Test Architectures for Testing Distributed Systems

Andreas W. Ulrich, Peter Zimmerer, Gunther Chrobok-Diening
Siemens AG, ZT SE 1, 81739 München, Germany
{andreas.ulrich, peter.zimmerer, gunther.chrobok-diening}@mchp.siemens.de

Abstract
This paper suggests two test architectures for testing distributed, concurrent systems: a global tester that has total control over the distributed system under test (SUT) and, more interestingly, a distributed tester 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 realize distributed testers, the correct global view on the behavior of the SUT must be maintained by the tester. This can be achieved if coordination procedures are established between the concurrent tester components. Coordination between tester components can be implemented by means of redundant observation of internal interactions within the SUT or by means of using synchronization events between tester components. First concepts of a distributed tester are realized in a prototype of a test and monitoring tool used for inhouse testing, monitoring, and analyzing distributed systems within Siemens. The project is conducted in Siemens' R&D Software and Engineering Department in charge of its Information and Communication Networks Division.

Keywords
Test architectures, distributed systems, concurrency, specification-based testing, monitoring.

1 Introduction

Distributed software systems have gained increasing importance in recent years. Such applications are not just client/server systems, but more and more complex systems consisting of a number of independent components running concurrently on different machines on top of a communication network. Today distributed systems are implemented using middleware technologies, such as Microsoft/COM, OMG/CORBA, or Java RMI. But there exists also a number of proprietary solutions to distributed systems, e.g. implementations of a H.323 video conferencing protocol and others.

Designing, implementing, and validating such complex distributed systems requires a deep understanding of the communication taking place and the order of communication messages

exchanged between the individual components to avoid deadlocks or livelocks during runtime. This applies in particular to the testing of these systems.

We assume in our discussion 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. Testing distributed systems usually follows the steps of single component tests, integration tests of components, and finally the system test. Especially, the integration tests is mainly aggravated by the following properties of distributed applications:

- true concurrency between components,
- race conditions and other types of non-determinism
- 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) and pure observation points (POs) between tester and SUT, and
- possibly test coordination procedures which coordinate the activities 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 which are also relevant for distributed systems as discussed in [8].

As a result, 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 execution progress in components of the SUT. Especially, a distributed tester is of interest since it makes better use of system resources than the global tester that can be a performance bottleneck during a test run.

Assuming a grey-box testing approach and the right choice of PCOs and POs between tester and SUT, we show that also a distributed tester works correctly and brings up the expected result. The PCOs and POs 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 considerations to the design of test architectures helped implement a prototype of a *Test and Monitoring Tool* (TMT) for testing distributed systems. This tool makes the concurrent flow of messages in a SUT visible and allows to compare the results of two test runs in order to detect differences in the execution order of messages in the SUT. The tool supports an on-line mode that displays message exchange in the SUT at run time and an off-line mode for post-mortem analysis. TMT provides different views on the concurrent system that focus on the hierarchical structure between components of the SUT, on the temporal order of messages exchanged between components, or on a detailed view of the contents of transmitted messages. Moreover, a TMT user is able to control the execution of the SUT interactively.

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 presents the prototype of TMT that implements a distributed test architecture. Finally, Section 5 summarizes the presentation.

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 component synchronize, removal of parallel operator $\parallel$ by applying interleaved-based semantics rules yields the composite machine $C_\Im$ representing the global behavior of the distributed system $\Im$.

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

The composite machine $C_\Im$ describes all possible sequences of actions the distributed system is allowed to perform, e.g. the system $\Im$ may perform the following sequences:

$\alpha = a.b.c.e.b.d$, $\beta = a.e.b.d$, $\gamma = a.b.d.e.a.b.c$ and so on.

During testing the system we have to show that the implemented system (SUT) follows the behavior described in its specification (specification-based testing approach). This testing concept is more formally defined in the following Section.

2.2 The Testing Goal

The testing goal within this specification-based testing approach is to show conformance between the specification of the distributed system and its specification, i.e. the SUT of the implementation obeys the requirements given in the (functional) specification, it implements the specification. To show conformance a formal model of the specification must be provided and the same formal model must be assumed for the SUT. We assume the formal model of a system of communicating machines as presented in the previous Section is used as a specification model as well it is imposed on the behavior of the SUT.

Conformance between SUT and specification can now be established formally.

