7

ARCHITECTURES FOR TESTING DISTRIBUTED SYSTEMS

Andreas Ulrich[a], Hartmut König[b]

a *Siemens AG, Corporate Technology, ZT SE 1, 81739 München, Germany*
E-mail: andreas.ulrich@mchp.siemens.de

b *BTU Cottbus, Department of Computer Science, PF 101344, 03013 Cottbus, Germany*
E-mail: koenig@informatik.tu-cottbus.de

Abstract Stabilizing network infrastructures has moved the focus of software system engineering to the development of distributed applications running on top of the network. The complexity of distributed systems and their inherent concurrency pose high requirements on their design and implementation. This is also true for the validation of the systems, in particular the test. Compared to protocol testing the test of distributed systems and applications requires different methods for deriving test cases and for running the test. In this paper, we discuss architectures for testing distributed, concurrent systems. We suggest three different models: a global tester that has total control over the distributed system under test (SUT) and, more interestingly, two types of distributed testers comprising several concurrent tester components. The test architectures rely on a grey-box testing approach that allows to observe internal interactions of the SUT by the tester. In order to assure a correct assessment of the test data collected by the distributed tester components, the tester has to maintain a correct global view on the SUT. This can be achieved either by the use of redundant points of control and observation or by test coordination procedures. We outline the features of each approach and discuss their benefits and shortages. Finally, we describe possible simplifications for the architectures.

Keywords: Distributed systems, concurrency, test architectures, specification-based testing, concurrent transition tour.

1 INTRODUCTION

Designing, implementing, and validating complex distributed systems requires a deep understanding of the communication taking place and the order of communication messages exchanged between the individual compo-

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35567-2_25

G. Csopaki et al. (eds.), *Testing of Communicating Systems*

nents to avoid unwanted behavior during runtime. This applies in particular to the testing of these systems. Compared to protocol testing the test of distributed systems and applications requires different methods for deriving test cases and for running the tests. In [12] we introduced the concept of a concurrent transition tour an approach for providing test cases for distributed systems. The derivation of concurrent transition tours was represented in [13]. In this paper we continue the approach with the discussion of possible test architectures to run these tests.

We assume that a distributed system consists of a collection of components, each of them realizing a sequential flow of execution. Each component has an interface that defines its incoming and outgoing Messages. The test of distributed systems usually follows the steps of single component tests, integration tests of components, and finally the system test. Especially, the integration test is mainly aggravated by the following properties of distributed applications:

- true concurrency between components,
- absence of a global clock, and
- message exchanges between components that are unobservable by a tester.

We assume that a test architecture for distributed systems consists of the following basic elements:

- the system under test (SUT), i.e. the executable implementation of the distributed software system to be tested,
- the tester that implements a test case and also assesses the results of a test run,
- points of control and observation (PCOs) between tester and SUT, and
- possibly test coordination procedures which coordinate the actions between different tester components.

The paper defines requirements to the test architecture that need to be matched to support the execution of test runs and their correct assessment afterwards. These requirements address the SUT, which must be prepared in a suitable way to perform a test run. Requirements in this category are referred commonly to rules of "design for testability". Further requirements are suited to design testers that correctly assess the results of a test run. Note that our discussion is confined for the testing of functional aspects. We do not consider quality-of-service and performance aspects of distributed systems that are discussed in [15].

Two principal test architectures are presented: a global tester that observes and controls all distributed components of a SUT in a central manner, and a distributed tester that consists of a number of distributed, concurrent tester components, each of them collecting only partial information about the exe-

cution progress in components of the SUT. Especially, a distributed tester is of interest since it makes better use of system resources and supports concurrency within the SUT directly. Whereas the global tester can be a performance bottleneck during a test run.

Assuming a grey-box testing approach and the right choice of points of control and observation (PCOs) between tester and SUT, we show that also a distributed tester works correctly and brings up the expected result. The PCOs must be provided in the implementation of the SUT to allow the tester to observe necessary information for assessing a test run. A common means for this purpose is the insertion of software probes into the SUT.

The paper is organized as follows. In Section 2 we introduce some basic notions as well as an example which are used in the following discussion. Section 3 describes the proposed test architectures. Section 4 discusses possible simplifications for the application of these architectures. The final remarks summarize the results and give an outlook on needed further research.

