TESTING AND DEBUGGING OF DISTRIBUTED SOFTWARE

José C. CUNHA

Department of Computer Science
Faculty of Sciences and Technology
New University of Lisbon
2825-114 Caparica, Portugal
e-mail: jcc@di.fct.unl.pt

Henryk KRAWCZYK

Faculty of Electronics, Telecommunications
and Informatics
Technical University of Gdansk
ul. Narutowicza 11/12
80-952 Gdansk, Poland
e-mail: hkrawk@pg.gda.pl

Abstract. This paper introduces the topic of testing and debugging of distributed software in this special issue of the Computers and Artificial Intelligence Journal. A global picture is given of the problems involved in developing distributed applications in order to motivate the need for testing and debugging activities. The main issues and approaches of testing and debugging are surveyed, the focus being on the identification of current and future trends. We conclude by introducing the papers which were selected for this special issue.

Keywords: Testing and debugging, distributed software.

1 DEVELOPING DISTRIBUTED SOFTWARE

The need of testing and debugging was recognized since the early days of computing. The testing activity searches for undesired results and patterns of behavior in an application. The debugging activity searches for the reasons of such undesired
behavior, aiming at locating and removing them from the program code. The whole process is based on a cycle involving the specification of selected program inputs or test data, the specification of the expected program behavior, and its comparison with the actually observed program results and behavior, followed by steps to correct the program code, and further analysis of the modified program.

The increased use of distributed computing platforms in a wide range of applications has made the task of developing distributed software more and more difficult. Traditional difficulties in precisely determining and evaluating the correctness and performance of a distributed application are mainly due to the need of dealing with the concurrency and nondeterminism issues. The widespread use of distributed systems technologies brings additional dimensions to this problem:

- Applications are more complex, and consist of a larger number of separately developed components.
- A large diversity of target heterogeneous execution platforms must be considered, and their distinct performance, scale and reliability characteristics must be taken into account.
- One must anticipate how evolving user and application requirements will influence the software architecture of a distributed application, and how such changing requirements can be satisfied without requiring a complete design of the application.

The focus of attention must be directed to a more precise identification and specification of the quality attributes of an application, corresponding to its desired functionality, performance, dependability and flexibility. Methodologies are required to help determining and evaluating the global quality of a distributed application in terms of such attributes. Finally, adequate frameworks and tools are needed to support the use of those methodologies, in an iterative and incremental fashion, during the whole lifespan of a software application.

The structure of the paper is as follows. Section 2 presents a brief survey of testing and debugging approaches. In Sections 3 and 4 current and future trends on distributed testing and debugging are discussed. Section 5 presents some conclusions on open issues. Finally, in Section 6 we present the papers selected for this Special Issue.

2 APPROACHES FOR DISTRIBUTED TESTING AND DEBUGGING

While testing is mainly concerned with the detection of errors, debugging relies on the observation of repeated executions of a program in order to find and locate the causes of those errors. Observing consistent global states of a distributed computation is difficult due to the characteristics of asynchronous distributed systems, namely the absence of a global clock and the unknown bounds on message communication times. On the other hand the nondeterministic behavior and the intrusion (or
probe) effect make it difficult to ensure deterministic re-execution of a distributed program, as required by the debugging activity.

The first step towards building a distributed debugger is to provide a mechanism to allow remote observation and control of the execution of the distributed sequential processes which are spawned by a distributed application. However, in order to allow deterministic re-execution of a given computation path, trace and replay approaches need to be applied. Such approaches require the use of specific techniques aiming at reducing the intrusion effect and the trace size, so that the resulting debugging systems are feasible in practice. In order to be able to specify and generate specific program execution paths which can then be repeatedly examined by a debugger, further approaches must be used. These approaches usually rely upon an integration of static and dynamic analysis. The basic idea consists of having an interactive testing tool allowing to specify the interesting regions of a distributed program which should be considered for analysis. After such analysis, the testing tool is able to generate information defining the corresponding testing scenarios which are then used to guide a controlled execution of the program, under the supervision of a distributed debugger. This supports a cyclic interactive testing and debugging session such that the user can systematically select and inspect program behavior under well defined conditions. A complementary approach can try to automate, at least partially, the above search by allowing the user to specify the relevant program correctness criteria in terms of global predicates. Once a global predicate is detected, the computation must be suspended at a consistent global state that satisfies the evaluated global condition. Although the efficient detection of such global predicates is limited to restricted classes of predicates, several strategies have been proposed for constructing such global states and halting the computation so that further examination of the error situations can be performed by the user.

