 4120, 39016 Magdeburg, Germany, e-mail: ulrich@cs.uni-magdeburg.de

Abstract
We present an approach to support the design for testability aspect of communication protocols. It combines the ad-hoc techniques partitioning and instrumentation known from integrated circuit testing. A protocol specification is divided into modules of reasonable size. This module structure is preserved in the implementation. Extra test points are added to observe inter-module communication. The test procedure consists of several steps. In the first step, modules are tested separately by applying a powerful test method, whereas following integration tests of modules exploit additional information provided by observers. The application of less sophisticated test methods is propagated for these steps. We show that this testing approach extends testability while fault detection capability is maintained.

Keywords
Network protocols; requirements / specifications; testing.

1 MOTIVATION

Due to the limited power of verification, testing has always been an important method in practice to validate the correctness of communication protocols. Nevertheless, the test of communication protocols has been proven to be difficult and expensive. Reasons are the complexity of communication protocols that makes exhaustive tests impossible as well as the need for complementary tests, e.g. development tests during the implementation phase of a protocol, conformance test to prove the compliance of the implementation with the specification or a protocol standard, interoperability test to demonstrate the ability of implementations to work together, performance test to measure, whether the implementation provides the specified efficiency, and robustness test to prove, whether the implementation behaves stable in erroneous situations.

Up to now, testing aspects are usually not considered during protocol design and protocol implementation. To make sophisticated test methods more efficient and applicable in practical testing, the test process itself has to be reconsidered. This demand is especially enforced by new requirements from high performance communication that require new protocols and communication architectures as well as new implementation techniques [Clar 90]. To make protocol implementations more testable, dedicated techniques and methods have to be applied already during the design phase in order to reduce efforts and costs of testing. In addition, testing aspects should be taken into consideration during the whole protocol development process. Therefore, *design for testability* (DFT) has become an important research topic in protocol engineering.

Testability, in general, is a property of an object that facilitates the testing process [Vuon 94]. It can be obtained in two ways: (1) by introducing special observation features that give additional information about the (internal) behavior of the object, and (2) by a systematic design for testability. The choice of the DFT strategy depends on two factors: the goals of the testing process, and the kind of application.

DFT has been applied in integrated circuit (IC) technology already for a long time. The techniques used there can be divided into two categories [Will 82]: ad-hoc techniques and structured approaches. Ad-hoc techniques solve the testing problem for a given design. They are not generally applicable to all designs. Examples of ad-hoc techniques are partitioning and extra test points. Structured approaches, on the other hand, are generally applicable techniques that are based on a certain design methodology with fixed design rules.

DFT is still a new topic in protocol engineering. It is obvious that some of the approaches worked out in the IC area are also tried to be applied in protocol engineering. First proposals, such as the introduction of points of observation [Dsso 91, 95], can be categorized as ad-hoc techniques according to the classification introduced above. Structured approaches have been not known, yet.

According to [Will 82], DFT comprises a collection of techniques that are, in some cases, general guidelines and, in other cases, precise design rules. Consequently, there will be not only a single approach, but several ones. For the protocol area, this means that the objective of DFT should be to develop a set of approaches that can be applied depending on the test context, the associated cost of implementing them, and the return on investment. Therefore, DFT research should not be limited to a certain test category. It should have a general view and consider all methods that improve the ability of detecting faults during testing and decreasing cost. A selection of specific DFT techniques is needed bearing in mind the benefits they will bring in a given test context.

Starting from this position, we present an testing approach to support DFT of communication protocols that combines the ad-hoc techniques of partitioning a protocol specification into module structures and adding extra test points to observe inter-module communication. The idea of the approach presented in this paper is to use instrumentation not only for getting additional information about the behavior of the implementation under test but also to use this information to decrease the testing efforts by reducing the length of the test suite. The proposed testing procedure is a step-wise one. In the first step, the modules are tested separately by applying a powerful test method, whereas for the following integration tests of the modules (in one or more steps) the application of a less sophisticated test method is propagated to decrease test efforts while fault detection capability is maintained.

The rest of the paper is organized as follows. Section 2 gives a short overview of the proposed testing procedure that is evaluated in more detail in Section 3. Section 4 is dedicated to aspects of multi-module testing and concurrency. Section 5 relates our work to existing ones, and finally, Section 6 concludes the paper.

2 A STEP-WISE TESTING APPROACH – OVERVIEW

The step-wise testing approach proposed in this paper follows the ad-hoc approach in integrated circuit testing [Will 82]. In particular, we use partitioning and adding of extra test points. According to these techniques, we propose to partition a protocol specification into a set of modules of reasonable size which can be executed sequentially and/or in parallel. Such a structuring is natural for protocol design. Most formal description techniques (FDTs) support a certain module structure in the specification, but structuring is usually not used to support testing.

