Generation of Self-Testing Components

Leonardo Mariani, Mauro Pezzè, and David Willmor

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

Abstract. Internet software tightly integrates classic computation with communication software. Heterogeneity and complexity can be tackled with a component-based approach, where components are developed by application experts and integrated by domain experts. Component-based systems cannot be tested with classic approaches but present new problems. Current techniques for integration testing are based upon the component developer providing test specifications or suites with their components. However, components are increasingly being used in ways not envisioned by their developer, thus making their test specification and suites invalid. In this paper, we propose an approach for implementing self-testing components, which allow integration test specifications and suites to be developed by observing both the behavior of the component and of the entire system.

1 Introduction

Software for the Internet often presents a strong integration of computation, data and communication aspects. Classic software products were often deployed and tested independently from communication services that were accessed as external libraries. In many Internet applications, such as e-commerce software, communication aspects cannot be easily separated from traditional software, but must be tightly integrated in the product. Such integration of heterogeneous aspects results in new development and testing challenges.

A popular approach for addressing complexity and heterogeneity relies on the use of components and component-based design methodologies. Components enhance reuse and can facilitate the sound integration of heterogeneous services. Components can be developed by different teams with specific expertise and abilities: communication experts may focus on communication services, while software and integration experts may focus on traditional data and computational aspects.

The independent development of reusable components and the adoption of component-based methodologies introduce new verification challenges that derive from the absence of knowledge about the final system when developing

components, and about the components' internals when developing the target application. Good components are widely re-used, and component designers cannot anticipate all possible uses at development time. System designers should be able to reuse components without knowing all internals to benefit from component developers' expertise. Test designers must be able to test single components independently from the applications, and component-based systems without accessing the internals of their components.

The scientific community is investigating various solutions to these new verification challenges: providing components with an associated specification [1], deploying together the component with its test suite [2], or developing components with testing facilities [3].

The approaches investigated so far work under specific hypotheses on both components and their integration, and thus inevitably restrict the use of components and may fail when the verification hypotheses are violated. In this paper, we propose a novel approach that tries to overcome these limitations by moving verification from development to deployment time. The proposed approach is based on self-testing components, a method successfully exploited in hardware design: components are augmented with self testing capabilities that can be exploited when the components are reused in a novel system. Self-testing components can self-verify their behavior in the new context and thus in principle can be reused without any a-priori limitation.

We propose a framework that generates self-testing features from components' test and execution. When testing and using components in traditional settings, we capture components' executions, and distill invariants that model the components' behavior as experienced during execution. These invariants provide the information for generating test cases that can be automatically executed when components are reused in new systems. Thus new systems can be tested without restricting components' reuse or requiring specific knowledge about components.

Section 2 presents basic idea underlying self-testing components which are mostly inherited from hardware testing. Section 3 describes the approach for deriving invariants that we use for generating test suites associated to components. Section 4 presents the different stages where generation of self-testing components can be performed. Section 5 presents the different techniques that we developed for generating test cases for self-testing. Section 6 discusses the problems of execution self-tests. Section 7 highlights the advantages of the new approach presented in this paper comparing it with related work. Finally, Section 8 outlines on-going and future work seeded by the technique proposed in this paper.

2 Self-Testing Components

Components are usually produced by different software vendors and are then assembled by component deployers, to build the final system. Component developers know neither the context of the components execution nor the ways

in which they will be used. Moreover, component developers implement components under implicit assumptions regarding the behavior of other components that may be violated in certain environments.

Component integrators have limited knowledge of the reused components, since components are often deployed either without or with incomplete specifications and the source code is seldom provided. Integration testing becomes difficult, and does not always provide a sufficient level of confidence in the final system. Many major failures confirm the difficulties in achieving an acceptable level of confidence even in critical scenarios, see the Mars Climate Observer [4], the Mars Polar Lander [5] or the Arianne 5 [6] problems.