Approaches like the ones mentioned above have been investigated and used to develop several distributed testing and debugging tools and environments [7], [18].

An important goal is to allow the user to specify and evaluate the program correctness conditions in terms of the concepts and abstractions defined by the distributed programming model, which may be called high-level debugging. Due to the large diversity of distributed programming models, a distributed debugging architecture should provide a clear separation between the support for high level debugging abstractions and the support for the low level observation and control services at the execution level. Several proposals have been made that can help us meeting the above goal [6], [13]. In this regard it is important to mention the task of the High Performance Debugging Forum, as a way to push forward the adoption of standard debugging interfaces [11].

The requirement to provide a flexible distributed debugging architecture becomes even more important for the development of applications composed of multiple heterogeneous components. Indeed, program analysis tools for such applications must address the abstractions for the (possibly dynamic) composition and configuration of an application, and for component interconnection, so that the user can reason in terms of the global structure and properties of the application. This will
put new challenges to the task of developing testing and debugging tools and environments.

3 NEW CHALLENGES IN SOFTWARE TESTING AND DEBUGGING

In the following sections, a brief overview is given of current and future trends concerning testing and debugging of distributed applications.

3.1 Abundance of Testing and Debugging Strategies

In many cases testing (detection of bugs) and debugging (their localization and elimination) is commonly used in software practice. In general the basic testing and debugging (T&D) procedures consist of four steps [21].

- Modeling the software environments to analyze testing contents, i.e., user interfaces, API, file system interfaces or communication interfaces.
- Selecting suitable testing scenarios from the set of possibilities according to the assumed testing criteria.
- Running these scenarios in both artificial and real environments, monitoring and logging software behavior for on- or off-line analysis.
- Identification and correction of software anomalies and measuring T&D progress to finish the procedure.

This general procedure can be modified and implemented in practice in different ways, under various circumstances.

Table 1 shows examples of various T&D strategies in accordance to phase of software development cycle, utilized programming platforms, and testing levels, i.e., what piece of software is actually under test. Additionally, available test bed, as a given computing platform with additional supporting tools, can also influence the task of any concrete T&D procedure. In consequence we have many testing methodologies, test design patterns, test cases, bug models and testing tools described in many books and defined standards [2], [7], [16].

Due to T&D complexity, time constraint and limited cost of testing, proposed testing strategies cannot check all combinations of data, cannot consider all orders in which statements can be executed, and cannot create the real user environment where the software products are running. In consequence, incomplete testing leaves users with an untested code and causes frustration when bugs happen during software exploitation. Therefore the basic challenge is still the same — how to design such T&D procedures which allow not only to minimize the number of residual bugs in software products, but also to evaluate them from the widest possible perspective, including many aspects of software quality.
In the recent past, T&D strategies focused on functionality and performance. Today, software is much larger and more complex objectives are taken into account [1]. A suitable T&D methodology is proposed which analyzes four test characteristics: interoperability, integrity, availability and performance. Interoperability determines which components interact and communicate with each other through commonly agreed mechanisms and protocols. Integrity focuses on data and system state if they are consistent at any given point in time. Availability determines if the software is available to provide services. Performance refers how the software responds to request and delivers acceptable services in a given time. Each of test cases could be used for more than one purpose. By analyzing many possibilities we try to find a compromise solution, which will ratify a given multi-criterion. In spite of cause-effect analysis, event exploration, suitable measurement techniques should be used as a basis for T&D strategies. In other words, T&D methodology will become closer to quality evaluation and quality assurance techniques. To reduce the high cost of software development (including testing cost), an interesting approach is largely used in a wide range of applications. It is called COSTS (Commercial Off The Shelf) where system software is made up of some standard components. Developers want to know whether such components themselves are reliable and whether the system partially made of these components will tolerate them. To evaluate these capabilities, a methodology [23] is based on black box component testing, uses fault injection techniques on system level and recommends wrapping methods for systems or components modifications. Typical certification procedure consists of several testing scenarios which can be automated. Such parameters as test scalability, test portability, cost of implementation, and effectiveness, can be estimated. Based on this information, better decisions can be taken and in consequence a higher quality system can be created.