We suppose that the module structure is preserved in the implementation. But we do not make any assumption that the specified inter-module communication is correctly implemented. The inter-module communication, however, is traced by extra test points used as points of control and observation (PCOs) or only as points of observation (POs).

Supposing such a module structure, testing can be executed step-wise in the following manner (cf. Figure 1):

1. *Module testing*: Each module is tested separately. This test is a black-box test. The extra test points associated to the module serve as PCOs. The modules can be considered as software ICs [Hoff 89].

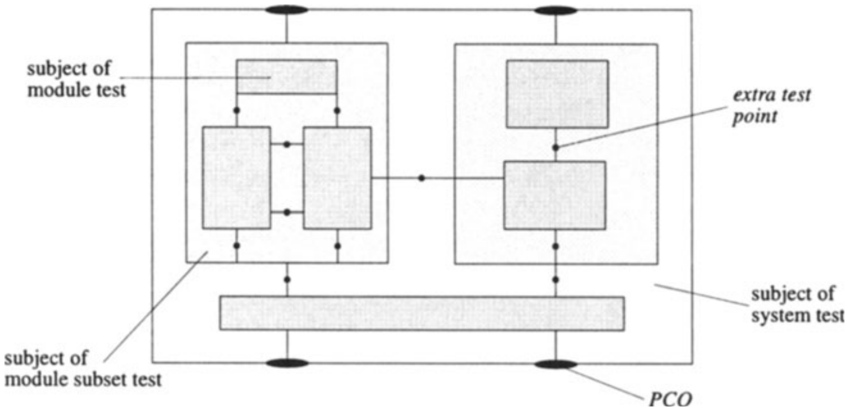


Figure 1 Subjects of test steps for a protocol entity.

2. *Module subset testing*: Reasonable subsets of modules are tested together. The used test method is grey-box testing. Extra test points between modules are POs to observe internal communication between modules.
3. *System testing*: The complete system is tested by integration of all modules and subsets of modules using again grey-box testing as described in the second step.

Steps 2 and 3 are *integration tests* [Myer 79] that test the correct cooperation of the modules, i.e. the correct implementation of the inter-module communication. Step 2 is optional and can be omitted, or may be repeated several times with changing subsets of modules.

The step-wise testing procedure takes advantage of the modularization within the protocol entity. First, each module is tested separately (e.g. by applying the W-method). After that, subsets of modules are tested, and eventually the whole system. Due to the testing efforts already done at module testing level, application of less sophisticated test generation methods is suggested at module subset or system level (e.g. the T-method). The simplification is motivated by the types of faults that can still appear at the second or third testing level (see Section 3.2). The necessary information to find faults that are usually not detectable by a transition tour will be derived from the observation of inter-module communication.

Applying this test strategy we have to show two things: (1) whether the proposed testing approach increases testability, and (2) whether a less sophisticated test generation method in combination with grey-box testing guarantees still high fault coverage. The feasibility of these requirements is discussed in Section 3.

To measure the degree of testability *T*, we apply the measure introduced in [Petr 94] for finite state machines (FSMs) under the complete coverage assumption:

$$T = \frac{o}{mn^2 p^{m-n+1}} \qquad (1)$$

where m is the number of states in the implemented FSM, n is the number of states in the reference FSM, p denotes the number of inputs, and o the number of outputs. In the case that the number of states of the reference FSM equals to the number in the implemented FSM, i.e. $m = n$, the formula is simplified to:

$$T = o/(n^3 p) \tag{2}$$

The measure is proposed to evaluate FSM based module structures, in order to compare different designs with respect to testability. It assumes that testability is inversely proportional to the amount of testing efforts. The latter is proportional to the length of the test suite needed to achieve full fault coverage in a predefined fault domain. Further, it is obvious that an implementation becomes more testable, if more outputs can be observed during testing.

The reduction of the length of a test suite has a larger impact on the increase in testability, since it more effectively cuts test efforts. Consequently, to estimate the increase in testability, we have to show that the average total test suite length of the step-wise testing procedure is shorter than the length of a test suite from the unstructured testing approach.

3 ADVOCATING THE STEP-WISE TESTING APPROACH

In this section, we want to discuss the feasibility of the step-wise testing approach. We suppose that the protocol specification is given in the form of interacting modules as depicted in Figure 1. In order to perform systematic tests, test suites must be derived that are complete to a chosen fault model. A test suite is complete if it can distinguish all faulty implementations among all implementations in the chosen fault model. For example, a complete test suite is produced by the W-method [Chow 78] under the assumption that the number of states in the implementation equals to the one in its specification [Petr 94]. Therefore, we apply the W-method as test generation method for module testing and show that under certain prerequisites the test suite of the less powerful transition tour method (T-method) [Sidh 89] is complete in case of integration test.

