Distributed Expectation-Driven Residual Testing

Leila Naslavsky, Roberto S. Silva Filho, Cleidson R. B. de Souza, Marcio Dias, Debra Richardson, David Redmiles
Department of Informatics
School of Information and Computer Science
University of California, Irvine
{lnaslavs,rsilvafi,cdesouza,mdias,djr,redmiles}@ics.uci.edu

Abstract

Software testing techniques are widely used to assure properties such as dependability, as well as to gather system usage information to support software evolution. However, in spite of such techniques, comprehensive testing of commercial-grade software is rarely performed, and thus software is released with neglected test obligations. The dissemination power of the Internet and the increased connectivity of systems have made the pressure to release software earlier even more apparent. Fortunately, at the same time, researchers have investigated using the infrastructure of the Internet for performing remote testing and analysis. One particular approach called residual testing allows the testing of software after it is deployed to beta testers and end users. This paper builds upon existing residual testing techniques, presenting an approach to integrate residual testing with a way to dynamically measure the behavioral conformity of the deployed system with its specifications. Our approach uses the facility of an extensible publish/subscribe infrastructure, dynamic instrumentation techniques and software tomography to address many of the issues faced by current remote residual testing strategies.

***Keywords**: Software Test Coverage Criteria, Distributed Residual Testing, Software Monitoring, Publish/subscribe.*

1. Introduction

As part of the quality assurance process (QA), software testing activities must always deal with the accuracy versus cost trade-off [1]. One can aim at proving system correctness, performing exhaustive testing or covering all the possible execution paths at the expense of infinite effort. Yet, time-to-market pressures, cost constraints, difficulty in predicting all possible combinations of hardware and software configurations, and other concerns impose practical limitations to the amount of testing that is accomplished. As a consequence, unforeseen problems are usually manifested in the field (on the client site), when software is exposed to different configurations and usage patterns. In this context, monitoring and identification of such problems provide useful information when reported back to the QA team, which facilitates fixing any defects identified or adapting and evolving the software to new configurations.

Residual testing techniques, as introduced by Pavlopulou and Young [2], allow software developers to cope with this inevitable situation. These strategies test software released either to beta testers or to end-users with respect to its remaining obligations (the residue). They employ selective software instrumentation and monitoring techniques to collect usage data for the purpose of software quality assurance and evolution. Combining their techniques with the connectivity of the Internet allows the automatic collection and delivery of the monitored information from the client sites to the software QA and development teams. Indeed, many techniques have been proposed that allow for monitoring of deployed software behavior aiming at isolating errors [3], increasing coverage with minimal degradation of system performance [4, 5], and distributing QA tasks to users [6], among other goals.

Clearly, ensuring that a part of the system was covered does not eliminate the need for ensuring that the results produced and the behavior of the system match the specification of the software. Thus, one of the major needs for the residual testing approaches is to add support for behavioral conformance. In this paper, we propose an approach to address this issue that allows QA and development teams to perform behavior validation of remotely deployed software instances. The main idea is to translate remote execution information into higher-level constructs, that can be combined with specification-based test oracles to check system correctness [7, 8]. System correctness can be verified with respect to the behavioral specification of the system, as expressed in many different ways such as: state machines, UML sequence and/or collaboration diagrams, among others. Our approach integrates residual testing and behavior verification to dynamically and incrementally measure the conformity of the deployed system with its specifications. Our ultimate goal is not only identify specified behavior but also detect unpredicted or abnormal usage of the system. We argue that we can achieve this goal by combining different techniques, including specification-based test oracles, dynamic instrumentation approaches that support changing monitored focus at runtime, software tomography [4], a publish/subscribe infrastructure, client-side event filters, event correlation and visualization gauges.

The remaining of this paper details our approach. It is organized as follows. Section 2 describes some existing approaches. After that, section 3 explains our approach as well as the current state of our implementation. Finally, section 4 provides conclusions and future work.

2. Related Work

In addition to the work of Pavlopulou and Young [2], several systems have been used to remotely monitor aspects of software after its deployment for testing purposes. As a matter of fact, most of them focus on low-level aspects such as statement or method coverage, hot-spots identification, error and exception detection and others. This section describes some of those systems.