Definition 2. Given the specification of a distributed system $\Im$ as a composite machine $C_\Im$. An implementation $SUT_\Im$ of $\Im$ conforms to its specification iff $SUT_\Im$ exposes the same set of action sequences $\textbf{\textit{Seq}}(SUT_\Im)$ as are described in $C_\Im$, i.e. $\textbf{\textit{Seq}}(SUT_\Im) \supseteq \textbf{\textit{Seq}}(C_\Im)$.

This definition states that the implementation at least needs to implement all possible action sequences of the composite machine defining the global behavior of the distributed system $\Im$. But it may implement more functionality.

To test a SUT for Definition 2, an acceptance test is appropriate. This test defines test cases that cover all transitions of the composite machine at least once. Such test cases are referred to a transition tour through a machine. Note that the definition includes also the internal events of a distributed system in order to assure that the communication between the components of the SUT works. This test is essential in the integration test of a distributed system, and a test architecture supporting this grey-box testing approach is required.

A composite machine is however not the best model to derive acceptance tests since it is usually too huge to be computed due to the state explosion phenomenon caused by interleaving. In [1] it was shown that a composite machine is not needed as a basis model for an accep-

tance test even if the same degree of test (fault) coverage should be maintained. Instead it is sufficient to derive acceptance tests from a reduced partial-order based model, called *Mazurkiewicz Trace Machine* (MTM), a model commonly used in partial-order verification techniques.

Possible test cases of our example 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 [1]. They differ only in different orders of the interleaving actions *b*, *d* and *e*.

2.3 Challenges in Testing Distributed Systems

There are many challenges in testing distributed systems. One problem is the generation of appropriate test cases to test distributed systems sufficiently with a certain degree of test or fault coverage. Test case generation is in particular aggravated by the state explosion problem that occurs when the global system behavior is being analyzed. However, this problem in not discussed further in this paper.

Another challenge of testing distributed systems one is faced in the test execution phase is the intrinsic non-determinism of a distributed system under test. There exist three types of non-determinism in distributed systems:

1. non-determinism due to interleaving of concurrent actions of the SUT,
2. non-determinism within a component of the SUT, and
3. non-determinism due to race conditions within the SUT.

The first type of non-determinism refers to the fact that concurrent actions of a distributed system are independent from each other. Consequently, their occurrence within a global time of the system cannot be predicted. The action *b*, *d* and *e* from the example system in Figure 1 are such concurrent actions.

The second type refers to the fact that a tester may be not able to observe all actions going on within a component of a distributed system. Due to this limited observation the behavior of the component appears to be non-deterministic to the tester.

Finally, the third type of non-determinism occurs in the global behavior of the SUT. A typical example of a race condition is the receipt of two concurrent messages by a component of the distributed system. The further behavior of the system depends now on the order, in which the concurrent messages are received (e.g. concurrent write access to a shared variable, followed by a read of this variable).

The complete observation of the internal communication (grey-box test method) is required to observed the correct execution of internal actions within the SUT, but also to enforce a deterministic test run. By observing the internal communication of the SUT by means of a *controller* according to [4], the SUT is forced to follow a certain execution order defined by a test case that is implemented in the tester. To (partially) omit the control of internal observation, it must be proved whether this does not lead to a non-deterministic behavior of the SUT.

The first type of non-determinism can be avoided if the tester itself is implemented as a distributed system. See Section 3.2 for this discussion. Furthermore, if the analysis of the system specification shows that the 2nd and 3rd type of non-determinism cannot appear, then instant replay techniques to assure a deterministic behavior can be safely omitted and the tester only needs to observe the internal actions of the SUT via POs.

3 Test Architectures for Testing Distributed Systems

3.1 The Global Tester

In this Section, we describe possible test architectures which can be employed for the test of distributed systems. We first start with the simplest model, the *global tester*, which entirely simulates the environment of the STU during test run. The global tester centrally collects information of the distributed SUT, and derives the test verdict. Thereafter we extend the simple model such that the tester itself represents a distributed system.

A global tester is always 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 interleaving 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.

For example, possible interleaving action sequences of our system $\Im = A \parallel B$ of Section 2 that can be used to describe a global tester are the following three sequences:

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

In these sequences the actions b, d and e are interleaved meaning that they may occur in any of the three possible orders during a test run. Thus, in practice it may be possible that a global tester implements σ_1 as the test case, but observes only sequences of σ_2 or σ_3 in a test run. The test verdict *fail* might be assigned to such a test run, although the SUT obeys the specification.