For the sake of simplicity, we consider only module testing and system testing. The necessity to introduce further module subset test steps depends on the complexity of the specification. It does not principally change the discussion here, because the procedure is the same as in the system test. It has only to be taken into account for evaluating a concrete test situation.

3.1 Assumptions and basic notations

To follow the sequel of the paper, we introduce some necessary assumptions on the protocol specification as well as some basic notations.

First, we suppose a formal protocol specification as a parallel composition $\mathfrak{S} = M_1 \parallel \ldots \parallel M_k$ of interacting modules. Each module realizes a certain part of the protocol. It is described by a sequential automaton (finite state machine, FSM). Modules communicate with each other solely via interaction points. The communication pattern used is synchronous communication and non-blocking send based on interleaving semantics. Transmitting messages and their receipt through interaction points are referred to *actions*.

To distinguish the different kinds of communication, we denote all inputs and outputs of the protocol implementation from/to the environment as *external*, analogously all inputs and outputs belonging to the inter-module communication as *internal*. Events appearing only inside a module are not considered.

In our discussion, we need to distinguish three types of automata: the module automaton M, the composite automaton CA, and the entity automaton EA.

Module automaton (M)

The module automaton specifies the expected behavior of the module within the protocol entity. It is modeled as a finite state machine.

A ***finite state machine (FSM)*** M is defined by a quadruple $(S, A, \rightarrow, s_0)$, where S is a finite set of states; A is a finite set of actions (the alphabet) consisting of a subset of inputs A_I and a subset of outputs A_O; $A_I \cup A_O = A$; $\rightarrow \subseteq S \times A_I \times A_O \times S$ is a transition relation; and $s_0 \in S$ is the initial state.

A transition $(s_1, a, b, s_2) \in \rightarrow$ with input a and output b is also written as $s_1{-}a_i/b_o{\rightarrow}s_2$. A *trace* denotes a sequence of actions a_i transferring M from state s to state s' and traversing a set of intermediate states: $s{-}a_1/b_1{\rightarrow}s_1{-}a_2/b_2{\rightarrow}s_2{-}\ldots{\rightarrow}s'$. With no loss of generality, we assume that each component FSM is initially connected.

Composite automaton (CA)

The composite automaton specifies the behavior of a subset of modules and of the complete protocol entity. The joint behavior of the multi-module system $\mathfrak{S} = M_1 \parallel \ldots \parallel M_n$ can be described by means of a so-called composite machine defined over $A_{\mathfrak{S}} \subseteq A_1 \cup \ldots \cup A_n$, the (global) alphabet of system $\mathfrak{S}$ that is defined by the parallel composition operator $\parallel$. According to the semantics of this operator, components execute shared actions that require rendezvous of a matching input/output pair of two component FSMs along with local actions that are executed by a component and its environment only.

A ***composite automaton*** of a given concurrent system $\mathfrak{S}$ of k FSMs $M_i = (S_i, A_i, \rightarrow_i, s_{0i})$ is the quadruple $(S_{\mathfrak{S}}, A_{\mathfrak{S}}, \rightarrow_{\mathfrak{S}}, s_{\mathfrak{S}})$, where $S_{\mathfrak{S}}$ is a global state space, $S_{\mathfrak{S}} \in S_1 \times \ldots \times S_k$; $A_{\mathfrak{S}} \subseteq A_1 \cup \ldots \cup A_k$ is the set of actions (the global alphabet), $s_{\mathfrak{S}} = (s_{01}, \ldots, s_{0k})$ is the initial global state. The transition relation $\rightarrow_{\mathfrak{S}}$ is given by the following three transition rules assuming P and Q are two given FSMs, s_P, s_P' and s_Q, s_Q' are states in P and Q, and a, x, b are actions in the corresponding subsets of inputs or outputs of action sets A_P or A_Q.

- If s_P–a/x→s_P' and $x \notin A_{Q_I}$ then (s_P, s_Q) –a/x→$_3$ (s_P', s_Q).
- If s_P–a/x→s_P' and s_Q–x/b→s_Q' then (s_P, s_Q) –$a/x/b$→$_3$ (s_P', s_Q').
- If s_Q–x/b→s_Q' and $x \notin A_{P_O}$ then (s_P, s_Q) –x/b→$_3$ (s_P, s_Q').

The notation of a global transition s_3 –$a/x/b$→$_3$ s_3' illustrates that after input a has occurred, internal action x between two modules is exchanged and output b is produced finally.

Entity automaton (EA)
The entity automaton specifies the global, observable behavior of the protocol entity. It can be derived formally from *CA* by restricting the global alphabet A_3 to the set of actions observable by the environment of the protocol, i.e. internal communication between modules is suppressed in the description of *EA*. The notion of an entity automaton is introduced here merely for the purpose of comparison.

3.2 Fault model

Now we discuss the types of faults that may appear in a faulty multi-module implementation. We assume that the specification has been verified to be correct. That means, there are no deadlocks or unreachable states in the specification.

