An approach to testing distributed software systems

Andreas Ulrich^{}, Samuel T. Chanson*
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
e-mail: {ulrich, chanson}@cs.ust.hk

Abstract

This paper studies the problem of testing distributed software systems consisting of concurrent units running over a network. We present an approach to derive a concurrent transition tour as a test suite from multi-module specifications described by a restricted class of CCS (calculus of communicating systems). The approach avoids the state space explosion problem by introducing a true concurrency model. A new distributed test architecture for executing the concurrent test suite is also described.

Keywords

Distributed testing; formal specifications; true concurrency model; test suite generation; distributed test architecture.

1 INTRODUCTION

Current work in protocol engineering is moving more and more from the lower layers of the OSI reference stack upwards to the application layer. Whereas lower layer protocols are reduced to a few standardized ones, the higher level protocols, especially the application protocols, vary greatly according to the different applications such as multi-media applications and distributed data base servers. The new requirements introduced by the new, distributed applications increase the complexity of designing, implementing, and testing these systems.

^{*} On leave from Dept. of Computer Science, Otto-von-Guericke-Universität Magdeburg, Germany.

An inherent property of these systems is their realization by concurrent processes distributed over a computer network. In the context of testing system behaviors, concurrent modules produce a huge amount of possible combinations of actions among them, resulting in the well known state space explosion problem. This phenomenon offers a major challenge in generating test suites to test such systems and for the testing process itself.

Existing approaches for test suite generation based on the traditional FSM-based methods (e.g. UIO method, W method, and others) [Bos91] [Luo94] [Aho88] mostly neglect the case of multi-module specifications. Either a single module is assumed or the multiple modules are first combined into a single module before processing.

Another approach to testing is based on the trace analysis of concurrent communicating systems [Kim93]. It assumes that a set of traces of all units in the system is already given and defines the properties of concurrent actions by applying the *happened-before* relation which captures causality information between them. A description of how concurrent traces can be derived from a formal specification and the properties they must satisfy to be valid test suites for the concurrent system were not presented.

This paper closes the gap between the formal specification of the system on the one hand and the application of concurrent action traces as test suites on the other hand by giving an algorithm to derive a *concurrent transition tour* covering all executable transitions of all modules in the specification.

Furthermore, the paper presents a novel approach to test execution in a concurrent environment. In this environment, the occurrence of concurrent actions renders the application of conventional test methods and test architectures inadequate because of the lack of controllability among the concurrent units in the implementation under test (IUT). To tackle this problem, a new distributed test architecture is introduced which supports the application of a concurrent transition tour in a test run.

1.1 Model of distributed systems

We first present our view of a distributed system. A distributed system can be described by a set of sequential units which run concurrently over a network. Each of the units realizes a basic function of the distributed system. In the distributed system, no global clock is assumed to be available.

Units interact with each other via message passing through interaction points. The types of coupling between them that may occur are:

- (a) synchronous (rendezvous principle) or
- (b) asynchronous (via FIFO queues).

Using standardized FDTs, like Lotos or Estelle, a concurrent unit is described by a set of one or more Lotos processes or Estelle modules.* We also assume that the implementation follows the same structure of the specification, i.e. there is a one-to-one correspondence between a module and a unit.

* To avoid ambiguity, we will use the term *unit* to refer to an implementation of a sequential module, and the term *module* to describe a structural construct in the FDT specification (i.e. a Lotos process or an Estelle module).

1.2 Model assumptions

Because of the complexity in distributed testing, some restrictions are imposed to simplify the model.

First, the system to be tested is assumed to be described in CCS with synchronous communication among the modules [Mil89]. The CCS specification is considered to be correct. The restriction to synchronous communication makes the system easier to analyze.

Second, the modules described in CCS should be strongly connected. This is a reasonable assumption since systems satisfying this assumption are usually more robust systems which are able to continue execution after an error has occurred.

A third restriction is that the structure of the system is assumed to be static. Dynamic creation and termination of units are not allowed, i.e. all units of the system must already exist when the test run is started, and no new unit can be created or existing unit terminated during the run.

1.3 Outline of the paper

The rest of the paper is organized as follows. Section 2 discusses the problems related to the testing of distributed systems and defines the notion of conformance in this context. Furthermore, the causality-based semantics which serves as a vehicle for generating test suites is introduced. Section 3 deals with the test suite derivation process. The efficiency of the approach is shown by means of an example. Section 4 discusses the issues in applying the generated test suite in a distributed test architecture. Section 5 gives a short overview of related work, and Section 6 concludes the paper.

2 PRELIMINARIES

2.1 The testing problem

Interleaving actions

The most severe problem facing the testing of distributed systems is the state space explosion problem. The major reason is the interleaving of actions at the concurrent units. The conventional approach to handle this problem is to generate all possible combinations of actions resulting in a huge amount of actions to be considered and heavy memory requirements.