To prevent this situation from happening, the global tester needs to control the execution of internal events of the SUT (PCOs also for internal actions of the SUT). This behavior can be achieved if *instant replay techniques* are applied to guarantee a *deterministic test run* [3]. The instant replay technique defines measures to include software probes in the SUT that allow the tester to control the subsequent behavior of the SUT.

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:

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

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 in cases that a desired component is not able to participate in a certain test event; the test architecture including the tester and the SUT deadlocks in this case.

Figure 2 shows the global tester to test the system $SUT_{\Im} = A \parallel B$. Note again that the tester has access to the internal actions of the SUT. The model of the global tester implementing σ_1 as the test case is given in Figure 3.

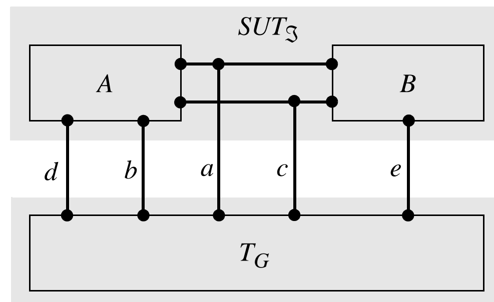

Figure 2. The global tester T_G for system $SUT_{\Im} = A \parallel B$.

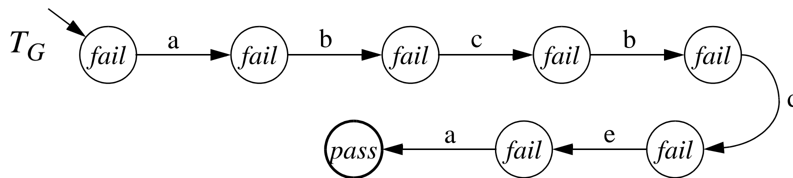

Figure 3. Global Tester of the system $SUT_{\Im} = A \parallel B$.

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 directly, with no further computations, 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 test events.

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 by the same model 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 PCOs and POs assigned to 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 and POs. PCOs exist for the external actions of the SUT, whereas POs are imposed on internal actions of the SUT. Their selection 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/POs 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

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

which are controlled by a TCP. Each tester component processes a sequential PTC via an acyclic machine that may be supplemented 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 $(T_{D1} \parallel T_{D2} \parallel \dots \parallel T_{Dn}) \parallel SUT$ 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. Additionally, the tester components exchange the synchronization event *sync*. Figure 4 shows this test architecture.

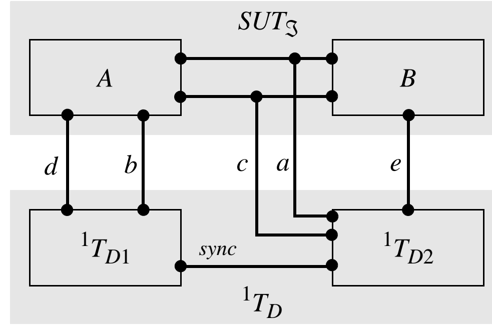

Figure 4. Test architecture of the distributed tester ${}^1T_D$ with synchronization event *sync*.

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. Let us assume the global test case $\sigma_1 = a.b.c.b.d.e.a$ is used. The projected partial test cases for the two tester components are the following:

${}^1\sigma_{11} = b.b.d$ for tester component ${}^1T_{D1}$ and
${}^1\sigma_{12} = a.c.e.a.$ for tester component ${}^1T_{D2}$.

These sequences must be supplemented with synchronization events. Synchronization events are always included when the control goes over from one tester component to the other. Consequently, the tester components run the following partial test cases:

${}^1\sigma_{11}' = sync.b.sync.sync.b.d.sync$
${}^1\sigma_{12}' = a.sync.sync.c.sync.e.sync.a$

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, 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. As a consequence, it is possible to replace the PCOs for the internal actions of the SUT to mere POs, thus controlling the behavior of the SUT only via the external actions of the SUT. Control over internal actions might be still required however in the case of the existence of race conditions or internal non-determinism. See Section 2.3 for this discussion.

3.2.3 Distributed Testers with Redundant Observation of Internal Actions

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 again discussed using the example from Section 2. It consists of two tester components

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

Each tester component controls and/or observes all interactions of a single component of the SUT. Thus, the tester represents an inverted image of the SUT. Figure 5 shows the test architecture of tester ^2T_D.