2 PRELIMINARIES

2.1 BASIC NOTATIONS

We suppose a specification of a distributed system as a parallel composition $\Im = M_1 \parallel \ldots \parallel M_k$ of interacting components. Each component realizes a certain function of the distributed system, e.g. in the form of a client and/or server. It is described by a sequential automaton (*labeled transition system*, LTS). Components 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*.

Definition 1. A *labeled transition system* (LTS or *machine* for short) 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); $\rightarrow \subseteq S \times A \times S$ is a transition relation; and $s_0 \in S$ is the initial state.

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

Consider the simple system $\Im = A \parallel B$ whose LTSs are given in Figure 1. Under the assumption that actions a and c in each LTS synchronize, removal

of parallel operator || by applying interleaved-based semantics rules yields the composite machine $C_{\Im}$.

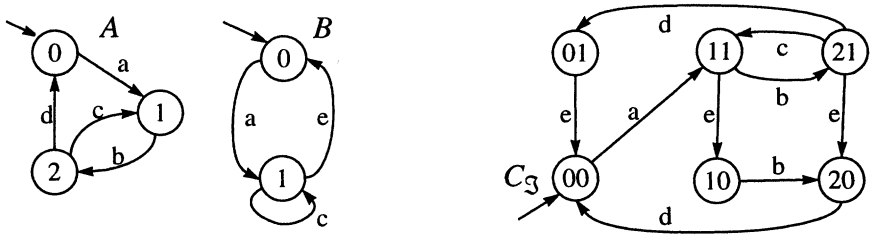

Figure 1. Example system $\Im = A \parallel B$ and its composite machine $C_{\Im}$.

2.2 TEST SUITES FOR CONCURRENT SYSTEMS

A conventional approach to test suite generation starts from the composite machine. The generation of a transition tour from the interleaving model of the composite machine 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 or not.

Possible test cases of our example derived from the composite machine in Figure 1 are the following sequences:

$\sigma_1 = a.b.c.b.d.e.a$, $\sigma_2 = a.b.c.b.e.d.a$, $\sigma_3 = a.b.c.e.b.d.a$.

All three sequences can be equally employed in a test run. They possess the same fault detection capability [7]. They differ only in different orders of the interleaving actions *b*, *d* and *e*. However, this order cannot be predicted in a sequential test run.

In [12], we extended the notion of a *transition tour* [8] to a *concurrent transition tour* (*CTT*) and applied it as a test suite for distributed systems. In the context of distributed systems, a transition tour is extended to a 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. It is depicted graphically as a time event sequence diagram where nodes are events and the directed arcs define the causality relation between events. A feasible construction algorithm of a CTT is presented in [13].

Definition 2. A *lposet* (*labeled partially ordered set*) is defined by the quadruple $(E, A, \leq, l)$, where E is a set of event names; A is a set of action names;

$\leq$ is a partial order expressing the causality information between events, i.e. $e \leq f$ if event e precedes event f in time; l: $E \rightarrow A$ is a labeling function assigning action names to events. Each labeled event represents an occurrence of the action labelling it, with the same action possibly having multiple occurrences. A *pomset* (*partially ordered multiset*) is an isomorphism class over event renaming of an lposet, denoted $[E, A, \leq, l]$.

Definition 3. (CTT) A *concurrent transition tour* of a concurrent system $\Im$ is the least pomset $CTT = [E_\Im, A_\Im, \leq, l]$ such that all actions of $\Im$ are covered at least once in the pomset, i.e. $a \in A_\Im$ for all actions in $\Im$ and $E_\Im$ is minimal.

Figure 2 depicts the CTT of the example system $\Im = A \parallel B$.

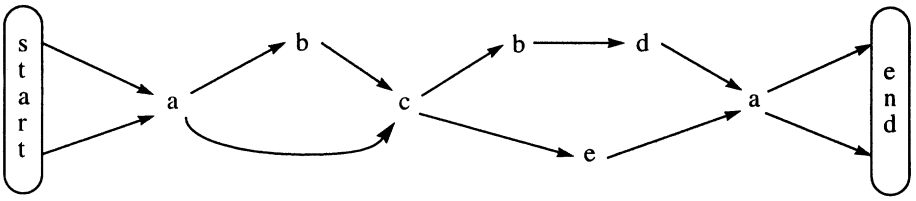

Figure 2. A CTT of system $\Im = A \parallel B$.

The CTT is used as a test case to test a distributed system. The CTT assumes that also the internal actions within the SUT are visible to the tester. Therefore, it supports the grey-box testing approach. To guarantee full coverage of all executable transitions of a distributed system, usually a set of CTTs is required that makes up the test suite for the distributed system.