Global states in a distributed system

The conventional interleaved-based approach to construct a single finite state machine (FSM) from a system of communicating FSMs generates a lot of global states which are, under the assumption of concurrency between the FSMs (the modules in the specification), not all consistent. Consistent global states only exist along the so-called *global synchronization cuts* of the distributed system (see papers [Mat89] and [Kim93] for more explanations). The test of consistent

global states for correctness would increase the confidence in the correctness of the entire system. The issue of testing global states in the context of distributed systems is still an open problem and requires further work. In this paper, we shall consider an approach to transition testing.

2.2 The Notion of conformance in distributed systems

Testing in general means validation of an implementation against its specification, i.e. to check if the function of the system described by the specification is correctly implemented.

Conformance testing, as a special case, tests the entire system using a black-box approach. A commonly used definition of *conformance* in the context of distributed systems is as follows:

Definition (1): A distributed system comprising multiple concurrent units conforms to its specification if the system performs the observable function defined by the specification. This function is described by the total order of observable actions.

According to this definition, the internal communication between units does not play a role in conformance. Only the outermost behavior of the specification, which is observable by the environment, must be fulfilled by the implementation. This definition of conformance is not sufficient in the context of distributed systems because it allows different interpretations of the communication between the units, some of which may lead to deadlocks, livelocks or data races. However, these errors may or may not occur in a test run due to nondeterminism in the total order of actions [Vuo93].

Another mode of testing is *diagnostic testing* which uses the white-box approach. This mode is very restrictive because it requires access (and possible modification) to the source code of the implementation, which is generally available only to the developers during program development. Therefore, we have adopted the grey-box approach to provide a general testing framework and still ensure conformance to the specification including the absence of deadlocks, data races and other possible error classes in a distributed system.

Definition (2): A distributed system conforms to its specification if all actions among its units as well as the actions between the units and the environment of the system conform to the total order of actions allowed by the specification.

In the grey-box approach, each unit is treated as a black box, but the communication among the units and the communication between the units and the environment of the system can be observed and/or controlled by a tester.

2.3 Error classes in a distributed system

We assume that a distributed system is modeled using the assumptions and restrictions described in Sections 1.1 and 1.2. The following types of errors may occur in an implementation which is derived from the given specification:

1. *Transition faults in a unit:* After an input event, the unit generates a wrong output event. (This is equivalent to the definition of mutant transitions.)

2. *State faults in a unit*: The transition is correct, but it ends in a wrong state.
3. *Deadlocks*: a) A unit receives an unexpected message which cannot be handled by it; b) two or more units are waiting mutually for messages which will never come (circular wait).
4. *Livelocks*: Due to non controllable interactions between units, there may exist infinite cycles between units which do not require any input from outside.
5. *Data races*: Concurrent access to local variables in a unit yields unexpected or random values of these variables.

The first two error classes are covered by traditional test methods [Sid89]. Note that the detection of state faults requires a specification model which models states explicitly. The error classes introduced in [Sid89] can be viewed as combinations of the two classes mentioned here.

The other three error classes only occur in a system of communicating units. The single units behave correctly, but the coupling among them may be faulty. The absence of these error classes (3 to 5) in a specification can be easily verified by applying interleaved-based semantics [Hol94]. Their occurrence in an implementation, however, may be detected only by chance in a test run unless a tester can control the internal actions [Kim94].

We argue that by testing a system using the grey-box approach, the probability of detecting these error classes is higher than applying black-box testing assuming that the specification is error-free and a one-to-one correspondence between the modules of the specification and the implemented units as described in Section 1.1 is preserved.

2.4 Introduction to true concurrency semantics

The construction of a total order of actions from interleaving actions makes sense only in the verification process of a specification, e.g. to show the absence of errors under all possible combinations of actions. However, interleaving leads to state space explosion. In the area of testing an implementation against its specification, the module structure has to be taken into account, and the duration of actions can no longer be neglected. The solution to this problem is a true concurrency model for the description of distributed systems.

In the literature, different semantics models expressing true concurrency have been proposed ([Lan94] [Fid92] [Lan92]). We favor the causality-based approach presented in [Coe92a] and [Coe92b] because this approach uses the CCS model by preserving branching information which is sometimes neglected by other approaches. We apply the causality-based model in order to express infinite behaviors expressed in terms of CCS by a finite representation of a set of events.

Based on the assumptions specified in Section 1.2, only the following CCS operators are needed: action prefix ($a.B$), summation ($E + F$), parallel composition ($E \mid F$), and recursion ($\text{rec } X.E$). According to the grey-box approach we also omit the operator for action concealment. Each CCS term can be expressed by a corresponding *labeled transition graph* defined by a *labeled transition system* (LTS) [Mil89].