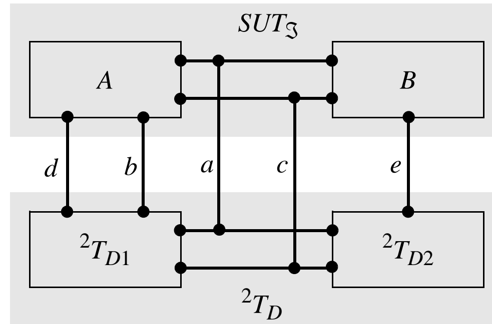

Figure 5. Test architecture of the distributed tester ^2T_D.

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. Using the global test case $\sigma_1 = a.b.c.b.d.e.a$ again 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:

$^2\sigma_{11} = a.b.c.b.d.a$ for tester component $^2T_{D1}$ and
$^2\sigma_{12} = a.c.e.a$ for tester component $^2T_{D2}$.

Since both tester components observe the internal actions of the SUT, they coordinate each other to maintain the correct global view on the system. A practical solution to implement the redundant observation of internal communication is that in case of message passing, a tester component observes the send event of the message, whereas another tester component observes the receive event of this message. The causal dependency between the send and the receive event for the same communication action (message passing) can be assumed. This is the way how an in-house prototype of a test and monitoring tool works (see Section 4).

Note that also here concurrency within the SUT is greatly supported if a tester component is provided for each component of the SUT. The advantage of the second distributed test architecture is that no additional synchronization events are needed to coordinate the tester components. The renunciation of PCOs for internal actions of the SUT is possible if no other types of non-determinism exist in the SUT. See Section 2.3 for this discussion.

3.3 Test Cases for Distributed Test Architectures

This Section finally emphasizes the need of an appropriate description of test cases for distributed systems tested by means of a distributed tester. As discussed above, test cases described as sequences of test events are not very suitable since they do not respect the concurrency that exists among interleaving events. Hence, a notation of test cases is required that respects concurrency of test events. Such a notation should use a partial-order approach to make concurrent test event explicit. A standard notation for this purpose that has been in use for several years to describe (concurrent) system requirements of distributed telecommunication systems is the *Message Sequence Chart* (MSC) representation [9].

Using a representation similar to a MSC, the global test case for the example system in Figure 1 can be represented as depicted in Figure 6.

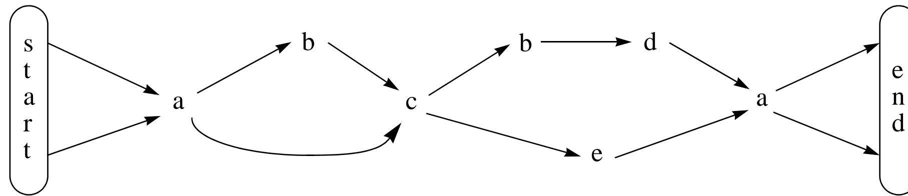

Figure 6. A concurrent global test case of system $\Im = A \parallel B$.

The concurrent global test case depicts the direct causal dependencies between test events. It does not assume a certain global order of all events, instead concurrent events are represented independently from each other. It can be easily seen that all three global action sequences σ_1, σ_2 and σ_3 that are contained in the composite machine of the example system in Figure 1 are still preserved in the concurrent test case [6]. In the diagram of Figure 6 the time axis is assumed to run from left to right.

4 A Tool for Testing and Monitoring Distributed Systems

4.1 Overview

In order to support clients within the Siemens Corporation, we are currently developing a *test and monitoring tool* (TMT) to support testing, monitoring and analyzing distributed applications. A first prototype of TMT was implemented in Java JDK 1.2 in more than 210 classes and 60,000 lines of code. The prototype can be applied to distributed systems implemented in C++ on UNIX or Windows NT platforms or to distributed systems in Java. The prototype was developed within the R&D Software and Engineering Department of Siemens as a contract work of its Information and Communication Networks Division. Figure 7 shows the main window of TMT prototype.