In hardware testing, several of these problems have been solved by developing Built-In Self-Test (BIST) features, i.e., "BIST is a Design-for-Testability technique in which testing (test generation and test application) is accomplished through built-in hardware features" [7]. BIST features are automatically derived from the design, and system integrators do not have to care about component testing since it is automatically performed by BIST features. The same idea can be extended to software components by producing *self-testing components*, i.e., components which automatically test their integration in a larger system.

Self-testing software components differ from BIST hardware for: the time of test execution, and the type of tests that are performed. In the case of testing hardware components, test cases are executed regularly to check for faults that may arise at any point because of electromagnetic problems, while tests for software components are executed *only once at deployment time* since software components do not change during their lifetime (unless they are replaced with newer versions). Moreover, since hardware can degrade its performance or stop working correctly, BIST embedded features refer mainly to unit testing. Whilst, in the case of software components, it is more important to consider *integration tests* since unit testing is performed during development.

There are several ways to implement self-testing components: by embedding specifications [8], by adding testing functionalities into the component [3], and by associating test suites to components [9]. Recent work suggests various approaches and provides encouraging preliminary results. Our work enhances the previous solutions proposing a novel approach for generating test cases to create self-testing components.

Self-testing components are provided with a set of associated test cases that are executed at system deployment time. The successful execution of all test cases associated with a component C newly integrated in a system assures that C integrates correctly into the system, i.e., the interactions from C to the system are correct. The successful execution of all test cases associated with components that interact with C assures that the system integrates well with C, i.e., all interactions from the system to C are correct. Insights regarding the execution of test cases are provided in Section 6.

Self-testing components present several advantages:

- System integrators who operate with classic components need to design integration test strategies without knowing details of each single component; while with self-testing components integration tests are obtained automatically.
- Self-testing components do not require generating test cases for each new system.
- Self-tests can be automatically re-executed for testing the correctness of modifications without additional effort.
- Self-testing components can be augmented with oracles derived from executions in previous systems, thus reducing there need for additional scaffolding.

Test suites associated with self-testing components can be derived in different ways:

- The test suite is *manually provided by the component developer* [10]. In this case the test suite is generated by using the experience and intuition of the component developer without using any explicit criterion. This brute force approach takes advantage of the ability of the developer, but rarely meets the requirements of soundness and completeness.
- The test suite is *generated from (formal) specifications* [11]. In this case a specification of the software component is provided from which the test suite is generated. Whilst this method produces test suites which are highly effective it is reliant upon the specification existing.
- The test suite is *generated from component executions* [12]. In this case the test suite is obtained by observing previous executions, selecting meaningful ones, and then storing them in a repository. This case is dependent upon the component being used completely during observation. Therefore the longer the component is observed the more likely it is for coverage to be obtained.

Our approach to creating self-testing components is based upon an amalgamation and extension of the second and third methods. A specification, in the form of *derived/inferred* properties, is generated from either source code or observed executions. This specification can then be used to *filter* observed executions to be used as test cases. Our proposed method has the following benefits over the previous methods: the technique (1) can be used on binary components, e.g., when a third party component is purchased, (2) is very practical, thus it requires little effort by either the component developer or the component assembler, and (3) is for the large part automatic.

3 Deriving Properties of Component Interaction

The test suite associated with each component stimulates interactions with other components. Good selection criteria must select executions that stimulate interesting interactions, i.e., executions that represent all possible behaviors. We select executions by referring to interaction and I/O invariants.

Interaction invariants are regular expressions associated to component services. An interaction invariant summarizes all possible sequences of interactions that can take place when the service is executed. Requests to other services are represented with suitable labels. For instance, the regular expression associated to a service `viewCart` of a `Cart` component for webshop applications can be

`(Catalog.getItemDetail(ImageUtility.loadImage` $+\epsilon$`))*`

that indicates that visualization of the content of the `Cart` causes from 0 to n interactions with the `Catalog` component for retrieving detailed information about items. For each item in the cart, an additional interaction with the `ImageUtility` component may be required if a picture of the item is provided.

I/O invariants are properties over parameters implied in interactions of the target component with other components of the system. For example an invariant $qt > 0$ associated to the interaction of a service `addItem` of a component `Cart` by a service `buyItem` of a component `Purchase` indicates that variable quantity (qt) is always positive when the service is invoked in that context.