Fault model of the module automaton
In our test approach, the module test is a black-box test, in which a test suite is applied that is complete to the fault model of a single module. We suppose in the following discussion that the single modules have been successfully tested and that they behave as specified.

Fault model of the composite automaton
At the level of integration tests the following faults are still possible*:

- *Data flow faults:* Data exchanged between modules may be faulty. This is a common implementation fault. Testing related to data flow is still a partly unsolved issue for which only specialized solutions have been found [Guer 96]. The observation of the inter-module communication can in part detect data faults. This may be in some cases useful because inter-module communication often consists of simple data structures as, for instance, signals that inform about

* Faults in module interactions that do not appear in an appropriate sequence or even incorrect sequences of actions can be considered as design faults of the protocol. They can be found by static analysis of the composite automata concerning communication inconsistencies (e.g. on the basis of Petri net analysis [Hein 92] [Ochs 95]). Synchronization faults due to a change in the communication pattern from synchronous to asynchronous communication or vice versa are not considered here since once the communication principle has been selected in the design phase of the protocol, it should remain the same throughout the design trajectory.

a state achieved in the sending module or transfer data as credit information. According to our approach false outputs, i.e. data of a wrong type, are detected during module test. Faults in the data flow caused by false values of components of the data are not considered in our discussion.

- *Coupling faults among modules*: Inter-module communication can be implemented by different means, e.g. procedure calls, shared variables, communication channels or others. It is also often a source for faulty implementations. Coupling faults appear if interaction points of the modules are erroneously connected with each other, i.e. the output of a module is sent to a wrong module that is, however, able to consume this event performing a corresponding input event. This type of fault must be detected during integration test. A coupling fault can be reduced to a state fault in the composite automaton.

3.3 Feasibility of the approach

To justify the step-wise testing approach, we have to show that

- the average total length of the test suite for the step-wise approach is shorter than the length of the test suite derived from the entity automaton;
- the fault coverage of the step-wise approach is the same as for the conventional approach based on the single entity automaton, i.e. all possible faults that can be detected in the conventional approach shall be detected by the step-wise approach, too.

Let A be a finite state automaton, *length*(*W*(*A*)) is the test suite length of the W-method applied to A, *length*(*T*(*A*)) is the test suite length of the T-method applied to A. The conjecture is that the following equation holds for a suitable number k of modules in the specification:

$$length(W(EA)) > \sum_{i=1}^{k} length(W(M_i)) + length(T(CA)) \quad (3)$$

The formula means that the total length of the test suite applied in the step-wise approach is shorter than the length of the test suite that would be derived from the entity automaton EA.

According to the formula, we have to show that the total length of test suites in a step-wise approach is generally shorter than the length of the test suite derived from the monolithic entity automaton. We demonstrate that this statement holds for the case of an equal number of states in the implementation and the specification.

To test the entity automaton, the W-method is applied since it produces a complete test suite in the fault model of implementations with an equal number of states. The number of states in the entity automaton EA can be estimated in the worst case by $n_{EA} \le \prod n_i$, where n_i is the number of states of module M_i. This estimation also assumes that automaton reduction applied when constructing the entity automaton does not contribute to a reduction in the number of states, i.e. all global states in the reduced composite automaton are distinguishable. Thus, the length of the W-method is bounded to $O(p_{EA} \cdot n_{EA}^3) = O(n_{EA}^4) = O(n_1^4 \cdot \ldots \cdot n_k^4)$ if we assume that the number of transitions is nearly the same as the number of states.

On the other side of the formula, we have a finite sum of the length of the test suites for all single modules (W-method) plus the length of the test suite for the composite automaton (T-method): $O(p_1 \cdot n_1^3) + \ldots + O(p_k \cdot n_k^3) + O(p_{CA} \cdot n_{CA}) = O(n_1^4) + \ldots + O(n_k^4) + O(n_{CA}^2)$. The number of states n_{CA} in the composite automaton is also bounded to the product of the number of states of individual modules: $n_{CA} \leq n_1 \cdot \ldots \cdot n_k$.

Since the length of the T-method is reduced by the power of 2 compared to the W-method, and a sum of numbers greater than 1 is always less than their product, it follows that the total length of test suites in the step-wise approach is shorter. It implies that testability increases according to the testability metric from [Petr 94] quoted in Section 2. In addition, testability will be further improved by the number of events additionally observed at points of observation.

Now we turn to the second requirement of our approach. We have to show that the transition tour in combination with the use of extra test points is a complete test suite for integration test. As known, a transition tour is only capable to indicate output faults (caused by erroneous inter-module communication), but not to detect wrong states. However after the module test has been carried out successfully, i.e. the correctness of the module implementations was verified, we can assume that wrong states in the composite automaton can only occur as result of coupling errors. Therefore, we must show that the transition tour together with the observation of inter-module communication will be capable to detect wrong coupling of modules.