The Gamma system [4] uses software tomography [9] to achieve low-impact and minimally-intrusive monitoring. Software tomography consists on dividing the overall system monitoring task (measured as statement coverage, for example) into different subtasks which are strategically instrumented and assigned to different deployed software copies. Thus, for each software instance, only small parts of its code needs to be instrumented, which significantly reduces the individual runtime performance degradation in each copy of the software. In order to obtain the full coverage of the system, the individual subtasks' execution information are collected and sent back to the QA site, where they are combined in order to measure the overall monitoring coverage of the whole system. Additionally, the Gamma System permits on-site code modification/update of the software as a strategy to support instrumentation (insertion and removal of probes). This dynamic approach allows adjusting the monitoring to the needs of the QA team.

The Skoll system [6] provides a client-server architecture for testing highly configurable software under different hardware and software configurations. This approach allocates and distributes tests to end user sites, where they are executed, and provides the QA team with information on test failures, incompatibilities and their context. The allocation of resources is based on information provided by the end user about his or her client platform (e.g. operating system, compiler, hardware). Different software configurations are defined by the steering agent component of the Skoll platform. This agent sends jobs to the Skoll client, installed in a client host, based on the client hardware/software configuration. The software being tested is then downloaded to some of the remote sites, locally compiled and automatically tested, with the results sent back to the agent for further analysis.

The EDEM (Expectation Driven Event Monitoring) system [10] goal is to monitor the usage patterns of end-user interfaces. The monitoring is based on high-level specifications (called expectations), that defines normal usage patters of the software based on GUI events. EDEM detects expectation violations (unexpected usage patterns), generating reports to the QA team. Expectations can be dynamically deployed using agents (event sequence detectors that use event correlation techniques [11]). When an expectation is violated, the agent sends notifications to the QA team in the form of e-mail messages. Expectation agents can be defined and deployed at execution time, with the help of a special agent editor. Data collected can be further analyzed and visualized in the form of charts, or be used by developers to improve the usability of the application.

3. Approach

We build upon the techniques described in the previous section in order to provide distributed residual testing based on behavioral specifications.

 provides an overall depiction of the approach, which is described below.

3.1. Approach Overview

From the overall system *behavioral specification*, the developer selects a partial specification he is interested in monitoring. Such *behavioral specifications* can be expressed in different ways (e.g. state machines, linear temporal logic - LTL, UML collaboration and/or sequence diagrams). They represent high-level descriptions of the system behavior to be verified. In order to be monitored, however, this high-level specification needs to be *mapped* to concerns in the name space of the system execution (runtime execution events). The product of this *mapping* is a *monitoring configuration*, which, alongside the software, is deployed to the software instances running on the end users' machines. In our approach, this configuration can be dynamically changed, thereby supporting evolution of the monitoring configuration according to the developers' needs. As the program instances execute, their execution data is optionally filtered and then published to the notification service. This data is then *collected* and *consolidated* in the QA team site, to obtain the overall monitoring information of the software. This execution data can be *visualized* and *analyzed* with respect to the expected system behavior, either at run-time or off-line, using the data collected.