2.3 ASSUMPTIONS ON A TEST ARCHITECTURE

A *test architecture* is defined as a parallel, synchronous composition of the *system under test* (*SUT*) of the distributed system $\Im$ and its tester. If a test suite comprises several test cases, a separate tester is built for each test case. It is required that the number of LTSs in the SUT is constant when the test run takes place (static system). Furthermore, the structure of the system $\Im$, i.e. its composition of n LTSs, is preserved in the implementation. From the testing perspective, it should be stated what actions can be observed and controlled. To avoid nondeterministic execution of the SUT due to nonobservability, we assume that all internal and external actions are observable during testing (grey-box testing approach).

In testing distributed systems the information about action names observed during a test run is not sufficient to assess conformance between specification and SUT. Due to the existence of multi-rendezvous among com-

ponents of the SUT, the tester must also know what components participate in a specific multi-rendezvous. Furthermore, the issue of nondeterminism within the SUT requires that the tester has the power not only to observe an action of the SUT, but must also control the occurrence of a multi-rendezvous. Thus, the crucial point in testing concurrent systems is to perform a *deterministic test run.*

The problem is addressed by applying *instant replay techniques* used for debugging concurrent systems [10]. The proposal in [10] assumes a global controller that is asked for permission by the components of the SUT before they are allowed to interact with each other. To achieve this, control code, called *probes*, is added into the source code of the components before each interaction invocation. Only after the tester has received all requests to execute a certain interaction from participating components, it grants this interaction. If not or if a wrong component asks for permission, an error in the SUT has occurred.

3 TEST ARCHITECTURES FOR TESTING DISTRIBUTED SYSTEMS

3.1 GLOBAL TESTERS

In this section, we describe possible test architectures that can be employed for the test of distributed systems. We first start with the simplest model, *the global tester*, that entirely simulates the environment of the STU as a sequential process during a test run. The global tester centrally collects information of the distributed SUT and derives the test verdict.

A global tester is implemented as a sequential machine T_G. The tester runs in parallel with the distributed system observing and controlling if necessary all external and internal actions of the SUT (grey-box test):

$T_G \parallel SUT$

After starting the test run the tester executes the events of the test sequence σ step by step. This leads to a rendezvous or multi-rendezvous communication between the tester and the SUT. The tester participates in the execution of a test event and records it together with the components of the SUT participating in this test event.

The global tester performs an action sequence derived from the composite machine of the distributed system as a test case. This action sequence contains a certain, assumed interleaving order of concurrent events. This assumed order must be validated to exist in the SUT during the test run. However since the interleaving sequence consists of concurrent test events with no

causal dependency between them, the action sequence assumed in the global tester is only observed by chance during a test run.

The global tester itself is modeled as an acyclic machine $T_G = (S_T, A_{CCT}, \rightarrow_T, S_{T0})$ with $|\sigma|+1$ states. The transitions in T_G are determined by the sequence of actions $\sigma = a_1.a_2. \ldots .a_n$ used as the test case:

$$S_{T0} -a_1\rightarrow S_{T1}, S_{T1} -a_2\rightarrow S_{T2}, \ldots, S_{Tn-1} -a_n\rightarrow S_{Tn}$$

For assigning the test verdict, a verdict label is attached to each state: *pass* to the final state and *fail* to all other states. If the tester reaches the final state after correctly executing σ, the test verdict *pass* is assigned to the test run. In all other cases the test run results in a *fail* verdict. The tester may not reach the final state if a desired component of the SUT is not able to participate in a certain test event; the test architecture including the tester and the SUT deadlocks in this case.

Figure 3 shows the test architecture with a global tester to test the system $SUT_{\mathfrak{Z}} = A \parallel B$. Note again that the tester has access to the internal actions of the SUT. The model of a global tester implementing $\sigma_1 = a.b.c.b.d.e.a$ as a test case is given in Figure 4.

The advantage of the global tester approach is its simplicity. Due to its global view on the distributed system under test, it can register the processed test events immediately as a unique sequence which preserves the correct causal dependencies between the actions of the SUT. A severe drawback of the global tester is however that it requires strict control over the execution of concurrent test events. This control might heavily intervene in the original behavior of the SUT. The question is whether there are other options to realize a test architecture that takes attention to the concurrency of actions in a SUT.