Interaction invariants can be derived both by statically analyzing the source code and by observing executions. I/O invariants can be derived only at run-time since they refer to other components of the system. In previous work, we experimented with automatic generation of both I/O and interaction invariants from run-time executions [13]. Inference of invariants from business components requires the capability both to intercept interactions on a running system and extract state data from complex objects. These issues have been addressed in detail in [14]. We report only an example to show how state information is extracted from complex objects. Let us consider for instance object `item`, and let us refer to its state elements as `item.getName`, `item.getQuantity`, etc. . . , where the labels are obtained by composing the object names with the methods names. We statically generate interaction invariants by building a reduced control-flow graph where nodes represent service requests.

Interaction invariants and I/O invariants represent two complementary sources of information: interaction invariants represent interaction properties while I/O invariants represent properties of data exchanged by components during execution. Filtering executions according to interaction and I/O invariants ensures a high degree of coverage of possible behaviors involving multiple components, and thus a high possibility of discovering system faults.

4 Scenarios

Construction of the test suite can happen at a number of stages within a components life. In this section, we outline three such stages: Development Time, Run-time, and Post Execution.

Development Time
: Constructing the test suite at development time allows the source code to be used to derive invariants. Information derived from source code represents

all possible interactions with other components. Thus, can be considered to be *complete* interaction invariants, as no other interactions are possible. Therefore, a test suite which completely covers these invariants would test all of the components possible interactions. It is not possible to derive I/O invariants from source code, thus, filtering criteria based on these invariants cannot be used to create test suites during this stage.

Run-time
: Constructing the test suite at run-time involves observing the component within its parent system in order to derive information on its behavior. This form of component monitoring produces invariants which are *evolving* in that they change as new behaviors are observed. New invariants incrementally approximate all possible behaviors. Thus, the filtering criteria can utilize the set of invariants as they are evolving allowing test suites to incrementally evolve as new behaviors are observed. A test suite created at this stage will be based upon observed behaviors and may not cover all possible behaviors of a component.

Post Execution
: Constructing the test suite post execution allows the observed behaviors of the component to be utilized in the derivation of invariants. Both interaction and I/O invariants may be derived in this manner. In this context post execution means that the system has been observed for a period of time and the observed behaviors are then analyzed off-line. Invariants derived in this manner represent the *observed* behavior of the system, similar to invariants generated at development time, the invariants remain static. Thus, the filtering criteria can utilize the derived invariants to select useful test cases from the set of observed behaviors.

5 Filtering Criteria

Test cases are constructed by filtering observed behaviors according to some form of *Filtering Criteria*. In this section, we discuss possible filtering criteria, which can be categorized as those based upon Interaction Invariants and those based upon I/O Invariants.

Interaction Invariants

Interaction invariants are expressed as regular expressions or equivalent finite state automata. Interaction invariant testing can be performed on either representation, each providing various characteristics that can be utilized in the creation of filtering criteria.

Regular expressions are defined over an alphabet. For example the regular expression $a^*b(b+c)$ is defined over the alphabet $\{a, b, c\}$. A filtering criterion can select test cases to cover all symbols, we call this ***alphabet coverage***. A test suite according to this criteria for the previous example would be (abb, bc).

Regular expressions combine symbols of the alphabet with operators: . specifies that any symbol can be matched, + is the union operator which specifies that either the left or right symbol can be matched, $*$ is the Kleene star which specifies that the previous symbol can be matched 0 or more times. A filtering criterion can be specified to select test cases to cover the whole alphabet and all operators, we call this ***operator coverage***. To cover the behavior of the Kleene star, test cases should be selected to cover 0, 1 and $n > 0$ occurrences of the proceeding symbol, and to cover the behavior of the union operator + test cases should be selected to cover both the left and right symbol of the operator; for example, a test suite would be $(bb, bc, abb, aabb)$.