In general, practical software is developing much faster than T&D theory and practice. Strategies such as α-, β-testing and regression testing [20] are widely acceptable. Therefore an integration of efforts is needed to create efficient and effective testing strategies, useful for new kind of user applications, such as multimedia systems, e-commerce or e-government, working in Internet or Wireless Internet environments.

A broad view on the detailed problems is shown in Tables 1 and 2. Table 1 classifies various types of testing activities and procedures with regard to four major aspects of software development, namely software life cycle models, programming platforms, abstraction levels and hardware environments.

### 3.2 Barriers in Software T&D

The main problem in T&D of software applications is completeness of testing procedures. This means that only small size programs can be tested exhaustively. In other cases only heuristic strategies can be executed in acceptable time. However, this causes a new problem related to T&D completeness. In general, test completeness is described by a test coverage parameter defined as the ratio of all checked
<table>
<thead>
<tr>
<th>Software development specific</th>
<th>Programming platforms specific</th>
<th>Unit under test specific</th>
<th>Test bed specific (Hardware environment)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformance testing, Unit testing, Integration testing, System testing, Configuration testing, Acceptance testing, α-testing, β-testing, Certification, Regression testing</td>
<td>Procedural testing, Object testing, WWW testing, Intelligent mobile agents testing</td>
<td>Module testing, Thread /process testing, Objects testing, Pages testing, Component testing, Application testing, Service testing, System testing</td>
<td>Single workstation, Network of workstations, Grid architecture, Internet environment, Wireless internet</td>
</tr>
</tbody>
</table>

Table 1. Specific aspects of testing/debugging strategies of user applications

mechanisms (e.g., instructions, paths, decisions modes, processes, objects) to all such mechanisms existing in a tested application. Besides, each software product must satisfy its user/designer requirements and all test cases should be traceable to these requirements. Because of their incompleteness and ambiguity, requirement coverage may be introduced. It is defined in a similar way as the test coverage. The requirement coverage is a ratio of all analyzed requirements to all required ones corresponding to user needs. In general the fault coverage can be expressed as a product of the test coverage and the requirement coverage. However, the fault coverage can be exactly assessed only for very well specified programs, for instance described by UML [3] notations. Otherwise a fault injection technique allows to estimate the above parameters.

Test completeness is strictly related also to two other problems: observability (possibility of monitoring real behavior of the tested applications) and reproducibility (possibility of recalculating states of the application under test). The high observability allows a tester to distinguish between potentially different behavior of fault-free and faulty applications by monitoring only their inputs and outputs. The high reproducibility means that the distributed application can be repeatedly executed for a given time interval, e.g., despite the inherent asynchrony of communication events. To improve both of the above parameters different kind of traps (checkpoints) can be used to monitor executions of various actions. However, these extra mechanisms can change the behavior of the applications so much that the collected data will not be adequate, what is known as the intrusion or probe effect. Therefore,
for a given application the testing scenarios must be carefully defined. This requires high understandability of source program of the applications. There may arise a new problem, since object programming software has a rich structure (methods, attributes, classes, association, aggregation, inheritance) which make object-oriented software hard to understand.

