Generation of Integration Tests for 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 often being used in ways not envisioned by their developers, thus the packaged test specifications and suites cannot be relied upon. Often this results in conditions being placed upon a components use, however, what is required is a method for allowing test suites to be adapted for new situations. 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 are often deployed and tested independently from communication services that are 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

[⋆] This work has been partially funded by a grant from the SegraVis Research Training Network – www.segravis.org 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 testing 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. However, self-testing features are targeted at testing for chip malfunctions and the assumption is made for hardware components that the functionality of the component is sound. 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 the basic idea underlying self-testing components. Section 3 describes how we derive the Invariants that we use for generating test suites associated to components. Section 4 presents the different techniques that we developed for filtering test cases for self-testing. Section 5 discusses the problems of executing self-tests. Section 6 highlights the advantages of the new approach presented in this paper comparing it with related work. Finally, Section 7 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, for example the Mars Climate Observer [4], the Mars Polar Lander [5] or the Arianne 5 [6] problems.

In hardware testing, some of these problems have been addressed by developing Built-In Self-Test (BIST) features. “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 both 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 due to chip 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]. This work suggests various approaches and provide encouraging preliminary results. Our work enhances the previous solutions proposing a novel approach for generating test cases to create self-testing components.

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 5.

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 the generation of 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 the need for additional scaffolding.

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

- The test suite can be *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 can be *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 existence of an accurate and complete specification.
- The test suite can be *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.

Our approach is based on three main phases:

Generating Invariants: in this phase, we generate both proper and likely Invariants. Proper Invariants are generated from statically analyzing the source code, and describe properties of interactions of the component with the systems it is used in. Likely Invariants are automatically generated by monitoring the component behavior when integrated in running systems, and describe both interactions and exchanged values between the component and the embedding systems. Invariants provide the information required during the test generation phase to discriminate relevant behaviors that are selected as interesting test cases.

Recording Behaviors: in this phase, we record behaviors during normal executions of systems that embed the target component, to gain information that can later be used for generating test cases.

Generating Test Suites: in this phase, we generate test suites by sampling the set of collected behaviors according to the recorded Invariants.

The main contribution of this paper is first in assembling previously exploited technology to derive Invariants and record behaviors and then in proposing a set of criteria to sample the space of behaviors for selecting test suites.

3 BCT Framework

The BCT Framework [13] is a suite of tools that we use to monitor and dynamically verify components. BCT dynamically extracts information regarding components' behaviors as Invariants and records actual behaviors. We will show in the next sections how to use this information to generate test cases.

Invariant Generation

The information extracted during component monitoring consists of a specification of components interactions in the form of Interaction and I/O Invariants. Interaction Invariants describe the protocol that the monitored component uses to communicate with the other components. I/O Invariants describe what information is passed between components.

Interaction Invariants represent the pattern of interactions of each service of a component with other components. These Invariants are captured at run-time as a finite state machine which can then be expressed as a regular expression. For instance, the Interaction Invariant associated to a service `viewCart` of a `Cart` component for webshop applications can be:

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

and 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.

Regular expressions consist of elements of the alphabet, and operators. The alphabet represents interactions with other components. In the context of Interaction Invariants two operators are used. $|$ is the union operator which specifies that either the left or right symbol can be matched, and $*$ is the Kleene operator which specifies that the previous symbol can be matched 0 or more times. In the example $a * (b|c)$ the alphabet consists of $\{a, b, c\}$ while the operators $\{*, |\}$ are used.

We derive Interaction Invariants both statically and dynamically. We statically generate Interaction Invariants by building a reduced control-flow graph where nodes represent services requests. A regular expression can then be inferred from this graph. We dynamically generate Interaction Invariants by monitoring the components interactions with the system, further details can be found in [13, 14]. Static generation produces Interaction Invariants which represent all possible interactions of a components. This is in contrast to dynamic generation which produces Invariants representing only the observed behaviors of a component.

I/O Invariants represent properties over parameters of the components interaction with 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 qt (item quantity) is always positive when the service is invoked in that context.

We derive I/O Invariants dynamically by monitoring the execution of a component. Thus, I/O Invariants are Invariants over observed executions, that represent "likely" Invariants for the target component. Further information regarding how state data is extracted from complex objects can be found in [13, 14].

Recording Observed Behaviors

The BCT Framework has the facility to record observed behaviors of a component. An observed behavior consists of some form of stimulus to the component as well as its effect in the form of an interaction trace (the sequence of interactions triggered by the stimulus) and I/O values. These observed behaviors can be stored allowing them to be used as test cases for integration testing. However, indiscriminately storing these behaviors will soon produce large test suites that are of no practical use. Therefore, it is necessary to define filtering criteria for distinguishing which behaviors are meaningful or more specifically those likely to expose integration faults. These filtering criteria are presented in Section 4.