Neither alphabet nor operator coverage will produce test suites that cover all possible combination of behaviors. A filtering criterion that would accomplish this must include tests for all possible expressions, i.e., all combinations of the alphabet, we call this ***expression coverage***. A test suite that fully covers the regular expression $a^*b(b + c)$ would be $(bb, bc, abb, abc, aabb, aabc)$.

When filtering test cases for a finite state automaton the simplest criterion would be to select test cases for each possible state. This would cover all nodes of the graph, we call this ***node coverage***. An extension would consider state transitions, i.e. which other states can be accessed from a certain state. A second filtering criterion can select test cases to cover all state transitions, that is all edges of the graph, we call this ***edge coverage***. Furthermore, when selecting test cases the graph can be considered in its entirety. Thus a filtering criterion can select test cases to cover all paths through the graph, we call this ***path coverage***.

Interaction Invariants may be derived by observing the behavior of the system. When observation occurs in real-time the invariants change as new behaviors are observed. Thus, a filtering criterion could select test cases that modified an interaction invariant, we call this ***interaction invariant modification***.

Interaction Invariant Modification does not necessarily cover the behavior of operators such as the Kleene star. As mentioned previously coverage is ensured by selecting test cases for 0, 1 and n. However, a Kleene star may be derived from observing frequencies 0, 2, and 4 thus not ensuring coverage. To improve the coverage of interaction invariant modification operators are annotated with the observed frequencies: if a new frequency occurs then this test case is deemed to have modified an interaction invariant, we call this ***interaction invariant modification with observed frequencies***.

I/O Invariants

Domain Values represent the possible values of a components service parameters, a filtering criterion can select test cases to cover possible domain values, we call this ***domain value coverage***. However, the specific filtering policy used to select test cases with respect to derived invariants is independent from the method for filtering test cases. For example, component C may have an I/O

invariant $1 \leq x \leq 10$, therefore a filtering policy based on domain values would select test cases which cover this range according to a specified criteria. The following set of criterion that we have identified with respect to the inferred invariants is inspired from the original idea of domain-based testing proposed in [15][1]:

- Invariants over any variable
  1. $x \in \{a, b, c\}$
     `3 test cases:` $x = a$, $x = b$, $x = c$
- Invariants over a single numeric variable:
  1. $a \leq x \leq b$
     `5 test cases:` $x = a$, $x = a + 1$, $a + 1 < x < b - 1$, $x = b - 1$, $x = b$,
  2. $x \neq 0$
     `4 test cases:` $x < -1$, $x = -1$, $x = 1$, $x > 1$,
  3. $x \not\equiv a (mod\ b)$
     `4 test cases:` $x < a{-}1 (mod\ b)$, $x = a{-}1 (mod\ b)$, $x = a{+}1 (mod\ b)$, $x > a + 1 (mod\ b)$
- Invariants over two numeric variables:
  1. $x < y$
     `2 test cases:` $y = x + 1$, $y > x + 1$,
  2. $x \leq y$
     `3 test cases:` $y = x$, $y = x + 1$, $y > x + 1$,
  3. $x > y$
     `2 test cases:` $x = y + 1$, $x > y + 1$,
  4. $x \geq y$
     `3 test cases:` $x = y$, $x = y + 1$, $x > y + 1$,
  5. $x \neq y$
     `4 test cases:` $y = x + 1$, $x = y + 1$, $x < y$, $y > x$,
  6. any invariant over $x + y$
     `3 test specs for each computed test spec where:` $x = 0$, $y = 0$, $x, y \neq 0$,
  7. any invariant over $x - y$
     `3 test specs for each computed test spec where:` $x = 0$, $y = 0$, $x, y \neq 0$
- Invariants over two sequence variables:
  1. $x < y$
     `n test cases: test specs for the single element applied to all elements`,
  2. $x \leq y$
     `n test cases: test specs for the single element applied to all elements`,
  3. $x > y$
     `n test cases: test specs for the single element applied to all elements`,

[1] (situations where no test cases exist have been omitted for clarity)

4. $x \geq y$
   `n test cases: test specs for the single element applied to all elements`,
5. $x \neq y$
   `n test cases: test specs for the single element applied to all elements`,
6. x subsequence of y, or vice versa
   `3 test cases: subsequence at the beginning, middle, and end`,

– Invariants over a sequence and a numeric variable:

1. $i \in s$
   `3 test cases:` i `at the beginning, middle, and end of` s

The presented list of operators covers all possible I/O invariants, since they are computed by the Daikon engine [16] which is an engine able to infer mathematical invariants from execution samples. The range of operators that can be inferred is fixed and coincide with those reported in the list above.

