Self-Test Components for Highly Reconfigurable Systems

Giovanni Denaro, Leonardo Mariani, Mauro Pezzè

Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Via Bicocca degli Arcimboldi, 8
I-20126 - Milano, Italy
{denaro, mariani, pezze}@disco.unimib.it

Abstract

Verification of component-based systems presents new challenges not yet completely addressed by existing testing techniques. This paper proposes a new approach for automatically testing highly reconfigurable component-based systems, i.e., systems that can be obtained by changing some components. The paper presents an industrial case that motivates our research and proposes a testing infrastructure that tracks run-time information for components. The collected information is used for automatic testing new versions of existing components and new configurations of existing systems.

1 Introduction

Component-based software engineering is increasingly used in several application domains. In the most general settings, component-based systems are made of several heterogeneous hardware and software components, independently developed either on-site or by third parties. Both the components and the component-based systems can be available in several different versions. Moreover, the same system may also be available in several configurations made of different sets of components.

Examples of such systems are the board systems of new generation cars and industrial test devices. Car board systems incorporate several hardware and software components, e.g., DVD, GSM, GPS devices, simple operating

[1] This work has been partially founded by the Italian Government in the context of the QUACK project (QUACK: A Platform for the Quality of New Generation Integrated Embedded Systems.)

systems, browsers, and drivers to provide complex services, e.g., Internet facilities, car alarm monitoring and integrated control of the devices. Some of these components are produced by third parties. For example, the GSM, the DVD, and the GPS devices are generally not proprietary. Different car models use different configurations. For example, base models may not include GPS and Internet facilities. Industrial test devices are usually produced in small scale for testing different end products. The similarities of both the products to be tested and tests to be performed can be addressed with test devices that differ in some hardware or software components. Thus, it is good design practice to design industrial test devices starting for a library of available components.

While the development of components and component-based systems is well supported by many technologies, e.g., .NET [3], EJB [9], Corba [10], the verification and validation of components and component-based systems is not well supported yet, although these systems entail verification requirements that can be hardly satisfied by means of traditional testing and analysis techniques.

Traditional verification techniques assume that testers have complete knowledge of requirements and execution environment of the SUT (Software Under Test). However, this is not the case of component-based systems. Producers of components are generally not aware of the context in which the components may be used. Thus, testing components in isolation is harder than traditional unit testing and cannot be generally completed before the assembly of the final systems. Moreover, producers of component-based systems may not be granted access to all details of the components, which are often distributed without source code or with incomplete documentation. Thus, techniques for testing component-based systems must work even with only partial knowledge of components specifications and source code, maybe requiring different kind of information.

Rosemblum provides a first formalization of the notion of test adequacy for component-based systems [14]. Briefly, the input space of a component `C`, considered in the context of a component-based program `P`, can be divided in two subsets: `P`-relevant, i.e., the inputs of `C` reachable by means of inputs provided to `P`, and `P`-irrelevant, i.e., the inputs of `C` that will be never executed in the context of `P`. An adequate test of `C` in the context of `P` has to be adequately representative of the `P`-relevant subset. A test that may be adequate for `C` in isolation, may not be adequate in the context of a program `P`. The extreme case is represented by a test that is adequate for `C` in isolation, e.g., because guaranteed the required level of structural coverage, but exercises only elements from the `P`-irrelevant subset, and thus is not adequate for `P`.

Developers of component-based systems need to verify the compliance of the components with the system. The separate verification of components and component-based systems becomes particularly difficult when the components are available in many different versions and the systems are deployed in many different configurations.

This position paper presents an on-going research that investigates how to support integration testing for highly reconfigurable component-based systems. We present an industrial case that motivates our research and we outline the proposal of a testing infrastructure that tracks run-time information for components. The collected information is meant for automatic testing of new versions of components and new configurations of component-based systems.

2 Case Study

Here we illustrate the problem tackled in this paper by referring to a specific class of industrial systems: the industrial test devices, i.e., devices used for testing electro-mechanic appliances, e.g., washing machines, dish washers, kitchen robots. Industrial test devices assess the quality of the products and discard faulty products. Data gathered during testing are stored to perform both short-term analysis, and long-term statistical studies. Short-term analysis allows to identify faulty devices, while long-term statistics can help locating common defects and identifying empirical relations among faulty components. Properties to be verified for different sets of devices can be very different, ranging from interface to stress testing.