The detection of these faults depends on the way how the observation of the inter-module communication can be performed. A pragmatic approach for realizing this observation would be to implement the extra test points in a such a way that the gates of the modules send the information to the observer, which data have passed. Thus, wrong data and coupling errors can be very easily detected, because the way the transition tour has taken in the integration test can be traced. But it would require that the implementations of the modules support extra test points. This approach influences the implementation and is therefore not feasible. We suppose in the following that the extra test points do not influence the module implementation. They can only "see" the data sent by the modules.

A coupling fault may appear, if there exist two equivalent traces Tr_{i1} and Tr_{i2} between a state s_{im} in module *m* and a state s_{jn} in module *n* such that the transition tour can follow another way. Since we know from the module test that the local actions are correctly implemented, the selection of another way can only be forced by a wrong coupling between modules. If we can prove that all traces between two CA states that include inter-module communication are distinguishable, then coupling faults in the composite automaton will lead to sequences of internal and external outputs that do not correspond to traces of the specified automaton.

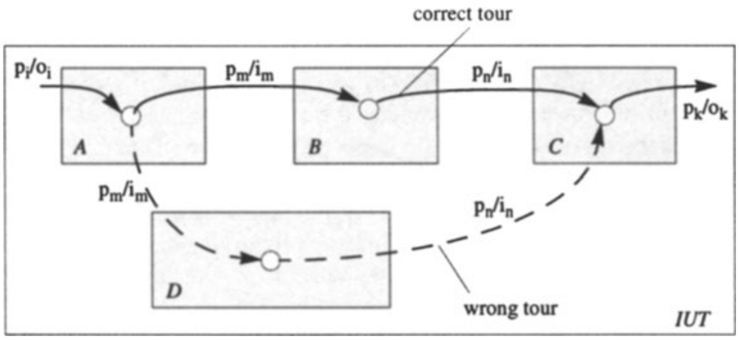


Figure 2 Wrong tour due to coupling error.

Let us now suppose that there exist a coupling fault between two modules and that the observed trace Tr_{i1} between s_{im} and s_{jn} coincides by chance with another trace Tr_{i2} between the two CA states. This is only possible if the states of the module automata passed by Tr_{i1} possess a same transition as Tr_{i2}. Figure 2 depicts this situation. In this case, a transition tour cannot detect without any further information the wrong coupling. To exclude this situation, we have two choices:

1. To make the states of the modules that are involved in inter-module communication distinguishable at the receiving side. This can be done by analyzing the specification for such states in advance and to introduce an additional loop transition back to the same state in the specification and implementation of these states. The transition tour executes the additional transition to validate that it has reached the correct state.
2. To use distinguishable messages for inter-module communication, i.e. the shared actions in A_3 are unique. In this case a data error will be observed.

If such a measures are accepted for DFT purposes, a transition tour is a complete test suite for integration test.

Example

To illustrate the above discussion, we consider the XDT protocol [Koen 96]. XDT (*eXample Data Transfer*) is an example protocol used for teaching protocol engineering. It provides a connection-oriented data transfer service based on the *go-back-N* principle. In our discussion we only consider the sender part. The sender starts with an implicit connection set up *(XDATrequ)*, which is indicated to the service user by a *XCONconf* when finished successfully, otherwise the attempt is stopped by an time-out (*to_t1*). After that the service user can continuously send (*XDATrequ*). The sending may be interrupted *(XBRKind, XBRKend)*, when the buffer for storing the *DT*-PDU copies is full. The sender repeats the transmission of a *DT*-PDU and the following (already sent) ones (*go-back-N*), when the transmission of a *DT* is not confirmed by an *ACK*-PDU within a certain time (*to_t2*). The connection is released (*XDISind*) after confirming the successful transmission of the last *DT*-PDU. The transmission can be aborted with an *ABO*-PDU by the receiver (indicated to the service user by a *XABOind*), when the PDU sequence is not reestablished in a reasonable time. The FSM of the sender entity is depicted in Figure 5 in the appendix.

To estimate the testability of the sender FSM we use the measure from [Petr 94] (see Section 2). The sender FSM has 5 states, 8 inputs and 5 outputs. The upper bound of the length of the test suite when applying the W-method will be $8*5^3 = 1000$, and the testability degree is $5/1000 = 0.005$.