In OO-procedural languages tracing procedures focus mostly on capturing function call sequences. In OO-languages two most important issues are capturing object lifetime (from the object creation to destruction), to monitoring problems at such application, and identifying the objects associated with each method invocation. There are three solutions. The first is to use program instrumentation which augments source code with trace generating code. The second is based on meta-class-programming, and is limited to object models based only on the metaclass concept. The third is to monitor objects on the basis of activation and deactivation of the functions in which the object is embedded. These ideas are implemented in the prototype tool called Program Explorer [17].

The more general concept of traceability concerns the ability to justify a modelling or coding detail by tracing it back to a requirement [5]. Such approach is useful for the development of software applications such that, when progressing through successive iterations, it is possible to converge towards the desired solution (see the next section). Software requirements can change in time. These changes and enhancement requests are usually specified in terms of the features affected not in terms of software components that must be modified. Software developers must locate and understand the code associated with the affected features before they can translate the change request into code changes. There is a problem how to exactly describe these activities and how to check their correctness. Moreover, no method has explicitly and effectively dealt yet with the bug detection specifications being nondeterministic, i.e., an application permits multiple output for the same sequence input events and internal state. There are many sources of nondeterminism, such as communication channel delay, spontaneous state transitions or nondeterministic decisions. There is an open problem how to determine and control causality in OO software, in order to make its possible re-execution or state and computational recovery. This is necessary to achieve repeatability of testing scenarios.

Fast development of WWW-applications opens new serious problems, strictly following from inadequate level of security and safety. We need new kind of testing — say security and safety testing. The aim of security testing is to detect such events where system functions are unintended or unauthorized [9]. The aim of safety testing is to predict and protect such erroneous situations which can eliminate the risk of a global catastrophe destroying the whole Internet infrastructure. The example of such a situation was the Y2K problem, where before January 1, 2000 hundreds of millions of lines of legacy code had to be understood, modified and re-tested to avoid software failures as the year changed from “99” to “00”. Generalizing this problem we can define new kind of hazards, so called imminent bugs. Even in fault free software, due to some changes in software environment many software activities can become unacceptable. The problem is how to predict such kinds of
faults, and how to take proper decisions in acceptably short time. In collaborative computing systems where human decisions are included directly in computations, user misunderstanding and user misinterpretation can have serious impact on system dependability. These are sources of another kind of hazards, which need completely new solutions.

<table>
<thead>
<tr>
<th>Processing models</th>
<th>Types of bugs</th>
<th>Basic testing methods</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Data, Computation, Control</td>
<td>Functional (black box), Structural (white box), Mixed (grey box),</td>
<td>Complexity, Observability, Test coverage</td>
</tr>
<tr>
<td>Concurrent/parallel</td>
<td>Synchronisation (race) Communication, Configuration, Performance</td>
<td>Random execution, Controlled execution, Determined execution, Concurrent testing and processing</td>
<td>Reproducibility, Probe effects, Tractability, Non-determinism, Causality, State recovery</td>
</tr>
<tr>
<td>(thread), Distributed (client/server)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed (peer to peer), Collaborative</td>
<td>Unpredictable exceptions, Hazards, User misunderstanding, User misinterpretation</td>
<td>Quality oriented testing scenarios</td>
<td>Interoperability, Security, Safety, Computation recovery, Measurements, Tuning</td>
</tr>
</tbody>
</table>

Table 2. Classification of bugs and testing/debugging methods

Table 2 summarizes these considerations by distinguishing basic problems according to some items. It relates types of bugs with processing models, indicates corresponding testing methods and names hard to solve research problems.