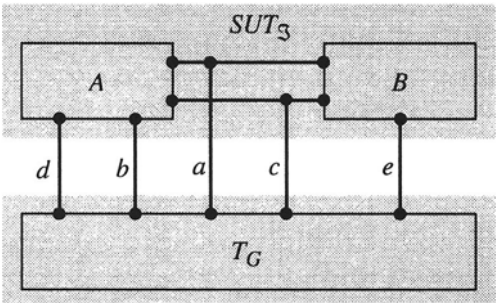

Figure 3. The test architecture with a global tester T_G for system $SUT_{\mathfrak{Z}} = A \parallel B$.

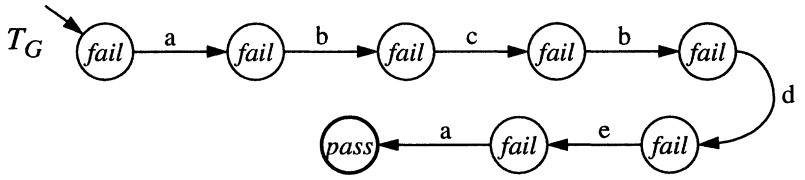

Figure 4. A global tester to test the system $SUT_3 = A \parallel B$.

3.2 DISTRIBUTED TESTERS

3.2.1 General Characteristics

A distributed tester is characterized by the following properties:

- It consists of several concurrently operating tester components which process together, but independently a *global test case* (TC). The tester can be described in the same way as the SUT, e. g. as a set of communicating machines.
- Each tester component executes a *partial test case* (PTC). A PTC is projection of the global test case TC which comprises only those events which can observed at the particular tester component. The causal dependencies between the test events of a PTC are determined by the global TC.
- Each tester component observes a subset of the set of all PCOs. Their selection and assignment to a tester component is a decision taken by the designer of the test architecture.
- The behavior of the tester components is controlled by a *Test Coordination Procedure* (TCP).

The general issue in distributed tester design is that the tester may assign a successful verdict to a test run although the SUT contains faults. This is possible because there is no global view on the SUT if a distributed tester is used. This problem can be tackled by a TCP between the distributed tester components, either by introducing synchronization events into the partial test cases of tester components or by the use of redundant PCOs to observe internal events of the SUT simultaneously by several tester components. For example, if a component of the SUT sends a message to another component, one tester component observes the send event of this communication, whereas another tester component observes the resulting receive event.

We can describe a distributed tester T_D by means of a set of concurrent tester components

$T_D = T_{D1} \parallel T_{D2} \parallel \ldots \parallel T_{Dn}$

which are controlled by a TCP. Each tester component processes a sequential partial test case in an acyclic machine that might be augmented with synchronization events. We further assume the same markings of *fail* and *pass* to assign a test verdict as discussed for the global tester.

In a test run, the distributed system ($T_{D1} \parallel T_{D2} \parallel \ldots \parallel T_{Dn}$) $\parallel$ *SUT* is executed. The tester components and the SUT participate in the respective test events and transfer from one local state to the next. If a test event or synchronization event of a TCP cannot be executed, the whole test architecture deadlocks. Only if all tester components reach their final state, the verdict *pass* is assigned to the test run.

3.2.2 Distributed Testers with Synchronization Events

A common distributed test architecture that is often used in testing distributed telecommunication systems uses synchronization events to implement a TCP. We discuss this type of distributed testers first.

We develop a distributed tester ${}^1T_D$ for the example system in Figure 1 and assume two tester components: ${}^1T_{D1}$ observes the actions *b* and *d* of the SUT, while ${}^1T_{D2}$ tests the actions *a*, *c* and *e*.

${}^1T_D = {}^1T_{D1} \parallel {}^1T_{D2}$

Additionally, the tester components exchange the synchronization event *sync*. Figure 5 shows this test architecture.

The derivation of test cases for the tester components is done as a projection of test events observable by the particular tester component from the global test case.

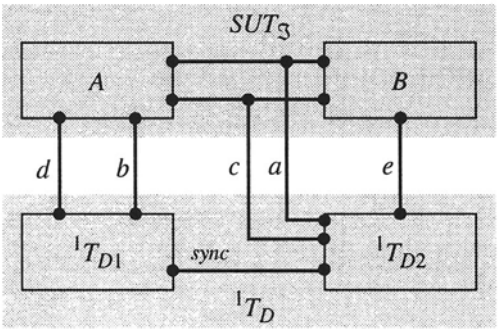

Figure 5. Test architecture with distributed tester ${}^1T_D$ and synchronization event *sync*.