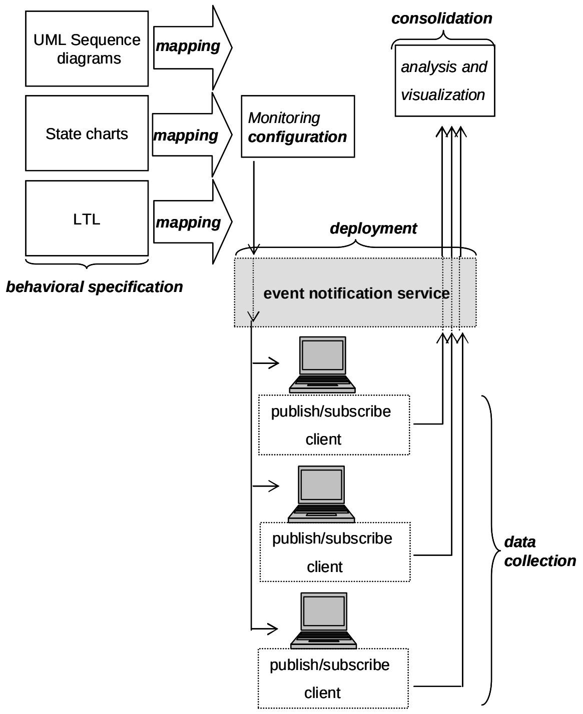

Figure 1 Overall description of the monitoring steps

Our approach differs from previous work in the following. The Gamma system collects coverage information as opposed to behavioral information. Additionally, in the Gamma system, subtask assignment and reassignment is based on the usage of the software instances, static and statistical analysis. We suggest that a more efficient task allocation and informative feedback can be obtained by allowing the developers to participate in this decision based on information such as end users' usage patterns, their machines' specific platforms and software configurations. The Skoll system uses user's resources to perform QA tasks, which is different from our approach, which monitors the actual system usage deployed to beta testers and developers. Finally, the EDEM system is limited to user interface events and cannot be used to monitor non-GUI programs. Thus, this solution only measures GUI-based interaction expectations, not being fit to monitor the overall system behavioral conformance.

3.2. Communication Infrastructure

Before introducing the design of the system in more detail, we must explain the publish/subscribe paradigm [12], which characteristics influence our design.

The components of our approach are integrated using the facilities of a publish/subscribe infrastructure. In a publish/subscribe style (also called implicit invocation), event producers publish information in the form of events (records, objects or attribute/value pairs), while consumers express interest on events using logical expressions called subscriptions. Subscriptions can include more advanced features such as event correlation expressions, allowing not only the filtering of information, but also the combination and abstraction of events into higher-level events[11]. For example, a sequence of events or a set of repetitive events can be combined and abstracted in higher-level events that express such a compound event. A notification server implements a distributed publish/subscribe paradigm, representing a logically centralized service that routes information from distributed publishers to the interested subscribers. This insulation between producers and consumers copes with dynamism and scalability of the system allowing publishers and subscribers to dynamically join and leave the distributed system. It also allows multi-cast communication style, which is important for group communication.

3.3. Detailed Description

This section describes, in more detail, the steps and main components of our monitoring infrastructure as generally explained in section 3.1. It also highlights the technologies used in each phase of our process.

3.3.1. Behavioral specification

Our approach requires the adoption of a behavioral specification (e.g. state machines, LTL, UML collaboration and/or sequence diagrams) to which the system execution must conform. For our current purposes, we have chosen to focus on object level residual testing, meaning that we are interested in gathering information about method invocations. This coarse-grained level of abstraction is used, rather than a fine-grained (such as statement execution) for the following reasons. First, deployed applications should be tested at a higher level of abstraction so as to avoid undue performance degradation. Second, this level of abstraction allows observation and analysis of software behavior at a level closer to software design, which facilitates our goal of providing dynamic support for verifying specified behavior. For the same reason, we use UML sequence diagrams as the system behavioral specifications.

3.3.2. Mapping

The mapping between high-level UML specifications into low-level monitoring configurations is performed by a parser component. A monitoring configuration describes where, in the deployed software, the probes need to be inserted, as well as which methods are to be monitored, together with specific *sequences* of methods to be detected. These last ones are expressed in the form of *subscriptions*. Since we are dealing with sequence diagrams, this mapping can associate one or more runtime method invocations with each object interaction in the UML description.

The adopted publish/subscribe infrastructure uses those subscriptions to detect the event sequences (either in the client site, using a special event correlation component, or in the notification server (as will be discussed in 3.3.4). Both monitoring configurations and subscriptions are deployed with software.

3.3.3. Deployment and Installation

In addition to defining which behavioral specifications should be verified, the developer must decide where to deploy the monitoring configuration. The criteria used for such decision is based on information such as end-users' software usage patterns or the site hardware and software configurations (similarly to the Skoll project [6]). This information is obtained by either automatic usage profile detection or by direct user inquiry.

Once deployed, the monitoring configuration must be installed in order to start collecting the necessary events, and identifying the required event sequences. The steps undertaken here are dependent on the technology adopted for instrumentation. For example, in the Gamma approach, this is done by dynamically replacing class files. Alternatively, using the Java debugger requires the configuration of the classes to listen to in the JDI. More details will be discussed in the implementation section.