4 TESTING AND DEBUGGING IN THE WHOLE SOFTWARE LIFE CYCLE

In this section, we illustrate the importance of providing a stepwise development model that can be applied to evaluate successive modifications made to a software application.
4.1 Stepwise Software Development Model

The traditional approach to software development, known as the “waterfall model”, has been criticized for not being able to cope with changing product specifications and design, the poor feedback from customers, the lack of risk assessment incurred by the changing market demand and no provision for ongoing tests of the evolving product [8]. Its tendency to limit product development by a strictly defined set of requirements that must be known before any design and implementation can ever be started makes it less useful today, despite its heavy emphasis on advance planning and attention to architectural and detailed design before any coding may actually take place. An alternative “iterative enhancement” model with an essential feature of the iteration model is a gradual enhancement of successive implementations in order to build a matured final product. It starts from a simple skeletal problem constituting a subset of the final set of requirements specification not necessarily known in advance — the latter would have been required to be known when using a “waterfall” model. A project control list is created to indicate all enhancements that are required to achieve the desired final implementation by starting from the current one. Each iterative step consists of selecting and removing tasks from the list by designing an enhancement into the evolving product, implementing and debugging its code, as well as analyzing the modified product in order to assess its quality and eventually identify new enhancements to be put to the project list.

An advantage of the iteration model is its ability to measure various quality characteristics of a product not yet fully complete in a series of simple experiments, which not only enable close monitoring of the development process, but also facilitate gradual development of test suites capable of demonstrating the evolving product quality.

Fig. 1. Two levels of stepwise software development

It can be seen in Figure 1 that there are two levels of development:

- a big circle involving relevant techniques and mechanisms provided by the programming platforms to enable achieving the required quality characteristic, and
• a small circle for tuning specific parameters of the implementation that contribute to some quality characteristic of interest.

Although characteristics being considered during the evolutionary development cycle may involve various quality attributes, throughout the rest of this paper we specifically focus on achieving a required level of dependability. We will argue that due to various complex relationships between quality attributes evaluation of the evolving product quality is multi-criterial and does not yield unanimous assessments. Specifically performance will be considered, as an often conflicting quality requirement. Our claims will be supported with a realistic case study and finally procedures for estimating performance and dependability will be outlined.

Let us consider now two factors, namely dependability (D) and performance (P). Let $d_i, p_i$ denote the value of dependability and performance, respectively, evaluated in each $i$-th development step, $i = 0, 1, \ldots, k$. $d_i \in \mathcal{D}$ and $p_i \in \mathcal{P}$. All possible evaluations of $d_i$ and $p_i$ constitute space $\mathcal{S} = \mathcal{D} \times \mathcal{P}$. Iterated development represents a trajectory $\theta : I \rightarrow \mathcal{S}$, where $I$ denotes a set of step indices. Let initial properties of the application under development be determined as user requirements $R(D, P)$ in step $i = 0$, while the characteristics obtained in step $i = k$ represent final product quality $Q(DP)$. Let $\Omega$ measure consistency $\omega$ of $R$ and $Q$, i.e., $\omega = \Omega(R, Q)$. During the stepwise development we look for $\min_{\omega}(\Omega)$. Unfortunately, we cannot practically describe $P$ and $D$ as stochastic processes normally assumed in performability analysis. What we need instead is a relationship between $P$ and $D$ such that proper decisions may be taken based in each development step.

4.2 Case Study – an Intuitive Approach

Consider a distributed system for remote monitoring of cooling containers. Each such container has a microprocessor for controlling the freezing temperature inside the container and repeatedly reporting measurements to the command center aboard a ship carrying them, or inside some confined storage area in the harbor if not in transit. Communication between each single container (sender $S$) and the command center (receiver $R$) involves a simple send-message get-acknowledgment scheme. Figure 2 shows schematically an iterative development of this application, starting from the skeletal implementation of the communication scheme and stepping towards more dependable implementations.

The skeletal implementation shown in Figure 2a is capable of performing a single communication, but needs enhancements to allow for many messages to be sent by $S$ without waiting for acknowledgment from $R$, as well as to achieve a good buffer utilization, especially when its size $|buf| >> |msg|$, the average message length. In this step it is already possible to design a test for checking whether a message being sent to $R$ is properly acknowledged by $R$. The skeletal implementation has been written in C and utilizes a TCP/IP-like datagram communication [4].