4 Test Case Selection

The monitoring of a component records a set of observed behaviors which can be used as possible tests. We produce test suites by extracting a subset of this set according to suitable *filtering criteria*. These filtering criteria utilize the generated Invariants as a specification of the components behavior. Our filtering criteria differ from classic testing methodologies in that we do not intend to test the components compliance to the Invariants, instead we use them as a means of identifying meaningful test cases. In this context we refer to a "meaningful test case" as one likely to expose integration faults.

In this section we propose a number of filtering criteria based upon the generated Interaction and I/O Invariants. These filtering criteria can be further categorized as those based upon post-execution filtering and those performed during the process of generating the Invariants.

Filtering criteria based on Interaction Invariants can be defined for Invariants expressed as either regular expressions or the corresponding finite state machines.

Three entities can be considered in a regular expression: the alphabet, the operators, and possible subexpressions. We therefore propose three filtering criteria based upon each entity:

Alphabet Coverage: A test suite TS satisfies the Alphabet Coverage criterion if for each symbol a in the alphabet, there is at least a test case TC in TS that contains a.
For example, the test suite $\{abc\}$ satisfies the alphabet coverage for the regular expression $a^*b(b|c)$.
Executing a test suite that satisfies alphabet coverage results in exercising all services accessed by the component-under-test at least once.

Operator Coverage: A test suite TS satisfies the Operator Coverage criterion if

- for each union operator | occurring in the regular expression, TS includes at least one test case TC_a that contains the first operand and one test case TC_b that contains the second operand.
- for each Kleen operator $*$ occurring in the regular expression, TS includes at least one test case TC_a that corresponds to no iterations of the operand, one test case TC_b that corresponds to exactly one iteration of the operand, and one test case TC_c that corresponds to more than one iteration of the operand.

For example, the test suite $\{bb, abc, aabb\}$ satisfies the Operator Coverage criterion for the regular expression $a^*b(b|c)$.

Expression Coverage: A test suite TS satisfies the Expression Coverage criterion if for each choice of operators that results in a sentence S up to consecutive iterations of the Kleen operators, it contains at least one test case TC corresponding to S.
For example, the test suite $\{bb, bc, abb, abc, aabb, aabc\}$ satisfies the expression coverage criterion for $a^*b(b|c)$.
Execution of a test suite that covers expressions result in the component executing each possible combination of behaviors.

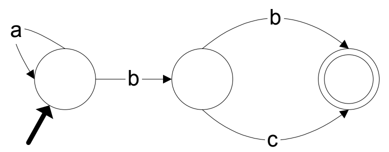

Fig. 1. The finite state machine that is equivalent to the regular expression $a^*b(b|c)$. The node with the string incoming edge represents the initial state, while the node with the double border represents the final state.

Three entities can be considered in a finite state machine (FSM): nodes, edges and paths. We therefore propose three filtering criteria based upon each entity:

Node Coverage: A test suite TS satisfies the Node Coverage criterion if for each node n of the FSM there is at least one test case TC that traverses n. For example, the test suite $\{bb\}$ satisfies the Node Coverage criterion for the FSM in Figure 1.
Execution of a test suite that satisfies the node coverage criterion corresponds to covering all states of the component-under-test where a decision about the next interaction can take place.

Edge Coverage: A test suite TS satisfies the Edge Coverage criterion if for each edge e of the FSM there is at least one test case TC that traverses e. For example, the test suite $\{abb, abc\}$ satisfies the Edge Coverage criterion for the FSM in Figure 1.
Execution of a test suite that satisfies the edge coverage criterion results in requesting the execution of all services accessed by the component-under-test at least once in each different situation.

Path Coverage: A test suite TS satisfies the Path Coverage criterion if for each path p of the FSM there is at least one test case TC that traverses p. Cycles are covered according to the boundary interior criterion that limits path coverage to each different path contained within the cycle at least once [15]. For example, the test suite $\{bb, bc, abb, abc\}$ satisfies the Path Coverage criterion for the FSM in Figure 1.
Execution of a test suite that covers all paths results in the component executing all possible "linear indepedent" combinations of behaviors at least once.