Let us assume the CTT in Figure 2 as the global test case for system S. The projected and linearized partial test cases for the two tester components are the following:

${}^1\sigma_1 = b.b.d$ for tester component ${}^1T_{D1}$ and
${}^1\sigma_2 = a.c.e.a.$ for tester component ${}^1T_{D2}$.

These sequences must be supplemented with synchronization events. Synchronization events are included when the control goes over from one tester component to the other.

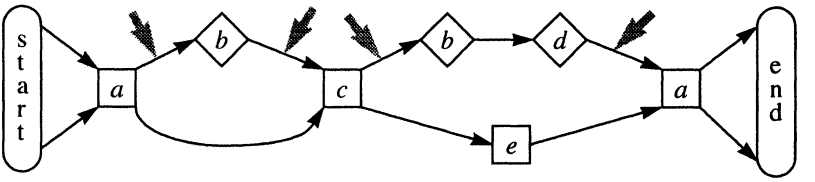

Figure 6. The CTT of the example system with arrows denoting necessary synchronization events within the partial test sequences.

We consider Figure 6. It depicts the global CTT of our example system. The actions of ${}^1T_{D1}$ are represented by squares , those of ${}^1T_{D2}$ by diamonds . The grey arrows mark the change from one tester component to the other one. At these points a synchronization event must be included to guarantee the correct global view on the SUT. Consequently, the tester components run the following partial test cases:

${}^1\sigma_1' = sync.b.sync.sync.b.d.sync$
${}^1\sigma_2' = a.sync.sync.c.sync.e.sync.a$

Note that the synchronization events are indispensable for assuring the correct global view on the SUT. If, for instance, the first appearance of action b in component A of the SUT follows action c, a distributed tester without a test coordination procedure would not detect this fault.

Assuming that a tester component is assigned to each component of the SUT, the distributed tester can fully exploit the concurrency among test events. For example, the total order of the interleaving actions b, d and e is not important for assessing a test run. Any test run is correct if only the causal dependency "b before d" is assured. Action e can interleave at any time instant within the constraints of the *sync* events.

Thus, the distributed tester does not need to control the execution of the test case entirely. Each tester component can run independently its partial test case. Only coordination between the tester components is necessary to guarantee the correct global view on the SUT.

3.2.3 Distributed Testers with Redundant Observation of Internal Events

The insertion of synchronization events is not the only possibility to realize coordination among tester components. Another and a more elegant option is the redundant observation of internal actions of the SUT by several tester components. This type of distributed test architecture is discussed again using the example from Section 2. It consists of two tester components

$^2T_D = {}^2T_{D1} \parallel {}^2T_{D2}$.

Each tester component controls and observes all interactions of a single component of the SUT. Thus, the tester represents an inverted image of the SUT. Figure 7 shows the test architecture of tester ^2T_D.

Partial test cases for the tester components are derived in the same manner as already described for the first type of distributed testers in Section 3.2.2 by projecting and linearizing the global test case of a CTT. We also assume the same marking of states in the tester components with *fail* and *pass* as discussed for the global tester. Using the CTT in Figure 2 as the basis for the derivation of partial test cases, the tester components $^2T_{D1}$ and $^2T_{D2}$ need to implement the following partial test cases:

$^2\sigma_1$ = *a.b.c.b.d.a* for tester component $^2T_{D1}$ and
$^2\sigma_2$ = *a.c.e.a* for tester component $^2T_{D2}$.

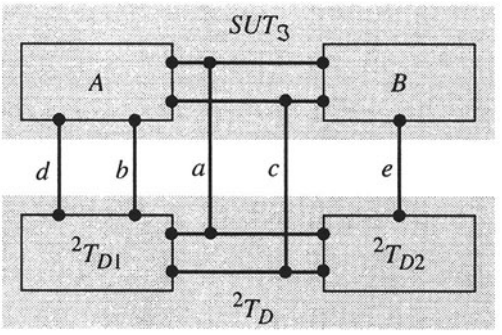

Figure 7. Test architecture with the distributed tester ^2T_D and redundant observation of test events.

Since both tester components observe the internal actions of the SUT, they are able to coordinate each other to maintain the correct global view on the system. In case of message passing, a practical solution to implement the redundant observation of internal communication is the observation of the send event of a message by one tester component and the observation of the

related receive event by another one. The causal dependency between the send and the receive event for the same communication action (message passing) can then be reconstructed.