I/O Invariants are derived from observing the behavior of the system. When observation occurs in real-time the invariants change as new behaviors are observed. Harder et al proposed a method for selecting test cases that modify an I/O invariant of a service [17]. A filtering criterion could select test cases that modified an I/O invariant, we call this ***I/O invariant modification***. This captures the case where component C is using another component in an unseen way. For example, instead of using a public method to modify a component, an attribute is accessed directly.

Relations among Filtering Criteria

The filtering criteria that have been identified do not produce test suites which are mutually exclusive. We can sort them with respect to the following criterion:

A filtering criterion C_1 *subsumes* a filtering criterion C_2 if every test suite that satisfies C_1 also satisfies C_2.

Figure 1 summarizes the relationships among the filtering criteria presented in this paper [2].

[2] whilst most subsumption relations can be observed from the criterions properties in the case of interaction invariant modification subsuming symbol coverage the relation exists due to the way in which invariant modification is defined in [13]. In this definition a new edge is added only if it has been observed thus covering all modifications implies that all edges are covered. Edge coverage does not subsume interaction invariants modification since covering all edges does not imply covering all modifications.

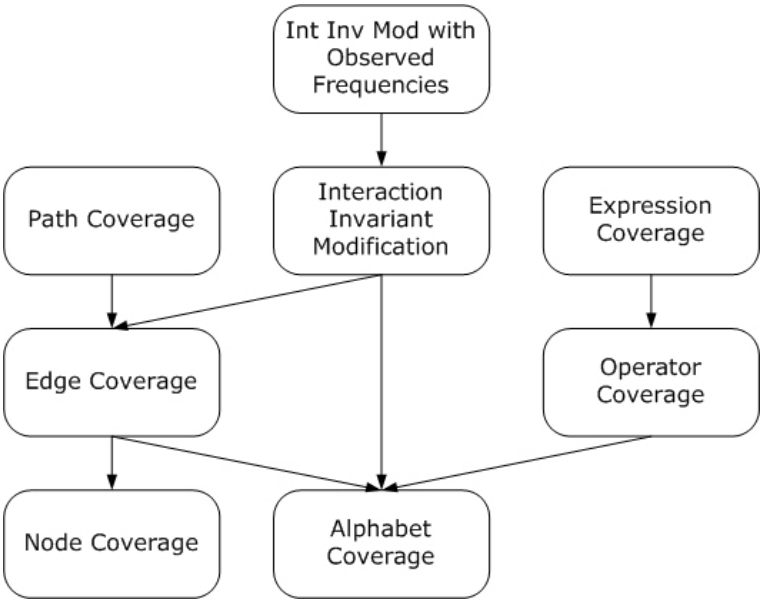

Fig. 1. Subsumption and equality relationships over the proposed filtering criteria

Applying Filtering Criteria to Scenarios

Section 4 identifies three points in a components life that test suites can be constructed. The filtering criteria mentioned previously may only be applicable to certain scenarios. For example, interaction invariant modification is only applicable when the invariants are changing, i.e., during run-time analysis. The applicability of filtering criteria to scenarios is summarized in Table 1.