We now divide the specification according to their logical function in 3 modules *M1*, *M2* and *M3* (see Figure 3 and Figure 4 in the Appendix). Module *M1* performs the connection set-up, *M2* the data transfer and *M3* supervises the acknowledgments. It also initiates the *go-back-N* mechanism and accepts the *ABO*-PDU. For inter-module communication the internal events *i1*, *i2*, *i3*, *i4*, *i5*, *i6* are introduced. The upper test suite lengths for each of the 3 modules, when applying the W-method, are $3*2^3 = 24$ (*M1*), $8*4^3 = 512$ (*M2*) and $5*2^3 = 40$ (*M3*). The upper length of the transition tour is $8*2*4*2 = 128$ (with 8 external inputs). The maximum length of the test suite is therefore 704 test events. The testability degree is $11/704 = 0.0156$ (with 11 internal and external outputs), i.e. the testability increases remarkably. The length of the transition tour for the system test can be even further reduced, because module *M1* terminates before the other two modules start. This knowledge from the specification could be also exploited in the step-wise test approach.

4 CONCURRENT MODULE STRUCTURES

In this section, we discuss the application of the step-wise testing approach for a protocol specification and its corresponding implementation, in which modules are executed concurrently. The assumption of true concurrency is realistic for protocol implementations. However, testing implementations based on multi-module specifications is complicated by a number of problems that are unique to the nature of concurrent systems. Under these problems the most important ones are the occurrence of concurrent events during testing; the reproduction of test runs with the same test data; and state explosion that occurs when the system is being analyzed.

A conventional approach to test suite generation starts from a monolithic, single automaton, i.e. from the entity automaton in our case. Since the entity automaton is usually not given in advance, it must be constructed, e.g., by computing the product of the module automata using interleaving semantics rules to obtain the composite automaton and reducing the composite automaton eventually to obtain its reduced automaton that equals to the entity automaton. The generation of a transition tour from the interleaving model of an entity automaton has its limitations since concurrent events are serialized. Due to a lack of controllability during testing, this approach is not feasible. The resulting order of concurrent events in a test run could not be predicted. The order of events is, however, essential to assess whether an implementation is correct.

If we apply the proposed step-wise testing approach, we are able to use the structure information given as a set of communicating modules during test suite generation. In [Ulri 95], we extended the notion of a *transition tour* [Sidh 89] and applied it as a test suite for distributed systems. A transition tour is defined for a single automaton as the shortest path that covers all transitions in the automaton at least once. In the context of distributed systems a transition tour is extended to a *concurrent transition tour* (CTT) such that all transitions in all modules of the system are visited at least once on the shortest possible path through the system. A CTT takes concurrency among actions of different modules into account.

A CTT is depicted graphically as a time event sequence diagram where nodes are events and the directed arcs define the causality relation between events. It can be considered as a set of local transition tours TT_i through the single modules of the system by taking into account synchronization constraints, i.e. $CTT = (TT_1, \ldots, TT_k)$. Its construction, however, does not necessarily follow from this definition. A feasible construction algorithm of a CTT is presented in [Ulri 97].

The actual length of a concurrent transition tour depends on the degree of concurrency among the modules. The lowest bound of the length is determined by the least common multiple of completed cycles of single transition tours through the modules if no branching occurs at all. In the worst case, the length of the concurrent transition tour equals to the length of a transition tour derived from the interleaving model, i.e. $length(CTT) \leq length(TT)$. Thus, using concurrent test sequences instead of interleaved based ones reduces test efforts further.

5 RELATED WORK

Design for testability is a relatively new approach in protocol engineering. It aims at decreasing the efforts in protocol testing and supporting a better detection of faults in implementations. The testability of protocols may be influenced by many factors in the context of design, implementation and testing. Dssouli and Fournier have, therefore, first proposed to introduce DFT as a development step in the protocol development process [Dsso 91]. A general framework for DFT for protocols was given by Vuong, Loureiro, and Chanson in [Vuon 94].

Grey-box testing is considered as the preferred approach to increase testability. Theoretical aspects of grey-box testing have been pioneered by Yao, Petrenko, Bochmann, and Yevtushenko [Yao 94] [Yevt 95]. A metric for testability based on finite state machines under the complete fault coverage assumption was proposed by Petrenko, Dssouli, and König [Petr 94]. Most approaches that follow this way use means to instrument the implementation with extra test points in order to observe the behavior of the implementation under test. A framework for this approach is proposed by Dssouli, Karoui, Petrenko, and Rafiq in [Dsso 95]. A generic scheme to automatically instrument a formal specification is described by Kim, Chanson, and Yoo in [Kim 95].

A similar incremental approach to structural testing was first proposed by Koppol and Tai in [Kopp 96]. Here, the incremental approach is used to alleviate state explosion during the derivation of test cases for a concurrent system using interleaving semantics. They establish test derivation on structural test coverage criteria, e.g. the coverage of every transition in the modules of the system at least once, instead of providing a fault model, and they do not discuss the degree of testability of their approach.

The work on a concurrent transition tour as a test suite for distributed systems [Ulri 95, 97] can be regarded as an alternative approach to test derivation to alleviate state explosion. It has been advocated by approaches on trace analysis [Yang 92] [Kim 96]. These approaches assume that valid sets of traces through the modules, i.e. valid execution sequences of the system, are already given, but do not provide methods to derive them according to a certain fault coverage. Since a concurrent transition tour requires a grey-box approach in testing to avoid nondeterminism in distributed systems, the test method proposed in this paper follows immediately.