The basic element in this approach is an *event*. An event e is a triple $e = pa_x \in \wp(X) \times A \times X$, where $P \in \wp(X)$ is a set in the power set of event names defining the set of predecessors of e ; a is an action in the set of actions A ; and x is the event name from the set of event names X .

Instead of interpreting causality information in the context of an interleaved-based model, we apply the notation of *labeled partially ordered sets* (*lposets*) which allows an interleaved-free representation of concurrent behavior [Pro91]. A *lposet* is defined by the quadruple $(E, A, <, l)$ where

E is a set of event names; A is a set of action names; $<$ is a partial order expressing the causality information between events (i.e. it is reflexive, transitive and asymmetric); $l: E \rightarrow A$ is a labeling function mapping each event to an action.

The relation $e_1 < e_2$ holds between the two events if e_1 precedes e_2 . More formally, $e_1 < e_2$ if there is a sequence of event names such that $x_{e_1} \in P_{e_i} \rightarrow x_{e_i} \in P_{e_j} \rightarrow \dots \rightarrow x_{e_k} \in P_{e_2}$ with a suitable index set I and $i, j, k, \dots \in I$, and x_e and P_e denoting the event name and the set of predecessors of event e respectively. If there is no relation between two events in one $lposet$, the events are concurrent to each other.

An action $a \in A$ corresponds to an action in a module description. The labeling function l should be globally unique in order to distinguish between all actions in the system.

A $lposet$ can be represented graphically by a time-event sequence diagram where nodes are events and the directed arcs define the causality relation between events. The time axis is assumed to be running from top to bottom (or from left to right). A concurrent test suite can be expressed using a set of $lposets$ as the underlying model.

Let us consider the following CCS term as an example:

$$A = (a.b.c.A) \mid (a.d.A).$$

Under the assumption that the actions a in each sub-term synchronize, removal of the parallel operator ‘ $\mid$ ’ by applying the interleaved-based semantics rules yields three possible sequences which are heavily redundant in the sense that each contains the same actions in different total orders:

$$A = a.(d.b.c.A + b.d.c.A + b.c.d.A).$$

Whereas by using the causality-based rules in connection with the notation of $lposets$, the behavior can be expressed as:

$$\{\{3\}a_1, \{1\}b_2, \{2\}c_3, \{1\}d_3\},$$

where the index of an event refers to an event name, and the set of prefixes is associated with the predecessors of the event. The meaning of the given $lposet$ is that action a is followed by actions b or d , and action c can happen only if b has happened before it. Furthermore, it can be realized that the actions b and c are concurrent to action d . The $lposet$ is presented graphically in Figure 1.

Figure 1 Time-event sequence diagram of A .

This representation provides the capacity to describe recursive behavior. For example, in Figure 1, after actions d and c have occurred, the initial action a can happen again as required in the given specification. It is easy to see that a concise description of the partial order of actions in a concurrent system can be obtained.

3 TEST DERIVATION FOR DISTRIBUTED SOFTWARE SYSTEMS

Now we are ready to present a methodology for deriving test suites from a given CCS specification. The system specification S is given in CCS as a set of modules M_i composed together via the parallel operator ' $|$ ':

$$S = M_1 | M_2 | M_3 | \dots | M_n.$$

In this specification, the CCS parallel operator ' $|$ ' can be interpreted as a concurrency operator ' $|_c$ '. The grey-box approach is used in the scheme.

We derive a concurrent transition tour which satisfies the following Definition (3):

Definition (3): A *concurrent transition tour* for a distributed system is defined by a set of concurrent action traces with the property that a trace represents a sequence of consecutive actions of one unit, and each sequence comprises all allowed actions in that unit at least once.

Definition (3) is analogous to the definition of a transition tour for a single FSM [Sid89]. In general, the number of concurrent transition tours which satisfy Definition (3) can be infinite. In the rest of this section, we give an algorithm which finds the shortest concurrent transition tour of a given specification.

3.1 An example

In the following we derive a concurrent transition tour from a specification $S = A | B$ which consists of two LTS modules A and B (see Figure 2) without interleaved-based combination of the two modules into a single module. The corresponding CCS terms of A and B are

$$A = a.x.y.b.A \text{ and } B = a.B' \text{ with } B' = u.B' + v.b.B.$$

Figure 2 The example specification of two concurrent transition graphs A and B .

In Figure 2, the nodes represent states and the arrows denote action transitions between states. The initial state is always state 0.

3.2 The test suite derivation algorithm

The algorithm can be separated into two phases:

1. Generate a representation of the modules by enriching them with causality information on the actions;
2. Obtain a concurrent composition of the modules and derive a concurrent transition tour.