	Development Time	**Run-Time**	**Post Execution**
Alphabet Coverage	YES	NO	YES
Operator Coverage	YES	NO	YES
Expression Coverage	YES	NO	YES
Interaction Invariant Modification	NO	YES	NO
Int Inv Mod with Observed Frequencies	NO	YES	NO
Node Coverage	YES	NO	YES
Edge Coverage	YES	NO	YES
Path Coverage	YES	NO	YES
Domain Values	NO	NO	YES
I/O Invariant Modification	NO	YES	NO

Table 1. Applicability of Filtering Criteria to Scenarios

6 Executing the Test Suite

Integration testing focuses upon how components interact within a component-based system, thus, two forms of interactions can be observed: how a component uses the system and how the system uses a component. Integration errors can occur in either or both situations. To test the first form of interaction, it is necessary to use a test suite covering component interaction invariants. To test the second form of interaction, we must execute the test suites of all components that interact with the component.

Components can be both stateless and stateful. Test cases are extracted from observed behaviors of the components and so are valid at that moment in time. If however, between extraction and execution, the state of the component has changed, the test case may not necessarily be compatible. Moreover, even if the state of the component has remained, the state of the system with which it interacts may have changed. Furthermore, if a test case was to fail, how does the tester know if it was caused by an integration error or a state error?

One possible solution to distinguishing the type of errors is to prevent test cases from being executed if the current state is not applicable. This can be accomplished by packaging test cases with conditions that specify what state the component and system must be in. Another solution to the problem of state is to design the components for testability, by adding an interface that would facilitate the setting or resumption of a previously occurred state.

Combining the existence of a repository of resumable states and query interfaces leads to the production of an effective framework for managing stateful components. The framework can be implemented with limited efforts on top of existing middleware that supports component persistency.

For instance, Enterprise Java Bean components [18] can be developed either with bean-managed or container-managed persistency. In the former case, the component already provides an interface for resuming previously stored instances. For example, a `Banking` component can provide a `findBigAccount(int minimum)` service to retrieve stored instances with an account value greater than `minimum`. In the latter case, requests for specific particular instances can be specified by the EJB-Object Query Language [18]. For example, you can specify a query to retrieve a specific stored object representing an account: `SELECT OBJECT(a) FROM Account AS a WHERE a.accountID IS NOT NULL`.

Currently the database which stores the persistent instances deletes them when the component is destroyed. By extending the database to implement warehouse functionality, a record of previously observed states would be available to the tester, allowing for component states to be resumed automatically.

Test cases are filtered from observed behaviors which can be seen as a constant stream of possible test cases. Test cases may be dependent on those that occurred previously, both within the same component and other components of

the system, thus we need a mechanism to capture and store these relations.

Capturing executions is complicated due to the representations not being suitable for persistent storage and the burden on system resources required to store them. The first problem can be addressed through serialization, e.g. within Java objects transferred through the network can be captured in a form easily stored. The second problem can be acceptable as the capture of executions, their filtering into test cases, and the testing of the components is largely automatic, is only performed when a component is deployed and when the system is off-line.

7 Related Work

The aim of the software engineering community is the conception of methods to produce perfect software. Therefore, there has been a significant amount of research on methods to test software generally, object-oriented systems in particular, and concepts tailored to component-based testing are being introduced.

Vitharana [19] summarizing the problems facing the testing of component-based systems highlighting how even though individual components are tested in isolation, it is very difficult to rely upon this, and complex integration testing is necessary to ensure the reliability of the system.

Binder [20] introduces the concept of building testability directly into the objects of an object-oriented system during their development, thus, reducing testing costs and improving the quality of the tests. This is achieved by a combination of approaches including formalized design and testing strategies and embedded test specifications (t-spec). However, whilst this method is possible when the entire system is being developed within a single group, in component-based systems, it would require all of these approaches being standardized and adhered to by a number of different developers.

Both Bertolino and Polini [2] and Martins *et al* [1] discuss different methods for making components self-testable. Their methods differ in that Bertolino and Polini package test cases with their components whilst Martins *et al* require the user to generate the test cases. However, in both cases the test cases are generated from a developer defined test specification that is packaged with the component. We propose that components may be deployed in ways not previously envisioned by the developer. Therefore, pre-defined test specifications may no longer be valid and it may be necessary to construct the test suite based upon the observed behaviors of the target system.