In a test run the distributed system ($T_{D1} \parallel T_{D2} \parallel \ldots \parallel T_{Dn}$) $\parallel$ *SUT* is executed. The tester components and the SUT participate in the respective test events. A multi-rendezvous between the SUT components and the tester components takes place when internal actions of the SUT are executed as test events. In this case, the participating tester components synchronize to preserve the global view on the SUT. If a test event cannot be executed, the tester components associated to this test event deadlock. Only if all tester components reach their final state, the verdict *pass* is assigned to the test run.

Note also here that concurrency within the SUT is supported to a large extend if a tester component is provided for each component of the SUT. The advantage of this second distributed test architecture is that no additional synchronization events are needed to coordinate the tester components.

4 SIMPLIFICATIONS OF THE PROPOSED TEST ARCHITECTURES

The test architectures presented above contain several restrictive elements. For certain applications, the following test assumptions may be too restrictive:

- the control and observation of all actions of the SUT including the internal actions, and
- inclusion of additional synchronization events.

In the following discussion we shortly sketch some conditions which permit simplifications of the test architecture. More details are given in [14].

4.1 OBSERVATION OF INTERNAL COMMUNICATION

The complete observation of the internal communication (grey-box test) was introduced to force a deterministic test run. By observing the internal communication of the SUT by means of a *controller* according to [10] the SUT is forced to follow a certain execution order defined by the CTT. To (partially) omit the internal observation, it must be proved whether the behavior of the SUT does not lead to a nondeterministic behavior which cannot be detected by the tester. This requires that we need to take into consideration the following types of nondeterminism*:

- nondeterminism within a component, i.e., it exists the possibility of two or more outputs in the same local state, and
- race conditions, i.e., it exists a global state with at least two inputs to the same component enabled.

If the analysis of the specification of the SUT shows that these types of nondeterminism do not appear, additional means to force a deterministic test run can be omitted. This approach requires however that the components of the SUT are correctly implemented. To assure this a step-wise test method as proposed in [4] can be applied.

4.2 RENUNCIATION OF TEST COORDINATION PROCEDURES

In cases in which the number of additional synchronization events that are needed to implement the distributed tester of Section 3.2.2 is very high or in cases in which the redundant observation of internal actions of the SUT as required for the distributed tester of Section 3.2.3 is not possible, it might be desired to renounce the test coordination procedures. To synchronize the then asynchronously acting tester components, the following possibilities exist:

Simulation of a global clock by means of a synchronization protocol
There exist several such protocols in literature. A well-known one is the *Network Time Protocol* (NTP) [5]. It supports a synchronization of a few milliseconds divergence for local area networks and some 10 milliseconds for wide area networks. This precision is scarcely acceptable for many practical applications as the following measurements of the transmission time of RPC messages show. RPC (*Remote Procedure Call*) is a network service for the transparent execution of procedures on remote machines. It is the basic service used by the middleware platform CORBA. The transmission time of an about 8 kilobyte RPC message over an Ethernet network is 7,2 msec. For a 100 Byte message, the time is 0,29 msec only. In an ATM network with a 155 Mbps transfer rate, these times shorten to 0,5 and 0,02 msec, respectively. These measurements indicate that within the precision limits of NTP (1 msec in the best case), several messages can be sent over an Ethernet such that the causal dependencies between messages cannot be correctly established in a test run. For ATM and other high speed networks, the situation is even worse.

---

* Another kind of nondeterminism is interleaving, since the selection among two concurrent actions is arbitrary and unpredictable. We do not consider interleaving as nondeterminism here, because we allow concurrent actions in a test run to be carried out if a distributed tester is used.

Use of logical clocks in the test architecture
Logical clocks [2] are a very reliable and robust means to synchronize actions in a distributed system. It requires, however, that logical clocks are available in all components of the distributed system, i.e. also in the SUT. That means, the software of the SUT must be augmented such that logical clock values are piggybacked for each message exchanged between SUT and tester and between internal SUT components.

Emulation of a synchronous execution by means of a synchronizer
In our current discussion we assumed an asynchronous execution between the components of the SUT. A synchronized execution can be forced by introducing a *synchronizer* that determines the time for the execution of a communication event [1]. However, a synchronous execution of components in a distributed system cannot be assumed for most implementations. Moreover, the introduction of a synchronizer for the purpose of testing only would change the implementation enourmously. An approach that works similar to a synchronizer is discussed in [3].