6 CONCLUSIONS

We have presented an approach to support design for testability for communication protocols. The approach combines a step-wise test procedure with grey-box test principles. Applying the approach, we have to consider two further aspects.

First, an appropriate module structure of the protocol specification has to be found. Its design depends often on subjective decisions made by a designer. However, protocols themselves support modularization in most cases. They usually consist of several protocol phases represented by separated (partial) services. These phases can be designed as different subsets of modules and implemented and tested separately. Such a modularization is also supported by the standardized FDTs.

In addition, a test architecture has to be provided that supports the step-wise testing approach. Extra test points must be designed in such a manner that they can be used as PCOs for module tests and POs for integration tests. Their inclusion should be automated as proposed in [Kim 95].

Nondeterminism is a real issue in testing concurrent systems as it was shortly pointed out in Section 4. This problem is aggravated further since additional forms of nondeterminism may exist in a concurrent system, due to nonobservability of internal interactions or data races, even if all its modules behave deterministically. In this case, only a grey-box testing approach and further measures must be taken into account to guarantee a deterministic test run [Tai 95].

Up to now, the step-wise testing approach has been elaborated and justified for concurrent modules communicating synchronously. However, work on an extension of the current method to asynchronous communication is needed. Furthermore, any impact of data flow on the internal behavior of modules has been neglected. A more sophisticated grey-box test procedure is needed to trace the influence of data exchanged over communicating modules. Suggestions for related techniques that are probably applicable in the area of protocol engineering are already known from software engineering of parallel processes (see e.g. [Lebl 87]).

Our approach also facilitates interoperability test because separate accessible modules can be tested against each other. The additional information obtained from POs supplements the test data recorded by a test monitor. Thus, these tests are useful in particular for locating faults when the interoperability test was not successful.

7 REFERENCES

[Chow 78] Chow, T. S.: *Testing Software Design Modeled by Finite-State Machines;* IEEE Trans. on Software Engineering (1978) 3,178-187.

[Clar 90] Clark, D. D.; Tennenhouse, D. L.: *Architectural Considerations for a New Generation of Protocols;* Proc. ACM SIGCOMM, 1990, 200-208.

[Dsso 91] Dssouli, R.; Fournier, R.: *Communication Software Testability;* In Davidson, I.; Litwack, W. (eds:): Protocol Test Systems III. North Holland, 1991, pp. 45-55.

[Dsso 95] Dssouli, R.; Karoui, K., Petrenko, A.; Rafiq, O.: *Towards testable communication software;* Proc. IWPTS'95, Paris, 1995, pp. 239-255.

[Guer 96] Guerrouat, A.; König, H.; Ulrich, A.: *SELEXPERT - A knowledge-based tool for test case selection;* In Formal Description Techniques VIII, Chapman&Hall, 1996, pp. 313-328

[Hein 92] Heiner, M.: *Petri Net Based Software Validation, Prospects and Limitations;* ICSI-TR-92-022, Berkeley/CA, 3/1992.

[Hoff 89] Hoffman, D.: *Hardware testing and Software ICs;* Proc. Pacific NW Software Quality Conference, Portland, 1989, pp. 234-244.

[Kim 95] Kim, M.; Chanson, S. T.; Yoo, S.: *Design for testability of protocols based on formal specifications;* Proc. IWPTS'95, Paris, 1995, 257-269.

[Kim 96] Kim, M. C.; Chanson, S. T.; Kang, S. W.; Shin, J. W.: *An approach for testing asynchronous communicating systems*; 9th IWTCS'96, Darmstadt, Germany; 1996.

[Koen 96] König,H.: The XDT Protocol. Technical Report 04-96. BTU Cottbus, Fakultät Mathematik,Naturwissenschaften und Informatik,1996.

[Kopp 96] P. V. Koppol, K. C. Tai: *An incremental approach to structural testing of concurrent software*; International Symposium on Software Testing and Analysis (ISSTA'96); San Diego, California; 1996; pp. 14–23.

[Lebl 87] Leblanc, T.; Mellor-Crummey, J. M.: *Debugging Parallel Programs with Instant Replay;* IEEE Trans. on Computers 36 (1987) 4, 471-482.

[Myer 79] Myers, G. J.: *The art of software testing;* John Wiley & Son, 1979.

[Ochs 95] Ochsenschläger, P.; Prinoth, R.: *Modelling of distributed systems*; Vieweg, 1995, (in German).

[Petr 94] Petrenko, A.; Dssouli, R.; König, H.: *On Evaluation of Testability of Protcol Structures;* In Rafiq, O. (ed.): Protocol Test Systems VI, North Holland,1994, pp. 111-123.