Liu and Richardson [9] introduce the concept of using software components with retrospectors. Retrospectors record the testing and execution history of a component. This information can then be made available to the software tester for the construction of their test plan. Liu and Richardson retrospectors do not take into account automatic generation and execution of test cases since test case generation is entirely delegated to the system integrator. Moreover, the problem of dealing with state is neglected. On the contrary, self-testing components are

deployed with test suites in a framework enabling their automatic execution, thus the system integrator does not have to generate the test cases.

8 Conclusions and Future Work

Component-Based software engineering is a promising approach for developing complex heterogeneous software systems that integrate classic computation and communication aspects. Testing component-based systems presents new problems with respect to classic software systems, since components are developed with limited knowledge about their use, and system developers have limited information about the internals of the reused components.

In this paper, we propose a new approach for automatically developing self-testing components. The proposed approach augments classic components with self-testing features by automatically extracting information from components when they are used as part of running systems. Automatically generated self-testing features support automatic execution of integration testing at deployment time. In this paper, we focus on the problem of automatically generating test cases from behavior invariants of components.

We are currently developing a prototype for experimenting with self-testing components to evaluate the effectiveness of the approach and to compare different criteria to generate test suites. Criteria will be evaluated with respect to their size, their efficacy in revealing faults and their applicability at different development and deployment stages.

References

1. Martins, E., Toyota, C., Yanagawa, R.: Constructing self-testable software components. In: Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01), Washington - Brussels - Tokyo, IEEE (2001) 151–160
2. Bertolino, A., Polini, A.: A framework for component deployment testing. In: Proceedings of the 25th international conference on Software engineering, IEEE Computer Society (2003) 221–231
3. Edwards, S.H.: A framework for practical, automated black-box testing of component-based software. Journal of Software Testing, Verification and Reliability **11** (2001)
4. Oberg, J.: Why the mars probe went off course. IEEE Spectrum **36** (1999) 34–39
5. Laboratory, J.P.: Report on the loss of the mars polar lander and deep space 2 missions. Technical Report JPL D-18709, California Institute of Technology (2000)
6. Weyuker, E.: Testing component-based software: A cautionary tale. IEEE Internet Computing **15** (1998) 54–59
7. Agrawal, V., Kime, C., Saluja, K.: A tutorial on built-in self-test. i. principles. IEEE Design & Test of Computers **10** (1993) 73–82
8. Binder, R.: Design for testability in object-oriented systems. Communications of the ACM **37** (1994) 87–101

9. Liu, C., Richardson, D.: Software components with retrospectors. In: Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA). (1998) 63–68
10. Beizer, B.: Software Testing Techniques. 2nd edition edn. Van Nostrand Reinhold Computer (1982)
11. Ramachandran, M.: Testing reusable software components from object specification. SIGSOFT Softw. Eng. Notes **28** (2003) 18
12. Leon, D., Podgurski, A., White, L.J.: Multivariate visualization in observation-based testing. In: Proceedings of the 22nd International Conference on Software engineering, ACM Press (2000) 116–125
13. Mariani, L., Pezzè, M.: A technique for verifying component-based software. In: International Workshop on Test and Analysis of Component Based Systems, Electronic Notes in Theoretical Computer Science (ENTCS) (2004)
14. Mariani, L.: Capturing and synthesizing the behavior of component-based systems. Technical Report LTA:2004:01, Università di Milano Bicocca (2003)
15. White, L., Cohen, E.J.: A domain strategy for computer program testing. IEEE Transactions on Software Engineering **6** (1980) 247–257
16. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27** (2001) 99–123
17. Harder, M., Mellen, J., Ernst, M.D.: Improving test suites via operational abstraction. In: Proceedings of the 25th international conference on Software engineering, Portland, Oregon, IEEE Computer Society (2003) 60–71
18. Microsystems, S.: Enterprise javabeans^{TM} specification. Final Release Version 2.1, Sun Microsystems (2003)
19. Vitharana, P.: Risks and challenges of component-based software development. Commun. ACM **46** (2003) 67–72
20. Binder, R.V.: Design for testability in object-oriented systems. Commun. ACM **37** (1994) 87–101