3.3.4. Data collection and correlation

The data collection phase is concerned with the identification of sequence of events that need to be sent to the developers. These identified sequences might represent an expected or unexpected execution of the software instance with respect to the related behavioral specification. The identification of such sequences is performed by a software component that correlates events; the correlation is done by interpreting the provided subscription and detecting complete or incomplete event sequences. Those event sequences are identified individually and may be summarized before being published to the QA and developer teams. In this phase, consolidation consists of identifying repetitive sequences and summarizing this information (e.g. a given sequence was identified ***x*** times), this activity can be supported by the publish/subscribe client, by the event notification server, or by an application running at the QA and developer team's site.

It is worth noting that the mapping process generates subscriptions to the set of events involved in a UML sequence diagram. These events may be generated, as specified, in the proper order, or may come in unexpected orders. Unexpected sequences are indicative of different sorts of problems: (1) The sequence is actually part of the overall system specification, but the developer did not include it in the partial specification to be monitored; (2) The sequence is not part of the overall system specification, in which case either the specification is incomplete and the system is still working properly, or an erroneous behavior was actually identified. In either case, once an unexpected sequence of events is detected, the detailed sequence should be sent back to the developers for further analysis. Otherwise, when a valid, expected, sequence is detected, the occurrence of such sequence is published.

3.3.5. Consolidation, Analysis and Visualization

Since our approach uses the software tomography strategy (see section 2) to deploy monitoring configurations to different software instances in the client sites, this information needs to be further consolidated for the sake of analysis. In so doing, aggregated information about the effectiveness of the residual testing on multiply deployed systems yields more significant system usage profiles and measure of test adequacy than obtained during pre-deployment testing (during development) and also more than could be obtained from a single deployment. This consolidation and its respective interpretation provide feedback to the developers about system usage, as well as an insight into adjustments that can be made to the monitoring configuration. Consolidation is done according to the developer's focus of interest, for example, one can consider data gathered from all executions on all software instances but for a specific usage profile group (e.g. groups that extensively use a specific kind of feature such as GUI, security, or database).

For the visualization, we use gauges, graphics and visualizations to present the results obtained from this consolidation. This can be performed, at runtime, as the data is being received and consolidated or afterwards, using local stored execution tracing information.

3.4. Implementation Issues

The whole infrastructure is being implemented in Java due to its run-time characteristics, which facilitates the dynamic change of the monitoring configuration through different approaches currently being studied.

The first step of our approach is the *mapping* from UML sequence diagrams to monitoring configurations, which contain the sequence of methods to be monitored in the software instances. Currently, we are studying the use of off-the-shelf software development tools, such as Rational Rose, which perform forward engineering. In this case, the methods in the source-code generated from the sequence diagram will be the same methods to be monitored. Since we are using an in-house event-notification server, YANCESS [13], a tool will be developed to translate sequence diagrams to subscriptions.

In our previous work [5], we based our *data collection* solution on the Java Debug Interface (JDI) of the Java Platform Debugger Architecture (JPDA). The JDI is a high-level Java API that supports event collection produced by Java virtual machines without requiring source code instrumentation. Nevertheless, due to a granularity limitation, it is not possible to configure the events to be gathered on a method-by-method basis. Thus, since we are going to monitor events from a UML sequence diagram and taking into consideration performance impacts, this approach is not the most indicated.

We are studying the application of a different implementation strategy, based on a modified JVM class loader, which is part of the Vavoom project [14]. The class loader controls the object activation and destruction, and intercepts method invocations and exception handlings. With such a strategy, the loader can be configured to filter specific method signatures, publishing events representing specific invocations in the program. This approach has the advantage of allowing java programs to be monitored without the need to change byte code. It also allows the dynamic change of the monitoring configurations by directly changing the loader policies. We believe that, by using this modified class loader, we can minimize performance degradation in software instances.

The *consolidation* phase is supported by the event correlation features provided by the event notification server YANCEES [13]. It was chosen for its extensibility and configurability. The server can be configured with client and server-side plug-ins to process events. In particular, event correlation plug-ins are available, which allow event sequence detection and abstraction in both client and server sides.

Finally, for *visualization and analysis* tasks we are studying the use of execution time visualization tools such as the one proposed by [15] which supports UML sequence diagrams visualization or off-line analysis tools such as used by EDEM [10], which summarizes the data in form of graphics and histograms.