[Sidh 89] Sidhu, D. P.; Leung, T. K.: *Formal methods for protocol testing: a detailed study;* IEEE Trans. on Software Eng. 15 (1989) 4, 413–426.

[Tai 95] Tai, K. C.; Carver, R. H.: *Testing of distributed programs*; in A. Zomaya (ed.): Handbook of Parallel and Distributed Computing; McGraw Hill; 1995.

[Ulri 95] Ulrich, A.; Chanson, S. T.: *An approach to testing distributed software systems;* Proc. 15th PSTV'95; Warsaw, Poland; 1995.

[Ulri 97] Ulrich, A.: *A description model to support test suite derivation for concurrent systems*; in M. Zitterbart (ed.): Kommunikation in Verteilten Systemen, GI/ITG-Fachtagung; Springer Verlag, 1997.

[Vuon 94] Vuong, S. T.; Loureiro, A. A. F.; Chanson, S. T.: *A Framework for the Design for Testability of Communication Protocols;* In Rafiq, O. (ed.): Protocol Test Systems VI, North Holland, 1994, pp. 89-108.

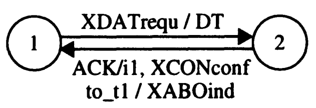
[Will 82] Williams, T.W.; Parker, K. P.: *Design for Testability - A Survey;* IEEE Trans. on Computers C-31 (1982) 1, 2-15.

[Yang 92] R. D. Yang, C. G. Chung: *Path analysis testing of concurrent programs*; Information and Software Technology; 34(1992)1, 43–56.

[Yao 94] Yao, M., Petrenko, A., Bochmann, G.: *A structural analysis approach to evaluating fault coverage of software testing in respect to the FSM model*; Proc. 7th FORTE'94; Bern, Switzerland; 1994.

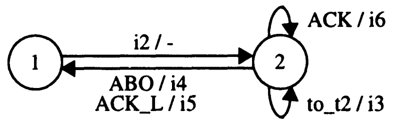
[Yevt 95] Yevtushenko, N.; Petrenko, A.; Dssouli, R.; Karoui, K.; Propenko, S.: *On the Design for testability of Communication Protocols;* Proc. IWPTS'95, Paris, 1995.

8 APPENDIX



p = {XDATrequ, ACK, to_t1}

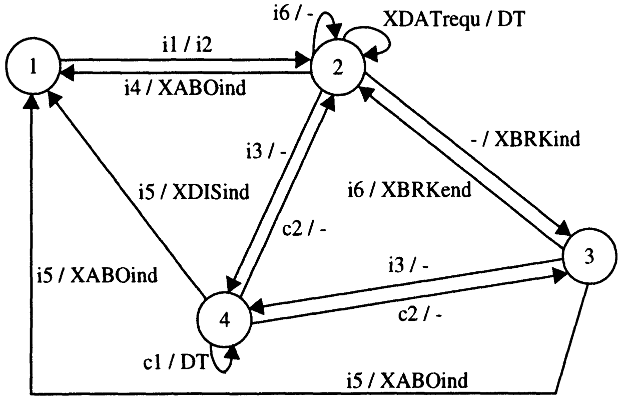
o = {DT, XABOind, XCONconf, i1}



p = {i2, ABO, ACK, ACK_L, to_t2}

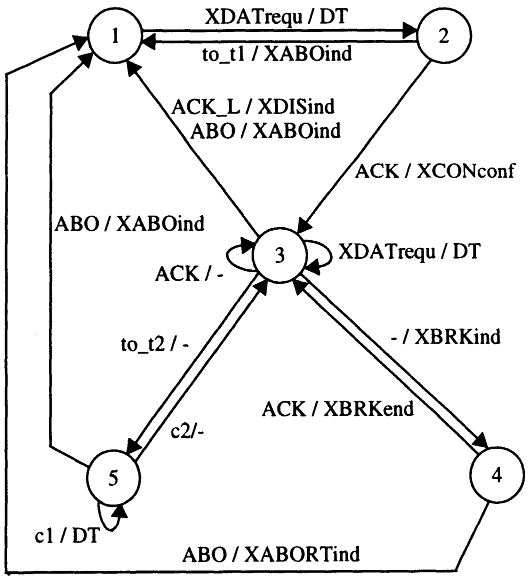
o = {i3, i5, i6}

Figure 3 FSMs *M1* and *M3*.



p = {i1, i3, i4, i5, i6, c1, c2, XDATrequ}
o = {DT, XBRKind, XBRKend, XDISind, XABOind, i2}

Figure 4 FSM *M2*.



p = {XDATrequ, ACK, ACK_L, ABO, to_t1, to_t2, c1, c2}
o = {DT, XDISind, XABOind, XBRKind, XBRKend}

Figure 5 FSM of the XDT sender.