I/O Invariants describe properties over possible values of the parameters of a component's service. We can therefore refer to the properties specified by I/O invariants to derive a new filtering criterion. White and Cohen proposed a test case selection criterion based on mathematical properties over variables [16]. Test cases selected by the White and Cohen criterion include stimuli that are both feasible and infeasible according to the specification. We propose a criterion that applies White and Cohen criterion to our set of I/O Invariants. We restrict the criterion to feasible stimuli only, since it is not possible to observe infeasible stimuli at run-time. The obtained filtering criterion is:

Domain Value Coverage: A test suite *TS* satisfies the Domain Coverage criterion if for each I/O Invariant it contains at least one test case corresponding to both boundary and normal values as shown in Table 1

Invariants over any variable

Invariant	Test Case Spec
$x = a$	-
$x = uninit$	-
$x \in \{a, b, c\}$	$x = a$, $x = b$, $x = c$

Invariants over a single numeric variable

Invariant	Test Case Spec
$a \leq x \leq b$	$x = a$, $x = a + 1$, $a + 1 < x < b - 1$, $x = b - 1$, $x = b$
$x \neq 0$	$x < -1$, $x = -1$, $x = 1$, $x > 1$
$x \equiv a(mod\ b)$	-
$x \not\equiv a(mod\ b)$	$x < a - 1(mod\ b)$, $x = a - 1(mod\ b)$, $x = a + 1(mod\ b)$, $x > a + 1(mod\ b)$

Invariants over two numeric variables

Invariant	Test Case Spec
$y = ax + b$	-
$x < y$	$y = x + 1$, $y > x + 1$
$x \leq y$	$y = x$, $y = x + 1$, $y > x + 1$
$x > y$	$x = y + 1$, $x > y + 1$
$x \geq y$	$x = y$, $x = y + 1$, $x > y + 1$
$x = y$	-
$x \neq y$	$y = x + 1$, $x = y + 1$, $x < y$, $y > x$
$x = fn(y)$	-
any inv. over $x + y$	triple tests with $x = 0$, $y = 0$, $x, y \neq 0$

Invariants over three numeric variables

Invariant	Test Case Spec
$z = ax + by + c$	-
$y = ax + bz + c$	-
$x = ay + bz + c$	-
$z = fn(x, y)$	-

Invariants over a single sequence variable

Invariant	Test Case Spec
Min and Max values	-
nondecreasing	-
nonincreasing	-
equal	-
inv. over all elem.	-

Invariants over two sequence variables

Invariant	Test Case Spec
$y = ax + b$	-
$x < y$, $x \leq y$, $x > y$, $x \geq y$, $x \neq y$	test specs for the single element applied to all elements
$x = y$	-
x subsequence of y, or vice versa	subsequence at the beginning, middle, and end
x is the reverse of y	-

Invariants over a sequence and a num. var.

Invariant	Test Case Spec
$i \in s$	i at the beginning, middle, and end of s

Table 1. Filtering criteria for I/O Invariants. x, y and z are variable names or sequences; a , b and c are constants; fn is any language specific function; and $-$ shows that no test cases are necessary to cover the Invariant.

All filtering criteria defined so far are applicable to a set of observed behavios that derive from previously recorded executions. We identify them as *post-execution* criteria.

Recording a large set of executions may be space consuming. Therefore, we define additional criteria that can be used at run time, thus reducing the set of executions to be recorded. We identify this new set of criteria as *run-time* criteria.

We propose the following set of *run-time* criteria:

Interaction Invariant Modification: the test suite is built incrementally by adding all observed behaviors whose execution causes an Interaction Invariant to be modified.

Interaction Invariant Modification with Observed Frequencies: this criterion extends the previous one by adding special cases corresponding to the Kleene operators. We add an observed behavior to the test suite if the behavior corresponds to a number of iterations of the operand of a Kleen subexpression not yet experienced.

I/O Invariant Modification: we incrementally add observed behaviors whose execution results in the modification of an I/O Invariant. This criterion is based upon the work of McCamant and Ernst [17].

Subsumption Relationship

The filtering criteria we have proposed do not produce test suites that are mutually exclusive. In that certain criteria expand upon the coverage provided by others. We can therefore classify filtering criteria into a subsumption hierarchy. We define subsumption as:

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

Figure 2 summarizes the relationships among the filtering criteria presented in this paper. The reasoning behind these relations is as follows:

– The subsumption relationship between Alphabet Coverage, Operator Coverage and Expression Coverage is quite obvious. As is the relationship between Node Coverage, Edge Coverage and Path Coverage.
– Edge Coverage subsumes Alphabet Coverage as each edge in the FSM corresponds to different symbols. Thus covering all edges implies that all symbols are covered.
– Operator Coverage subsumes Edge Coverage as operators are represented as edges. Thus covering all operators implies that all edges are covered.
– Expression Coverage subsumes Path and Operator Coverage as it considers all possible combinations of behaviors.
– Interaction Invariant Modification subsumes Edge Coverage due to the way in which Invariant modification is defined in [13]. In this definition a new edge is added if a new behavior is observed. Thus covering all modifications implies all edges are covered.
– Finally, Interaction Invariant Modification with Observed Frequencies is a clear extension of Interaction Invariant Modification.