Phase (1): Generation of module representation

A sequential module M is described by a finite set of CCS terms $M = \{M', M'', \dots, M^{(n)}\}$. The set of all modules in the specification is denoted by $\mathbb{M} = \{M_1, M_2, \dots, M_n\}$.

Each event is represented by a triple $e = pa_x$, where P is the set of predecessor events; a is the action name; and x is the event name.

Due to the properties of the causality-based approach, each sequential module can be expressed simply by a set of events and the information about the initial value for the predecessor of the first event in the module, i.e. $M^c = \{\{ip\}, E\}$ where ip is the initial predecessor event name and $E = \{e_1, e_2, \dots, e_m\}$ is the set of events.

The sequential CCS specification of a module can be described in general by a set of expressions in the following form, where some terms $a_j.M^{(j)}$ may be absent:

$$M^{(i)} = a_0.M + a_1.M' + a_2.M'' + \dots + a_n.M^{(n)} \quad 0 \leq i \leq n$$

The mapping from a module $M_i \in \mathbb{M}$ to its causality-based representation M_i^c is described in the following algorithm:

1. Assign a unique name^{*} to each term $M, M', \dots, M^{(n)}$:
e.g. $0 \rightarrow M, 1 \rightarrow M', 2 \rightarrow M'', \dots, i \rightarrow M^{(i)}, \dots$ and so on.
2. Transform the actions in the construct $a_i.M^{(i)}$ into events by using the names assigned to $M^{(i)}$ as event names or as predecessor names and add them to E (E is initially empty.):
e.g. $a_0 \rightarrow \{\{i\}, a_0, 0\} \in E; a_1 \rightarrow \{\{i\}, a_1, 1\} \in E; a_2 \rightarrow \{\{i\}, a_2, 2\} \in E;$ and so on.
3. The causality-based representation $M_i^c \in \mathbb{M}^c$ of module M_i can now be written as:
 $M_i^c = \{\{0\}, E\}.$

Application of this algorithm yields a finite representation of infinite behaviors. Furthermore, the branching information is preserved. An event $\{x\}a_x$ characterizes a self-loop back to the original state in a labeled transition graph (see module B of the example).

Phase (2): Concurrent composition and test suite derivation

Concurrent composition of a system S^c can be expressed by:

$$S^c = (\dots((M_1^c \mid_c M_2^c) \mid_c M_3^c) \mid_c \dots \mid_c M_n^c).$$

^{*} Note that the names must be globally unique in the entire specification of the system.

We assume that each module M_i^c in the specification S is strongly connected (see Section 1.2). Each module comprises a finite number of traces which must be executed at least once in a concurrent transition tour. The goal of the algorithm in this phase is to find a possible configuration which fulfills the requirements of a concurrent transition tour (see Definition (3)). This configuration can be interpreted as a test suite as described in Section 4.

The algorithm works incrementally by adding the traces of the next module to an intermediate configuration. The data structure to express a configuration is a time-event sequence diagram (for an example see Figure 1).

1. **Initialization:** Select one module $M \in \mathbb{M}^c$.
 - Create an event trace from the set of events $E \in M$ according to the causality constraints between the events and under the restriction that each event occurs in the trace at least once:
 $trace(M) = \{ip\} a_0 \cdot \{0\} b_1 \cdot \{1\} c_2$
 - This trace is the initial configuration $CONF = trace(M)$ of the algorithm.
2. **Loop:** For each module $M_i^c \in (\mathbb{M}^c \setminus M)$ apply the following steps:
 - Take the initial event e_{init} from E_i .
 - Combine the configuration $CONF$ with e_{init} according to the causality information provided with e_{init} .
 - Select the next causal event e from E_i . Apply the previous step to the new event until all events in the module have been processed at least once.
 - Continue with the remaining modules in $\mathbb{M}^c$ using the newly obtained configuration $CONF := CONF|_c M_i^c$ until all modules have been considered.

The configuration $CONF$ obtained after all the modules in the specification have been processed represents a concurrent interpretation of the entire system S . It maintains the same information as its counterpart in the interleaved-based framework. Since $CONF$ contains all allowed actions of every module in the system at least once, it can be used as a test suite in a distributed test architecture to test an implementation implemented according to the specification S (see Section 4).

3.3 Application of the algorithm to the example

In this section, the above strategy is applied to the example given in Section 3.1.

Phase (1)

The CCS specification of module B is defined by the following:

$$B = a.B'; \quad B' = u.B' + v.B''; \quad B'' = b.B.$$

Applying the algorithm in phase (1), module B can be represented by its causality-based notation: $B^c = \{\{0\}, \{\{0\} a_1, \{1\} u_1, \{1\} v_2, \{2\} b_0\}\}$.