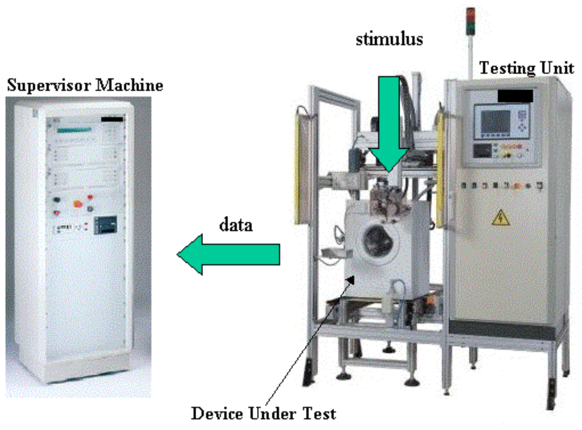

Fig. 1. An industrial test device

Industrial test devices include a hardware scaffolding of two main components: a testing and a supervisor machine, as shown in Figure 1. Scaffoldings include actuators and connectors, such as mechanical arms and wiring cables, that suitable exercise the Devices Under Test (DUT). The supervisor machine saves the data gathered during testing an individual DUT, evaluates the test results according to predefined pass/fail criteria, and then sends the collected information to a main server. The main server perform statistical analysis.

Different, although similar industrial products, e.g., different models of washing machines, are tested with specific test devices, usually produced in small series. Test devices for similar as well as different products may share sets of hardware and software components. Efficient development of industrial test devices relies on the availability of hardware and software components, both in-house produced or third-party developed, that can be suitably deployed and assembled to obtained different devices.

As shown in Figure 2, the required components can be classified at three different levels: hardware components, embedded software components, and application components. Hardware components represent the mechanical parts of the industrial test devices and include communication infrastructures, such as cables, wires, electronic circuits, electrical paths and wireless devices. Embedded software components are usually deployed on the hardware components in order to control their operations. Application components manage the underlying system in order to provide high-level features that directly address problems in the user domain.

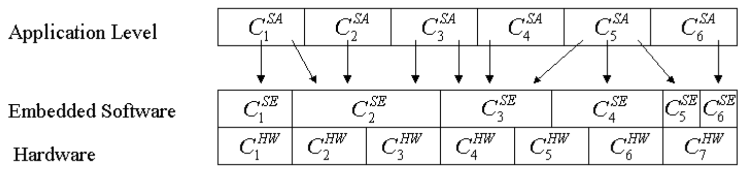

Fig. 2. Overall architecture of a component-based machine

In some cases, embedded software may be specific for a given hardware component, e.g, embedded software that manages sensors of a specific hardware in order to perform monitoring activities. In the general case, however, the relation between hardware components and embedded software components is a many-to-many relation, i.e., several embedded software components may be deployed into several hardware units with a non-trivial mapping. For example, the same embedded software may be deployed into several hardware components that provide similar behaviors, or the same hardware component may require a set of embedded software components for different operations, e.g., the camera manager consists of several embedded components (including driver, image processing utilities and compression utilities) deployed in a single camera device. The layering between hardware and embedded components in Figure 2 sketches the possible mappings.

In turn, the application components, based on classic component technologies (e.g., CORBA [10], EJB [9] and DCOM [4]) and communication paradigms (e.g., RPC [2] and RMI [11]), participate in relations with the embedded components. The arrows from the application to the embedded layer in Figure 2 describe different interactions, e.g., method invocations, messages and upcalls, either local or remote, synchronous or asynchronous.

The components indicated in the design drafted in Figure 2 are likely to be reused across different industrial test devices, because of the following properties of the domain:

Parameters to be monitored recur. Often a requirement takes the form of a set of parameters and properties that must be monitored or checked over time. Parameters and properties recur for different different versions of the same product, since advances in technology are not so quickly to continuously require monitoring of new parameters. Furthermore, there exist several standards for the certification of the quality (i.e. ISO9000, ISO9001), thus the same set of controls can be required for different appliances (of different vendors too). Recurrence of parameters to be monitored leads to reuse of components that deals with these parameters.

Test procedures recur. The same test procedure is often reused over several appliances of different vendors and different versions.

Component technology is well known in manufacturing. Component technology originates in engineering and manufacturing. Producers build their products buying components from third parties and optimize their productivity with reuse. This approach increases commonalities within the products of a given industrial sector. For example two washing machines build by different manufacturers often share several components. Commonalities of different DUTs are the nourishment for reuse.

Reuse of the interfaces. The reuse of the same (or similar) interface for different appliances is a particular case of reuse located in the DUT. Interfaces have particular relevance because the testing machine stimulates the DUT through the interfaces, so reuse of interfaces directly affects the possibility of reusing components of the testing machines.