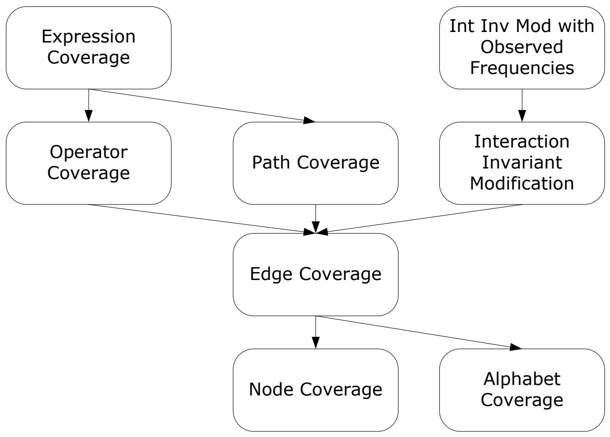

Fig. 2. Subsumption relationship over the proposed filtering criteria

5 Executing the Test Suite

Integration testing focuses upon how components interact within the embedding system. Two forms of interactions can be observed: how a component uses the system, and how the system uses a component. Integration faults can occur in either or both situations. The first form of interaction can be tested by executing a test suite covering the Interaction and I/O Invariants of the component over the accessed services. The second form of interaction can be tested by executing test suites of all components that interact with the component.

Components can be both stateless and stateful. Stateless components can be tested by simply invoking services over the tested component and checking the result. Stateful components could be directly executed but since our test cases are extracted from observed behaviors of the component they are only valid when the system is in that state. Therefore, if actual state is different from the state at test case extraction time, 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, the tester does not 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. Whilst this approach prevents state errors from occurring it risks test cases not being executed negating the confidence in the test suite.

The solution that we envision is to design the components for testability, by adding an interface that would facilitate the storing and resumption of state. In this way, it is possible to store the state of components involved in an execution and to resume them when necessary. 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. In fact, some middleware already provides features for storing and resuming the status of components as well as functionalities for querying the storage of status, for instance see the EJB framework [18] and the EJB-Object Query Language [18].

6 Related Work

The basic problems of testing component-based systems are summarized by Vitharana [19] . Vithrana highlights 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. This is achieved by a combination of approaches including formalized design and testing strategies and embedded test specifications. 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 assume that components may be deployed in ways not previously envisioned by the developer. Therefore, pre-defined test specifications may no longer be valid. We address this situation by constructing test suites based upon the observed behaviors of the running 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 testers 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 not addressed. On the contrary, our 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.

7 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. Through this experimental work we aim to be able to evaluate the filtering criteria, specifically with reference to the fault revealing capability and the size of the test suite.

Regarding the use of filtering criteria to create test suites, we do not expect to find a single ideal criterion. As with many situations within software engineering the choice of filtering criteria largely depends upon the intentions of the component developer or system integrator. For example, a developer constructing a component for an embedded system (where resources are limited) may prefer a test suite of minimal size whilst still covering all common integration errors. This is in contrast to a developer of a safety critical component who may prefer a test suite which covers all possible interactions and where test suite size is less important. We aim to evaluate each filtering criteria, highlighting their strengths and weaknesses, both generally and related to specific applications scenarios, e.g., enterprise components versus embedded components. We expect that this work will allow a developer to select the most appropriate filtering criteria for their specific application.

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. Jet Propulsion Laboratory: 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 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) 16–28
14. Mariani, L.: Capturing and synthesizing the behavior of component-based systems. Technical Report LTA:2004:01, Università di Milano Bicocca (2003)
15. Howden, W.: Methodology for the generation of program test data. IEEE Transaction Computer (1975)
16. White, L., Cohen, E.J.: A domain strategy for computer program testing. IEEE Transactions on Software Engineering **6** (1980) 247–257
17. McCamant, S., Ernst, M.D.: Predicting problems caused by component upgrades. In: proceedings of the 9th European Software Engineering Conference and the 10th International Symposium on Foundations of Software Engineering, ACM Press (2003) 287–296
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. Communications of the ACM **37** (1994) 87–101