Similarly, module A is represented by: $A^c = \{\{3\}, \{\{3\} a_4, \{4\} x_5, \{5\} y_6, \{6\} b_3\}\}$.

Phase (2)

The system S is now interpreted as a system containing two concurrent modules $S^c = A^c \parallel B^c$. An initial configuration can be obtained by deriving an action sequence from module A^c . The shortest trace satisfying the requirement that each action occurs at least once is:

$$CONF_i = trace(A^c) = \{3\} a_4 \cdot \{4\} x_5 \cdot \{5\} y_6 \cdot \{6\} b_3.$$

Module B is described by the event set $E_B = \{\{0\} a_1, \{1\} u_1, \{1\} v_2, \{2\} b_0\}$ and the initial event $\{0\} a_1$. To derive the new configuration, the existing configuration is enriched with the events from module B step-by-step according to their causality relations in module B .

We start with the initial event in B by synchronizing it with the first event in $CONF_i$. Applying the causality-based semantics rules [Coe92a], we obtain an intermediate configuration and event set:

$$CONF_{inter} = \{0, 3\} a_7 \cdot \{7\} x_5 \cdot \{5\} y_6 \cdot \{6\} b_3 \quad \text{and} \quad E_{B_{inter}} = \{\{7\} u_7, \{7\} v_2, \{2\} b_0\}$$

Now, the events u and v can be applied because they both follow event a . To obtain a configuration which includes all events at least once, we choose event u before v because u represents a self loop back to the same state. In the new intermediate configuration $CONF_{inter}$, the partial trace $\{7\} u_7 \cdot \{7\} v_2$ is now concurrent to the partial trace $\{7\} x_5 \cdot \{5\} y_6 \cdot \{6\} b_3$.

Finally, the event $\{6\} b_3$ of the intermediate configuration synchronizes with $\{2\} b_0$ of module B , and all events in B occur in the new configuration $CONF$ whose set of events can be given as:

$$E_{CONF} = \{\{8\} a_7, \{7\} x_5, \{5\} y_6, \{2, 6\} b_8, \{7\} u_7, \{7\} v_2\}.$$

The representation of $CONF$ in a time-event diagram is given in Figure 3a. The numbers along the arrows represent the causal relationships between the events. In general, there is more than one way to derive a configuration. Figure 3b shows another configuration with multiple occurrences of event u .

The restriction of using every event only once in a concurrent transition tour, if possible, gives the smallest configuration with a minimal number of events. The configuration shown in Figure 3a corresponds to the smallest configuration satisfying the definition of a concurrent transition tour and can be used as a test suite.

3.4 Assessment of the proposed approach

The approach avoids state space explosion caused by interleaving of actions by introducing a true concurrency semantics. The semantics reflects the structural decomposition of a system into concurrent modules and is much easier to understand than the interleaved-based semantics which is based on abstract states.

To show the effectiveness of the new approach by way of comparison, one can construct the interleaved composite LTS of the example $S = A \parallel B$. It contains 7 states and 12 transitions. In that LTS, the concurrency between the actions x and y of module A and u and v of module B cannot be expressed. A transition tour through this composite LTS has to go through all 7 states and comprises at least 18 transitions, whereas the concurrent transition tour consists of 6 events.

Figure 3 Time-event sequence diagrams of two configurations of the example in Figure 2.

A test run with a transition tour obtained from the composite LTS would go through the single modules A and B multiple times. Furthermore, concurrent actions on different modules are serialized which requires a centralized control mechanism. In contrast, the concurrent transition tour generated by our proposed method terminates when every transition in each module is visited once, and concurrent actions are allowed to be executed concurrently. The price to pay for the simplification in our approach is that the derived test suites possess a lower test coverage than FSM-based test derivation methods because it tests only transitions but not states.

The proposed algorithm to construct a concurrent transition tour is very simple and may fail in some cases. A crucial point is to find a suitable initial configuration. To solve this problem, we are currently working on a new basic notation based on a set of sub-traces of events describing a module rather than a simple set of events used in this paper.

4 DISTRIBUTED TEST ARCHITECTURE AND ITS APPLICATION

4.1 Presentation of the distributed test architecture

Testing a distributed system requires a distributed test architecture. The term *distributed* means that the tester consists of two or more parts which are physically separated. The distributed tester can be seen as another distributed system running in parallel with the IUT. Therefore, it can be expressed using the same distributed model as the IUT itself.

Each tester part observes and/or controls a subset of interaction points at the IUT. It is assumed that all inter-unit interactions can be observed by the tester via points of observation (POs), and that the interactions between the units and the environment of the system can be controlled and observed via points of control and observation (PCOs).