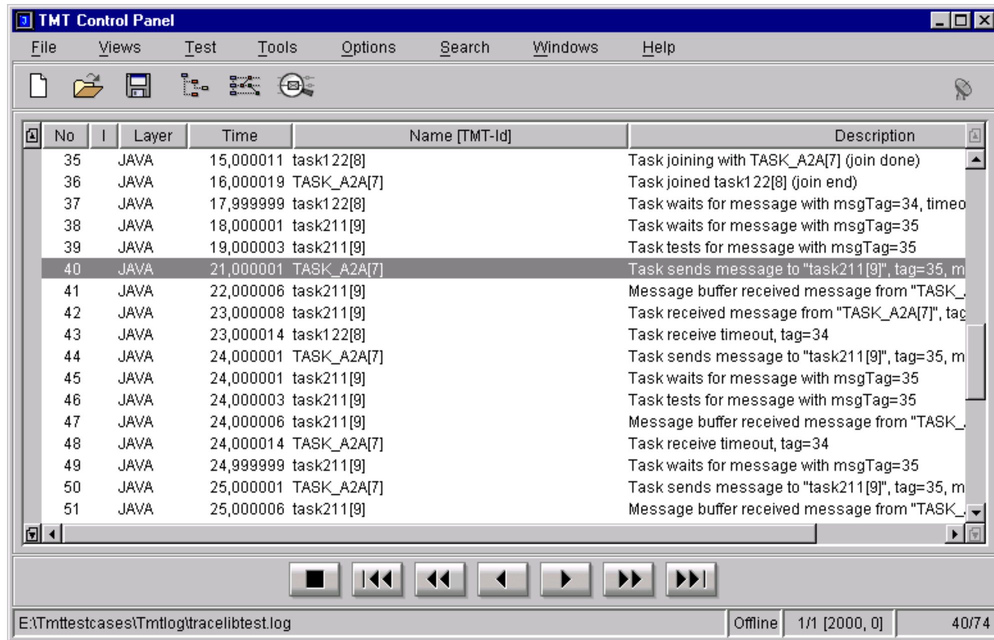

Figure 7. Main Panel of TMT.

The main panel shows the list of recorded test events observed from the SUT. The events are listed in an ascending temporal order of their receipt by the tester. Since the time stamps are generated locally, i.e., on the local machines where components of the distributed system are running during the test, the received sequential order of test events might not reflect the correct causal dependencies between the events if the accuracy of local clock synchronization is insufficient. This is however not a drawback since TMT considers only the causal dependency of test events for functional testing. The values of the time stamps matter only in case of performance tests that is not focus of this paper.

Further functions of the main panel of TMT include buttons to open different view charts of the SUT in the upper part of the window and VCR-like buttons to scroll through the event list.

4.2 Operation Modes of TMT

The TMT prototype supports two operation modes: on-line and off-line mode. In the on-line mode, TMT collects and displays all test events during the test run. Test events are produced from an instrumented source code of the SUT and refer to interaction events within components of the SUT or between SUT and environment. Thus, TMT realizes a global tester.

The off-line mode, on the other hand, allows to record partial information of a test run in distributed tester components. The tester components produce local files containing the observed test events during the test run. These local files are read by TMT after the test run has finished. The local information of test event sequences are merged within TMT according to the time stamp values of the test events. In this scenario, TMT realizes a distributed tester. The distributed tester components allow currently only to write observed test events to a file, but do not perform an assessment of the (partially observed) test run. This functionality is still centralized within the main tool of TMT.

Since the tester components do not use test coordination procedures between them, the observed test events must be issued redundantly by the SUT as discussed in Section 3.2.3. This is entirely in the responsibility of the developer of the distributed system to provide a SUT that is sufficiently instrumented to produce the desired test events.

4.3 The Test Library

TMT realizes a monitor tool and a distributed tester based on a redundant observation of internal interactions of the SUT. To generate the distributed tester components for a specific SUT with a set of communicating system components, a test library is provided that contains function calls to record test events by the tester. This test library resides on the machine where a component of the SUT is running. Connection between the tester and the SUT is established in such a way that the source code of the SUT is augmented with additional function calls of the test library in cases where interactions between components of the SUT will occur.

The test library does not allow in its current version to perform also the assessment of a test run. This functionality is entirely moved to the central main part of TMT (see Figure 7). The tester components implemented in the test library only allow to record the observed test events in a file or to send them directly to the TMT main component. The advantage of this approach is that the test library is generic for a large variety of implementations of distributed systems. Since the augmentation of the SUT is done in its source code, implementations in C++ and Java are currently supported by the test library only.