In the context of industrial test devices, components of all types, from hardware- to application-level, are configurable and available in several versions. Consequently, the same unit may behave differently if configured in a different way. The configuration of components maybe used to setup the machine for a particular appliance model. When the appliance model changes, we must change the configuration of some components. New configurations of a test device may result in new failures and thus need to be tested. Techniques for optimizing the testing of new configurations of test devices can reduce the verification and validation costs.

This paper proposes a new technique for testing such kind of components, based on the record/playback of executions of components in different systems.

3 Incremental Verifications of Component-Based Systems

Highly reconfigurable systems are often updated by changing one or few components in an existing system, e.g., because of the availability of a new release of a component or the dynamic linkage of a new library or a change in the running environment of a mobile system.

When a new system is obtained by substituting an existing component with a new one, we need to verify the *behavioral equivalence* of the new component with respect to the old one, i.e., beside fitting the syntax of the target system, the new component must provide valid behavior for all inputs accepted by the old component (less stringent pre-conditions) and must assure at least the same results of the old component (more stringent post-conditions) [15].

Unfortunately, behavioral equivalence is very difficult to verify in practice. Formal approaches that prove compliance of pre- and post-conditions, face two main problems: They require formal specifications which are seldom available, and are often based on sophisticated theorem proving facilities, which may hinder the practical applicability of the approach. Necula's proof carrying code [12] may lead to an interesting solution, but has many drawbacks in the component world, as discussed later in this paper [2] . Regression testing may be impracticable: Even in case of fully automated regression testing: Substituting a single component would entail re-execution of the whole system (which may include a large amount of components) and the set up of complex test environments, that could include, for example, hardware devices, hardware simulators and distributed software elements.

3.1 Test Recording Component Companions

The main idea that underlies our research is that, for a given component, it is possible to capture the portion of functionality that is relevant to a particular user system, in terms of a meaningful number of executions of the component in the context of the component-based system. Once suitably recorded, such executions may be used to test for behavioral equivalence of a new version of the same component. In this approach, behavioral equivalence is tested at the user site, with the support provided by the component producers.

Our proposal can be summarized as follows. Each independently developed component is delivered along with a particular software item that can monitor and control the component executions. In the following, such items are referred to as the test-recording-component-companions (TRCCs).

When the component is executed in the context of a particular user system, e.g., during system and integration testing or also on-field during alpha and beta testing, the associated TRCC wraps the component and records execution data. In particular, TRCCs act as follows:

[2] Section 5 provides further details

- They monitor the access interface of the associated component for recording the input values when the component is invoked and the output results when it returns.
- They control the internal state of the associated component for recording state information on invocation and returning of the component.
- They record other execution information that can be useful for their own purposes.

To the end of testing behavior equivalence, when a component is substituted with a new version, the executions gathered in the associated TRCCs are re-executed on the new component and the new results are automatically compared with the ones of the previous component. Notice that, beside requiring that a new component version is compatible with the TRCC the old version (for allowing use of the recorded data), this does not require neither re-execution of other system components nor set up of the entire testing environment. Moreover, through TRCCs, the collection of information on component execution at the user site does not require availability of the component source code.

4 Technical issues

For implementing the TRCC approach, we are evaluating the possibility of using aspect-oriented technology [6] for providing a tool that is able to automatically derive TRCCs as separate implementation units for already developed components. TRCCs can be deployed based on few information about the interface and the internal state of the components.

In general, TRCCs can record a limited number of test cases, because of the need of trading-off precision of verification results and memory and time requirements. Suitable test selection policies must be configurable for TRCCs. The selection policy is responsible of making decisions on which test cases are to retain or discard on exceeding maximum limits.

Possible selection policies can be preliminary sketch as follows:

- Retain test cases that have been executed a high number of times or more recently;
- Retain test cases that increase or maximize structural coverage on the monitored component. This requires that producers release components suitably instrumented;
- Retain test cases that assure the best coverage of the relevant input space, e.g., by using similarity techniques to establish if a new test case is better than an old one for this purpose;
- Use a combination of the previous policies.

Finally, an important technical issue is that the presence of TRCCs in a system may interfere with the testing experiments by influencing the timing/synchronization of the wrapped components and, in general, the performance of the whole system. For this reason, TRCCs must be turned off when testing non-functional requirements, such as response time. This does not downgrade our approach because, in our experience, testing of functional and non-functional requirements are separate activities in industrial software processes. However, how to test equivalence of two versions of a component against non-functional properties is still an open problem.