Note that no communication between the tester parts during the execution of a test suite is needed; a tester part stores the information gathered during the test execution. After the test run, the information is collected and used in an assessment process to compute the test verdict. Depending on practical needs, the tester might react on some events which must be evaluated before the test run continues. In this case, communication between tester parts may be necessary. However, it is not a prerequisite in our approach.

Figure 4 An example of a distributed test architecture

In the example test configuration of Figure 5, the IUT consists of two units IUT_A and IUT_B which are the implemented modules A and B respectively. The tester consists of two parts T_1 and T_2 . The derived concurrent transition tour of our example comprises the internal actions a and b which are observed via POs in tester part T_1 . The other actions are exchanged between the IUT and the tester, and can be observed and controlled via the PCOs.

The prerequisite to determine the correct behavior of the IUT is the test of the total order of actions of the IUT (see Definition (2) of conformance). However, each tester part is only able to observe a partial order of actions which is not a sufficient condition for conformance [Dss90].

A way to construct a correct total order of actions from the set of partial orders is feasible by using logical timers [Mat89] [Fid91]. In this concept, each tester part and each unit of the IUT maintains a logical clock (a simple counter) and assigns the current value of the logical clock to actions sent out or received. After a communication between tester parts and/or IUT units, the value of the clock is increased. The concrete realization of logical timers is not considered in this paper.

The total order of events is fixed in the concurrent transition tour. A conforming test run for an IUT implemented according to its multi-module specification must result in an action sequence which satisfies the causality relations among the actions of the test suite. Due to concurrency, the actual order of concurrent actions obtained in a test run is nondeterministic. However, any order of actions is correct as long as the causal order of actions holds.

4.2 Application of the distributed test architecture to the example

A possible test architecture to test the correctness of an implementation based on the example in Section 3.1 consists of two independent tester parts T_1 and T_2 (see Figure 4). T_1 observes and/or

controls the actions a , x , y , and b , whereas T_2 observes and controls the actions u and v . The concurrent transition tour derived in the previous section has to be adapted for each tester part by projecting the concurrent transition tour to partial sequences of actions which are visible to the corresponding tester part. In our example, this means that T_1 observes the partial sequence $a \cdot x \cdot y \cdot b$, and T_2 the partial sequence $u \cdot v$.

The test assessment process after the test run must be able to combine all observed partial sequences together to obtain the total order of actions. As mentioned above, this can be solved by using logical timers. After combining the partial sequences according to the values of the logical clock readings which are piggybacked on the exchanged actions, the derived total order of actions can be assessed. For example, a test run may result in a total order of actions $a \cdot x \cdot u \cdot v \cdot y \cdot b$ or $a \cdot u \cdot v \cdot x \cdot y \cdot b$. In both cases, the implementation conforms to its specification, whereas a sequence $a \cdot x \cdot u \cdot y \cdot b \cdot v$ indicates an error in the IUT.

The definition of a concurrent transition tour allows a concise notation to address this kind of nondeterminism. Furthermore, it does not force the test execution to a single sequence of actions which is generally feasible only if the tester has full control over the next actions to be executed. That means, for example, that the tester does not need to control the step-by-step execution of the concurrent actions x and v as it will have to do if the test suite is derived from a composite module using interleaved-based semantics. Especially in the context of a distributed system, the requirement of full control is often difficult to achieve.

5 RELATED WORK

Test methods for protocols using the black-box model which support test suite generation from a single module deterministic finite state machine (FSM) have been widely studied [Aho88] [Bos91] [Cha89]. All methods adopt the conceptual test architecture as defined in ISO IS 9646/1. The FSM-based methods can be extended to test both the control and data flows of a protocol described in a single extended FSM (EFSM) [Cha93] [Cha94] [Wan93].

In [Klo92] [Luo94] and [Tri92], the test generation approach for deterministic FSMs was adapted to nondeterministic ones. Also the problem of test suite generation from communicating FSMs was recognized. The proposed solutions still use the approach of constructing a single FSM from the system of communicating FSMs and deriving the test suite from the composite FSM. Apart from computational problems, the technique neglects aspects of observability of the interaction points and the occurrence of interleaving events.

A very general approach to test suite generation from EFSMs is given in [Wan93] where interactions between modules are modelled by variables. Although dynamic creation of modules can be handled, the failure model must be supplied explicitly by an expert familiar with the specification and no approach on how to execute a test run of concurrent modules is presented.