In the next step (Figure 2b) all postulated enhancements have been added: action $s$ is using buffered (non-blocking) send, thus the inner loop in $S$ will be able to
send as many messages as possible, while action n is a non-blocking receive statement
iterated by the outer loop. This implementation may rely upon the solutions provided by
the PVM-like platforms [12], with a richer offer of communication protocols than
TCP/IP, namely asynchronous non-blocking message passing. At this stage it
may also be believed to have better dependability characteristics than TCP/IP used
in the previous step. Moreover, in the step in Figure 2b the existing test suite may
be extended to exercise utilization of communication buffers. Further experiments
may, however, reveal still too poor dependability because of occasional failures of S
or R during regular communication. This enhancement is considered in the third
step shown in Figure 2c. Diagnostic code returned by functions implementing send
actions s may be checked to raise an exception, which in turn is handled by spawning
the failed counterpart process. Note that we still stay with PVM, but may want to
switch from C to C++ offering a robust exception handling mechanism.

At this stage of product development we may extend our test suite further with
test cases specifically exercising failures of receiving processes that may damage the
communication scheme specified here. Note that the solution introduced in this
step has a minor deficiency, namely all messages awaiting delivery will be lost for
the newly spawned (recovered) processes. This observation gives reason to the next
enhancement, specifically using a replication of processes that may be lost. We do
this in Figure 2d by creating a group of replicated processes, capable of acting as one
sender S. In this step we decide to switch with our implementation to a CORBA
based platform [19], supporting replication and a reliable group communication using
a client-server paradigm. Although this solution implements full fault-tolerance and
has the best dependability characteristics of all steps in Figure 2, efficiency of the
replication scheme used in the last step may be found sometimes unsatisfactory
during tests.

Fig. 2. An example of stepwise enhancement
Let us denote by $D_i$ decisions taken during project development, and by $T_i$ a test suite for evaluating these decisions. Decisions and tests become more accurate as software development progresses. Table 3 collects all successive dependability requirements, related tests and development decisions.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Tests</th>
<th>Decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer utilization ($D_{11}$)</td>
<td>relevance of buffer size to node failure ($T_{11}$)</td>
<td>improve dependability</td>
</tr>
<tr>
<td>platform choice ($D_{12}$)</td>
<td>performance acceptance ($T_{12}$)</td>
<td></td>
</tr>
<tr>
<td>process-level exception</td>
<td>fault-injection message recovery ($T_{11}$)</td>
<td>improve performance</td>
</tr>
<tr>
<td>handling ($D_{13}$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>group-level process replication ($D_{14}$)</td>
<td>$T_{11} \cup T_{12} \cup T_{13}$</td>
<td>change to client-server</td>
</tr>
</tbody>
</table>

Table 3. Description of development in Figure 2

Decisions taken at the respective steps of this example application development are rather intuitive and are based on professional engineering experience. There is no any complete theory fully supporting decisions listed in Table 4.2. We would like however, to attempt finding some formulas to guide us in working out a consensus, maximizing the dependability to performance ratio.

In realistic applications related decisions would certainly be more complex, but we wanted to outline the general methodology and estimate some basic relationships between dependability understood as a number of defects and performance understood as application complexity. For this simplistic view we have obtained an interesting “heuristic” formula, generalizing the reliability growth model, although such arithmetic formulas are not sufficient in general to justify development decisions, and some logical formulas must be provided additionally. This implies collecting a database knowledge, constituting a basis for at least a simple expert system.

5 CONCLUSIONS

Summarizing our considerations we list below some observed trends in T&DC methodology and formulate some well-known questions which are still open. Among different trends we choose the following:

1. Practical testing methods and tools may be grouped into two generic classes:
   - Feature-based, concentrating on various language construct abuses and anomalies;
• Behavior oriented, taking into account reaction of programs on various stimuli, like internal and external events and environmental changes.