4.4 The Sequence Chart View of TMT

This view shows the execution process of interactions within a distributed system under test in a *Sequence Chart View* that is similar to a MSC graph. The test events received from the SUT are ordered according to their causal dependencies. Each sequential component is represented as a separate line within this view. Different icons in this view denote different interaction events (send, receipt of messages, creation of a new component and so on). See next Figure.

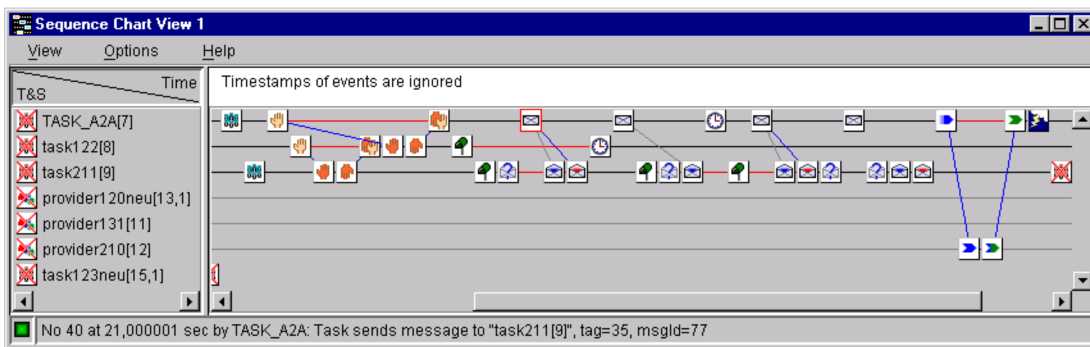

Figure 8. Sequence chart view of TMT.

Other views provided with TMT are

- a *Hierarchical View* on the structure of the SUT that depicts the composition of the SUT in components and sub-components, and

- a *Detailed View* to show details on received test events, e.g. certain parameter values.

4.5 Interactive Mode of TMT

TMT allows the generation of special interactive test events that help the tester control the subsequent execution of interactions in the SUT. At the moment this test event requires a manual interaction with the user of TMT. For this purpose a windows pops up that requires user inputs (see Figure 9). The further execution of the test architecture waits until this desired user interaction was performed. In this way, control over the behavior of the SUT can be achieved by TMT. This control is necessary to avoid non-determinism and to assure a deterministic test run (see Section 2.3).

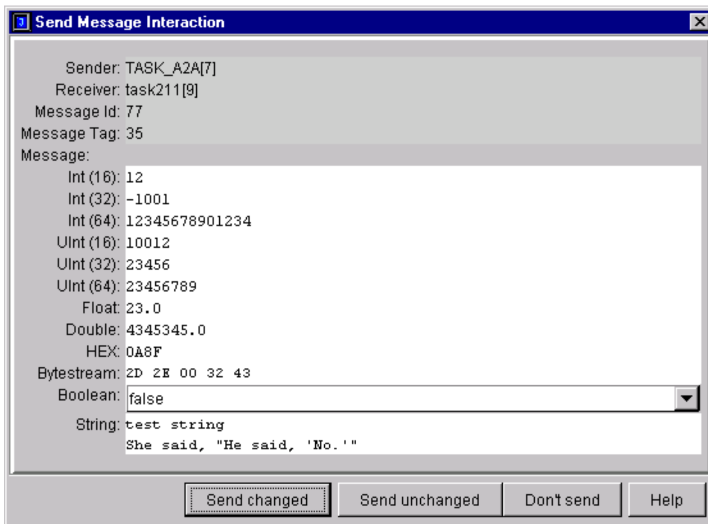

Figure 9. User interaction pop-up window of TMT.

4.6 Assessing Test Runs

An important feature of TMT is its ability to compare test runs with test cases or other test cases obtained in previous test runs. This assessment is, unlike the discussion in Section 3.2, is performed centralized in the main component of TMT to allow more flexibility of TMT for different test runs.

The implemented feature of assessing test runs takes into consideration the possible concurrency among the observed test events in order to correctly assign the test verdict. For example, although the two sequence chart views in Figure 10 look different, the test events in these views still obey the same causal dependencies. Thus, both test event sequences are considered to be equivalent by TMT, and the test verdict assigned to this test run is *pass*.