5 Related Work

Behavior equivalence of components can be based on Necula's proof carrying code [12]. Proof carrying components should come with an attached proof of their correctness with respect to a safety policy (in this case, related to behavioral equivalence) published by the user system. Thus, when integrating a new component, the user system can exploit the attached proof and easily verify whether all pre-requisites are fulfilled. The proof carrying code approach presents advantages derived from being a formal verification approach, i.e., components are accepted only if they are formally proven to satisfy all requirements. However, the proof carrying code approach presents several problems: Safety policies are difficult to write and, in general, establishing their correctness requires further verification effort; Generation of safety proofs is difficult and cannot be fully automated; Incorporating safety proofs can increase the size of a component of a factor that, in principle, is exponential in the size of the stand-alone component. Moreover, safety policies have to be published by component users. This would entail that the component producers knew in advance all possible users, which is clearly impossible in component-oriented software market.

The restrospector approach of Liu and Richardson [7] uses retro-components (components with retrospectors), which are able to record test information on the producer site and make it available in the user environments, aiming at taking advantage of the results of testing of components in isolation. In our TRCC approach, information is collected directly at the user site, aiming at solving the problem of increasing the adequacy of component testing in the context of specific user systems, which producers are not aware of.

Self testing components are explicitly addressed by Martins et al. with specific reference to object oriented classes implemented in `C++` [8]. The authors, which in turn refer to works by Binder [1] and Hoffman [5] consider that, in addition to an implementation, a self-testable class contains built-in test capabilities (for accessing the state and monitoring the execution) and a test specification from which test cases and test oracles can be derived. However their self-testing components do not possess any test recording capability.

Finally, the idea of taking advantage from information gathered during the operational use of a software system has relations with the research on perpetual testing. For example, Pavlopoulousee and Young propose to maintain part of the instrumentation in the software for measuring increments in the statement coverage during the beta testing phase [13].

6 Conclusions

Industrial problems are often addressed with highly-reconfigurable systems, i.e., systems distributed in different versions and configurations made of different subsets of components. The flexibility introduced with this technology allows for producing complex systems at low cost, that can be distributed even in small quantities. This paper presented an interesting application domain where such technology is rapidly spreading.

The quality of systems is not directly related to the amount of shared components, and the quality of single components is not directly related to the quality of the component-based system. Even a system obtained from an existing system by replacing only few components with new versions of the same components, may present new problems not revealed in the former system, independently from the quality of the new components. This paper suggests a technique for automatically testing such kind of systems based on the record and replay of execution information.

We are currently developing and experimenting the proposed technique in the context of the QUACK (A Platform for the Quality of New Generation Integrated Embedded Systems) project [3].

References

[1] Binder, R., *Design for testability in object-oriented systems*, Communications of the ACM **37** (1994), pp. 87–101.

[2] Birrel, A. D. and B. J. Nelson, *Implementing remote procedure calls*, ACM Transactions on Computer Systems **2** (1984), pp. 39–59.

[3] ECMA, *Common language infrastructure (CLI) partition I: Concepts and architecture*, Final draft, Published by ECMA TC39/TG3 (2002).

[4] Eddon, G. and H. Eddon, "Inside Distributed COM," Microsoft Press, Redmond, WA, 1998.

[5] Hoffman, D., *Hardware testing and software ics*, in: *Proceedings of the Northwest Software Quality Conference* (2001), pp. 234–244.

[6] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, *Aspect-oriented programming*, in: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (1997).

[3] QUACK is partially supported by MIUR, the Italian Ministry of University and Education through the Cofin Program

[7] Liu, C. and D. Richardson, *Software components with retrospectors*, in: *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, 1998.

[8] Martins, E., C. Toyota and R. Yanagawa, *Constructing self-testable software components*, in: *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)* (2001), pp. 151–160.

[9] Matena, V. and M. Hapner, *Enterprise javabeans^{TM} specification*, Public Draft version 1.1, Sun Microsystems (1999).

[10] Merle, P., *Corba 3.0 new components chapters*, TC Document ptc/2001-11-03, Object Management Group (2001).

[11] Microsystems, S., *Java^{TM} remote method invocation specification*, Technical report, Sun Microsystems (2002).

[12] Necula, G. and P. Lee, *Proof-carrying code*, Technical report, Technical Report CMU-CS-96-165, Canergie Mellon University (1996).

[13] Pavlopoulou, C. and M. Young, *Residual test coverage monitoring*, in: *Proceedings of the 21th International Conference on Software Engineering (ICSE'99)* (1999), pp. 277–284.

[14] Rosenblum, D., *Challenges in exploiting architectural models for software testing*, in: *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, 1998.

[15] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," ACM Press and Addison-Wesley, New York, NY, 1998.