Generation of synchronizable test suites
Synchronizable test suites which can be separately executed on different tester components without synchronization events are an attractive solution for this problem. Existing approaches [6], [11] run short because they can only be applied if a global composite machine for the distributed SUT is available. Approaches that support communicating machines as the description model for the SUT do not exist up to now.

5 FINAL REMARKS

In this paper we have suggested different architectures for the test of distributed systems. We have discussed rules for the positioning of PCOs as well as means to ensure a deterministic test run. Two principle test architectures have been considered: a global and a distributed tester. The global tester is a sequential component that observes all actions of the SUT at its PCOs, whereas the distributed tester consists of several concurrent tester components that observe partial behavior of the SUT only. These tester components must synchronize each other by means of a test coordination procedure to assure a correct and consistent global view on the SUT.

The presented approaches base on a synchronous communication paradigm between the components of the SUT and the test architecture. The assumption of synchronous communication is required for two reasons. First it restricts the state space of the distributed systems to a finite set, and sec-

ondly it allows to preserve the correct causal dependencies between the test events. Using an asynchronous communication paradigm, these properties would not hold anymore. The state space would be infinite and the causal dependencies between test events could not be correctly assessed. Nevertheless, it is necessary also to pursue research within this direction, because asynchronous communication is common in practice.

References

[1] C. T. Chou, I. Cidon, I. Gopal, S. Zaks: *Synchronizing asynchronous bounded-delay networks*; IEEE Transactions on Communications, vol. 38, no. 2 (Feb. 1990); pp. 144–147.

[2] C. J. Fidge: Logical Time in Distributed Computing Systems; IEEE Computer, vol. 24, no. 8 (August 1991); pp. 28-33.

[3] C. Jard, T. Jeron, H. Kahlouche, C. Viho: *Towards Automatic Distribution of Testers for Distributed Conformance Testing*; IFIP Joint Int'l Conference on Formal Description Techniques, and Protocol Specification, Testing, and Verification (FORTE/PSTV'98); Paris, France; 1998.

[4] H. König, A. Ulrich, M. Heiner: *Design for Testablility: A Step-wise Approach to Protocol Testing*; Proceedings of the IFIP 10th Int'l Workshop on Testing of Communicating Systems (IWTCS'97), Seoul, Korea; 1997; pp. 125-140.

[5] D. L. Mills: *Precision Synchronization of Computer Network Clocks*; ACM Computer Communication Review, vol. 24, no.2 (April 1994; pp. 28-42.

[6] 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.

[7] A. Petrenko, A. Ulrich, V. Chapenko: *Using partial-orders for detecting faults in concurrent systems*; Proceedings of the IFIP 11th Int'l Workshop on Testing of Communicating Systems (IWTCS'98), Russia, 1998.

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

[9] K. C. Tai, R. H. Carver: *Testing of distributed programs*; in A. Zomaya (ed.): *Handbook of Parallel and Distributed Computing*; McGraw Hill; 1995; pp. 956-979.

[10] K. C. Tai, R. H. Carver, E. E. Obaid: *Debugging concurrent Ada programs by deterministic execution*; IEEE Trans. on Software Eng., vol. 17, no. 1 (Jan. 1991); pp. 45-63.

[11] K. C. Tai, Y. C. Young: *Port-synchronizable test sequences for communication protocols*; 8th Int'l Workshop on Protocol Test Systems

(IWPTS'95); Paris, France; 1995; pp. 379–394.

[12] A. Ulrich, S. T. Chanson: *An approach to testing distributed software systems*; 15th PSTV 1995; Warsaw, Poland; pp. 107-122; 1995.

[13] A. Ulrich, H. König: *Specification-based testing of concurrent systems*; IFIP Joint Int'l Conference on Formal Description Techniques, and Protocol Specification, Testing, and Verification (FORTE/PSTV'97); Osaka, Japan; Nov. 18-21, 1997.

[14] A. Ulrich: *Testfallableitung und Testrealisierung in verteilten Systmen*; Dissertation (in German), Magdeburg University; Shaker Verlag, 1998.

[15] T. Walter, I. Schieferdecker, J. Grabowski: *Test architectures for distributed systems: state of the art and beyond*; 11th Int'l Workshop on Testing of Communicating Systems (IWTCS'98), Russia, 1998.