2. Bug testing migrates from towards general quality testing concentrating on all quality attributes such as functionality user satisfaction, performance, dependability and flexibility.

3. A new component oriented design and design for testability is expected to decrease the high cost of testing and increase low test coverage. This corresponds to significant decreases in the average bugs ratio for applications with fully reused components when compared to applications without reuse.

4. There is a big need for new testing concepts for modern networked applications, various services and WWW processing systems.

5. A growing market pressure on security and safety of distributed and collaborative software applications requires to analyze a new class of bugs, so called imminent bugs.

Besides, there are still many open problems which can be expressed as the following questions:

1. Why software testing is so hard?

2. How to test modern application services and systems?

3. How to estimate quality of test cases and testing scenarios?

4. What role does testing play in quality evaluation, quality control and quality assurance?

   Answers are not easy because existing testing models do not describe sufficiently and completely all the testing, especially those encountered in Web-based information systems. We can rephrase the above then as two important questions:

5. What consequences of bugs will be in the global Internet environment?

6. How to assure the required level of dependability for WWW software applications?

After answering to the above questions we can evaluate the maturity of T&D methodologies.

6 THE PAPERS IN THIS ISSUE

The papers which were selected for this Special Issue provide a representative sample of current research towards improving testing and debugging of distributed software.
The first paper, by Jacques Chassin de Kergommeaux, Michiel Ronsse, and Koen De Bosschere [14], is about execution replay. This research concerns the development of program development support tools for the Athapaskan programming model. Athapaskan supports message based communication between distributed threads which run on distinct computing nodes, such that shared-memory communication is allowed between threads within each node. This model is suitable to exploit distributed clusters consisting of multiple shared-memory multiprocessors. The paper presents a detailed discussion of an efficient implementation of execution replay, focusing on the reduction of the intrusion effect and of the trace size, in this shared-memory/distributed-memory model.

The second paper, by Bogdan Wiszniewski [22], is about testing and debugging of distributed object-oriented systems. The paper discusses how testing and debugging can extend to distributed applications consisting of heterogeneous components. The paper focuses mostly on design issues and presents the basic motivations and needs to go for more advanced distributed testing and debugging frameworks. In this sense, this paper addresses several of the new challenges which were identified in a previous section.

The third paper, by Chyi-Ren Dow and Cheng-Min Lin [10], is about the detection of global breakpoints and the construction of consistent global states, in relation to the evaluation of certain classes of global predicates. The authors report on their work towards more efficient solutions to reconstruct such global states, based on checkpoints and the reduction of their size.

The fourth paper, by Dieter Kranzlmuller [15], discusses an approach for incremental tracing and replay, that supports the debugging of isolated processes. The approach allows the user to interactively select the process of interest and then it provides the necessary tracing data allowing the user to perform deterministic re-execution of the process. The information describing the interaction of the selected process with other processes is automatically managed by the system.

REFERENCES


Software.


Manuscript received 22 November 2000

José C. Cunha was born in Viseu, Portugal. He got the Diploma in Electrical Engineering from Instituto Superior Técnico in 1975, the Diploma in Informatics Engineering in 1979, and a PhD in Computer Science from Universidade Nova de Lisboa (FCT/UNL) in 1989. He is associate professor in the Computer Science Department of FCT/UNL. He is the head of the Computational Systems and Architecture Group and of the Parallel and Distributed Processing Group. He coordinates research in distributed languages, tools, and heterogeneous environments, and has been publishing regularly in these topics.

Henryk Krawczyk was born in 1946. He graduated, received Ph.D., and D.Sc. all from TUG, in 1969, 1976 and 1987, respectively. He has been nominated full professor in 1995. Currently he is a professor and chair of the Department of Computer Architecture at the Faculty of Electronics, Telecommunications and Informatics at TUG. His research interests include parallel and distributed systems, dependability analysis, Internet applications and quality assurance. He has authored or co-authored five books and over 200 papers published in journals and conference proceedings worldwide.