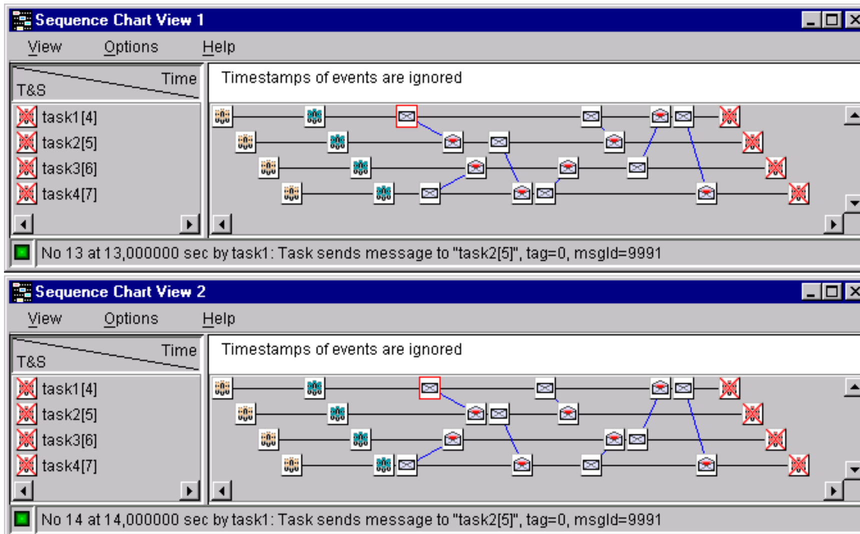

Figure 10. Two similar test runs.

5 CONCLUSIONS

In this paper we have suggested different architectures for the test of distributed systems. Two principle test architectures have been considered: a global and a distributed tester. The global tester is a sequential module which observes and controls all actions of the SUTs at its PCOs, whereas the distributed tester consists of several concurrent tester components which observe a partial behavior of the SUT only, but supports concurrency within the SUT. The latter must provide test coordination procedures to assure a correct and consistent global view on the SUT.

A prototype implementation TMT of a distributed tester was also presented. The current version of this prototype allows to monitor distributed applications and to assess a recorded test run according to a test case derived from a specification or recorded from a reference test run done before. This test assessment does not compare the test events of a recorded global test sequence in a sequential way, one after the other, but exploits the causal dependencies between test events to avoid misinterpretation caused by interleaving concurrent events.

6 ACKNOWLEDGMENT

The authors wish to thank all colleagues within Siemens who have participated in the TMT project. Special thanks in particular to Kai Tödter, ZT SE 2, and Klaus Berg, ZT SE 1, who designed the TMT architecture. Many thanks also to Michael Kälbling, Anja Hentschel, ZT SE 5, and Jürgen Schmitz-Foster, ZT SE 2, who implemented most of the TMT features, as well as to Sylvia Jell, ZT SE 2, who was responsible for the test library implementation of TMT. Last but not least, we thank Dietmar Lehmann, ICN WN ES D 42, and Erwin Reyzl, ZT SE 2, who helped funding and organizing the project.

7 REFERENCES

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

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

[3] 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.

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

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

[6] 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.

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

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

[9] Z.120: *Message Sequence Chart (MSC)*; ITU-T, Geneva, April 1996.

8 BIOGRAPHIES

Andreas received the M.Sc. degree (Diplomingenieur) in Computer Sciences from the University of Magdeburg, Germany, in 1992 and his Ph.D. degree (Doktoringenieur) from the same university in 1998. In 1998 he joined the Corporate Technology Division, the research and development center at Siemens AG, in Munich, Germany. His research interests include test architectures for distributed and concurrent systems, test case generation and specification-based testing, and formal specification techniques. He is an active participant at international conferences such as PSTV, FORTE, IWTCS and others.

Peter studied Computer Science at the University of Stuttgart, Germany and received his M.Sc. degree (Diplominformatiker) in 1991. He then joined the Siemens AG, Corporate Technology, and has been working in the field of software testing for object-oriented (C++, Java), distributed, and component-based software. He is co-author of several international conference publications, e.g. at EuroStar, Conference on Testing Computer Software and Software Quality Week.

Gunther is with Siemens AG, Corporate Technology in Munich, Germany. His current interests are testing distributed systems, component-based and object-oriented software. Gunther started his career in 1992 at DLR, the German Aerospace Center. He holds a M.Sc. degree in Applied Mathematics from the University of Kaiserslautern and a M.Sc. degree in Meteorology from the University of Hannover.