One of the first papers on testing communicating FSMs is [LSK93]. This paper describes an *on-the-fly* testing method that assumes all transitions in a single module also occur in the composite behavior. A random walk method is applied to select the next possible input to the IUT which causes the problem that not under all cases, progress in executing new transitions can be assured. A general disadvantage of *on-the-fly* testing is that coverage of the test suite cannot be computed before the test run is executed since the test suite is created dynamically during testing. The second approach presented in [LSK93] is based on pruning the modules of a specification according to the test purposes before the modules are combined using conventional

interleaved-based semantics. This technique cannot guarantee that the pruned specification still maintains the same behavior as the original one.

The paper [LDB94] is dedicated to the problem of distributed PCOs for a single IUT. The restriction here again is that the IUT has to be specified as one FSM. The paper describes an algorithm to construct a test sequence which ensures that the IUT can be tested correctly via the distributed PCOs. It is shown that a suitable test sequence guaranteeing the same test coverage as a test sequence over local PCOs cannot be obtained in the general case. This problem is solved in the approach presented in our paper by introducing logical timers.

Reachability analysis is a topic that is strongly related to the problem of test suite generation. In this case, the goal consists in constructing a reachability graph (RG) from the specification in order to verify properties of the specification. To limit the efforts in constructing an RG and to be able to keep the state space under control, the context of a partial behavior of the system can be used [Che93]. This approach generates a contextual partial RG that is usually smaller than the same RG without using context constraints. Unfortunately, the application of this approach to protocol descriptions is not straightforward. The method is most applicable to protocols that use parallel composition of modules with many synchronizing actions common to a large number of modules. These properties may not always occur in a protocol specification.

The approach presented in this paper extends the test derivation methods suggested in the literature in three important aspects:

- a proposal to derive test suites to test a system consisting of more than one unit;
- avoids the state space explosion problem during the generation of the test suite; and
- shows how to apply the derived test suite in a distributed test architecture.

6 CONCLUSIONS

The paper proposes an approach for testing distributed systems comprising of multiple concurrent units. A scheme for test suite derivation based on a concurrent transition tour was presented which allows the notion of concurrent action sequences. In our model, a multi-module specification is explicated by a causality-based semantics to express concurrency between modules and thus, between actions. An important side-effect of this approach is the avoidance of state space explosion by avoiding an interleaved representation of actions. The approach was demonstrated using a simple example.

The interpretation of the derived concurrent transition tour in the context of a distributed test architecture providing the grey-box test method was given. It was shown that the traditional approach to derive a composite module using the interleaved-based semantics is not practical because a test suite derived from the composite machine requires the tester to have full control over the execution order of actions in all units of the IUT. This requirement is achievable only by introducing control mechanisms in the implementation [Kim94].

Further refinements of the algorithm will include a differentiation of actions between inputs and outputs and an analysis of the controllability of the derived test suite by a distributed tester. To explore the potential of the technique, we are applying the algorithm to a wide range of protocol specifications under different test contexts, including multi-party tests of lower level protocols and conformance tests of application protocols. Last but not the least, we are interested in extending the approach to global state testing and data flow testing [Cha94].

7 REFERENCES

- [Aho88] A.V. Aho, A.T. Dahbura, D. Lee, M. Ü. Uyar: *An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours*; 8th International Symposium on Protocol Specification, Testing and Verification; Atlantic City, USA; 1988.
- [Bos91] B. S. Bosik, M. Ü. Uyar: *Finite state machine based formal methods in protocol conformance testing: from theory to implementation*; Computer Networks and ISDN Systems 22 (1); 1991; pp. 7–33.
- [Cha89] W. Y. L. Chan, S. T. Voung, M. R. Ito: *On test sequence generation for protocols*; 9th International Symposium on Protocol Specification, Testing and Verification; 1989.
- [Cha93] S. T. Chanson, J. Zhu: *A Unified Approach to Protocol Test Sequence Generation*; IEEE INFOCOM; San Francisco, USA; 1993; pp. 106–114.
- [Cha94] S. T. Chanson, J. Zhu: *Automatic Protocol Test Suite Derivation*; IEEE INFOCOM; Toronto, Canada; 1994; pp. 792–799.
- [Che93] S. C. Cheung, J. Kramer: *Enhancing Compositional Reachability Analysis with Context Constraints*; 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering; Los Angeles, USA; 1993; pp. 115–125.
- [Coe92a] R. J. Coelho da Costa, J. P. Courtiat: *A Causality-Based Semantics for CCS*; 1st North American Process Algebra Workshop, Stony Brook, USA; 1992.
- [Coe92b] R. J. Coelho da Costa, J. P. Courtiat: *A True Concurrency Semantics for LOTOS*; 5th International Conference on Formal Description Techniques; Perros-Guirec, France; 1992.
- [Dss90] R. Dssouli, R. Fournier, G. v. Bochmann: *Distributed Observation and FIFO Queues*; 3th International Conference on Formal Description Techniques; Madrid, Spain; 1990.
- [Fid91] C. Fidge: *Logical Time in Distributed Computing Systems*; IEEE Computer, vol. 24, no. 8 (Aug. 1991); pp. 28–33.
- [Fid92] C. J. Fidge: *Process Algebra Traces Augmented with Causal Relationships*; 5th International Conference on Formal Description Techniques; Perros-Guirec, France; 1992.
- [Hol94] G. J. Holzmann: *An Improvement in Formal Verification*; 7th International Conference on Formal Description Techniques; Berne, Switzerland; 1994.
- [Kim93] M. C. Kim, S. T. Chanson, S. T. Vuong: *Concurrency model and its application to formal protocol specifications*; IEEE INFOCOM; San Francisco, USA; 1993; pp. 766–773.
- [Kim94] M. C. Kim, S. T. Chanson: *Design for Testability of Protocols based on Formal Specifications*; submitted for discussion in the working group P.54/Q.10 “Formal Methods in Conformance Testing”; 1994.
- [Klo92] H. Kloosterman: *Test derivation from non-deterministic finite state machines*; 5th International Workshop on Protocol Test Systems; Montreal, Canada; 1992.
- [Lan92] R. Langerak: *Bundle Event Structures: A Non-Interleaving Semantics for LOTOS*; 5th International Conference on Formal Description Techniques; Perros-Guirec, France; 1992.
- [Lan94] R. Langerak: *True Concurrency Models for LOTOS*; Tutorial script; 7th International Conference on Formal Description Techniques; Berne, Switzerland; 1994.