4. Conclusions and Future Work

This paper presents an approach which extends current residual testing techniques and tools such as expectation-driven monitoring, software tomography and dynamic program instrumentation, and publish/subscribe systems, to provide behavioral conformance checking. In our approach, deployed software is tested with respect to its conformance to dynamic high-level models such as UML sequence diagrams. The approach uses the facilities of a versatile publish/subscribe service that supports the data collection, sequence detection and filtering. The software tomography strategy is used to disseminate monitoring configurations to specific clients coping with scalability. End-user usage profiles and hardware and software configurations are collected and used to support the software tomography task distribution. Expectation-driven monitoring is used to monitor execution patterns in the deployed software. Dynamic program instrumentation strategies are used to provide flexibility to the approach, allowing the QA team to refine and deploy new tasks to the programs in the software on a client site.

We are currently completing the implementation and integration of the different components of the system (notification server, the Vavoom JVM monitor, and the UML sequence diagram specification parser).

As future work, we intend to validate our approach on applications with real users. We also plan to experiment with other behavioral specification models (e.g. state machines, LTL), as well as to experiment with residual testing in a coarser-grained level of abstraction (e.g. components).

5. References

1. Harrold, M.J. *Testing: a roadmap*. in *22nd International Conference on on Software Engineering, Future of Software Engineering Track*. 2000. Limerick, Ireland.
2. Pavlopulou, C. and M.Young. *Residual Test Coverage Monitoring*. in *21st international conference on Software engineering*. 1999. Los Angeles, California, United States: IEEE Computer Society Press.
3. Liblit, B., et al. *Sampling User Executions for Bug Isolation*. in *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS'02)*. 2003. Portland, Oregon.
4. Orso, A., et al. *Gamma System: Continuous Evolution of Software after Deployment*. in *ACM International Symposium on Software Testing and Analysis (ISSTA'02)*. 2002. Rome, Italy: ACM.
5. Naslavsky, L., M. Dias, and Richardson D. *Multiply-Deployed Residual Testing at the Object Level*. in *IASTED International Conference on Software Engineering (SE2004)*. 2004. Innsbruck, Austria.
6. Yilmaz, C., A. Porter, and D.C. Schmidt. *Distributed Continuous Quality Assurance: The Skoll Project*. in *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS'02)*. 2003. Portland, Oregon.
7. Richardson, D.J., S.L. Aha, and T.O. O'Malley. *Specification-Based Test Oracles for Reactive Systems*. in *14th International Conference on Software Engineering*. 1992. Melbourne, Australia: IEEE Computer Society.
8. Baresi, L. and M. Young, *Test Oracles*. 2001, University of Oregon, Dep. of Computer and Information Science.
9. Bowring, J., A. Orso, and M.J.H. Proc. *Monitoring Deployed Software Using Software Tomography*. in *Workshop on Program Analysis for Software Tools and Engineering (PASTE'02)*. 2002. Charleston, SC, USA.
10. Hilbert, D. and D. Redmiles. *An Approach to Large-scale Collection of Application Usage Data over the Internet*. in *20th International Conference on Software Engineering*. 1998. Kyoto, Japan: IEEE Computer Society Press.
11. Liu, G., A.K. Mok, and E. Yang. *Composite Events for Network Event Correlation*. in *IFIP Symposium on Integrated Management (IM'99)*. 1999. Boston, MA: IFIP.
12. Eugster, P.T., et al., *The many faces of publish/subscribe*. ACM Computing Surveys, 2003. **35**(2): p. 114 - 131.
13. Silva-Filho, R.S., C.R.B.d. Souza, and D.F. Redmiles. *The Design of a Configurable, Extensible and Dynamic Notification Service*. in *Second International Workshop on Distributed Event-Based Systems In conjunction with The ACM SIGMOD/PODS Conference*. 2003. San Diego, CA.
14. Dourish, P. and J. Byttner. *A Visual Virtual Machine for Java Programs: Exploration and Early Experiences*. in *ICDMS Workshop on Visual Computing*. 2002.
15. Gogolla, M. and M. Richters. *Aspect-Oriented Monitoring of UML and OCL Constraints*. in *4th AOSD Modeling With UML Workshop*. 2003. San Francisco, CA, USA.