- [LDB94] G. Luo, R. Dssouli, G. v. Bochmann, P. Venkataram, A. Ghedamsi: *Test generation with respect to Distributed Interfaces*; Computer Standards & Interfaces, vol. 16, no. 2 (June 1994); pp. 119–132.
- [Luo94] G. Luo, G. v. Bochmann, A. Petrenko: *Test selection based on communicating nondeterministic finite state machines using a generalized Wp-method*; IEEE Transactions on Software Engineering, vol. 20, no. 2 (Feb. 1994); pp. 149–162.
- [LSK93] D. Lee, K. K. Sabnani, D. M. Kristol, S. Paul: *Conformance Testing of Protocols Specified as Communicating FSMs*; IEEE INFOCOM'93; San Francisco, USA; 1993; pp. 115–127.
- [Mat89] F. Mattern: *Virtual Time and Global States of Distributed Systems*; International Workshop on Parallel and Distributed Algorithms; Gers, France; 1988.
- [Mil89] R. Milner: *Communication and Concurrency*; Prentice Hall, New York; 1989.
- [Pro91] D. K. Probst, H. F. Li: *Using Partial-order Semantics to Avoid the State Explosion Problem in Asynchronous Systems*; 2nd International Conference on Computer-aided Verification; New Brunswick, USA; 1991.
- [Sid89] D. P. Sidhu, T. K. Leung: *Formal Methods for Protocol Testing: A Detailed Study*; IEEE Transactions on Software Engineering, vol. 15, no. 4 (Apr. 1989); pp. 413–426.
- [Tri92] P. Tripathy, K. Naik: *Generation of adaptive test cases from non-deterministic finite state models*; 5th International Workshop on Protocol Test Systems; Montreal, Canada; 1992.
- [Vuo93] S. T. Vuong, A. A. F. Loureiro, S. T. Chanson: *A Framework for the Design for Testability of Communication Protocols*; 6th International Workshop on Protocol Test Systems; Pau, France; 1993.
- [Wan93] C.-J. Wang, M. T. Liu: *Automatic Test Case Generation for Estelle*; International Conference on Network Protocols; San Francisco, USA; 1993.

8 BIOGRAPHY

Andreas Ulrich received the Diplom-Ingenieur in Computer Sciences from the Otto-von-Guericke University Magdeburg, Germany, in 1992. He is currently on leave at the Hong Kong University of Science and Technology to prepare for his Ph.D. thesis. The leave is partially supported by the German Academic Exchange Service (DAAD). His research interests include distributed systems, formal specification techniques in protocol engineering, and protocol testing.

Samuel T. Chanson received his M.Sc. and Ph.D. degrees in Electrical Engineering and Computer Sciences from the University of California, Berkeley, in 1971 and 1975 respectively. He was an Assistant Professor at Purdue University for two years. He then joined the Department of Computer Science at the University of British Columbia where he is a Professor and Director of the Distributed Systems Research Group. Since 1993 he has been on leave at the Department of Computer Science, Hong Kong University of Science and Technology. He has served on program committees of various international conferences on communication protocols and distributed systems. His research interests include communication protocols, high speed computer networks, and distributed operating systems.