Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Robustness testing for software components

Bin Lei [a,b], Xuandong Li [a], Zhiming Liu [b,*], Charles Morisset [b], Volker Stolz [b]

[a] *Department of Computer Science and Technology, Nanjing University, China*
[b] *International Institute for Software Technology, United Nations University, Macao*

A R T I C L E   I N F O

Article history:
Received 17 March 2009
Received in revised form 26 January 2010
Accepted 11 February 2010
Available online 12 March 2010

Keywords:
Robustness testing
Component
Contract
State machine

A B S T R A C T

Component-based development allows one to build software from existing components and promises to improve software reuse and reduce costs. For critical applications, the user of a component must ensure that it fits the requirements of the application. To achieve this, testing is a well-suited means when the source code of the components is not available. *Robustness testing* is a testing methodology to detect the vulnerabilities of a component under unexpected inputs or in a stressful environment. As components may fail differently in different states, we use a state machine based approach to robustness testing. First, a set of paths is generated to cover transitions of the state machine, and it is used by the test cases to bring the component into a specific control state. Second, method calls with invalid inputs are fed to the component in different states to test the robustness. By traversing the paths, the test cases cover more states and transitions compared to stateless API testing. We apply our approach to several components, including open source software, and compare our results with existing approaches.

1. Introduction

Component based design is about how to use existing components in the design or maintenance of a new and larger application. In order to integrate a component within a larger system, three major properties, the fitness, the correctness, and the robustness, have to be verified without information about the implementation of the component, as the source code is in general not available to the designer of the application program. The fitness of a component for an application is in general treated as the compatibility of the contract of the provided interface of the component and the specification of the required interface of the application. The correctness of a component is its ability to return the correct output when provided with the correct input, while the robustness concerns the absence of a behaviour possibly jeopardising the rest of the system, especially under a wrong input. To check correctness, we require the explicitly specified contract of the interface of the component which describes the black-box interface behaviour of the component [31]. Such a contract is enough only if the component is closed in the sense that it fully implements the functionality (or services) which it provides, without requiring services from other components. To simplify the argument, we assume that the component is closed. In the case where the component is open and also requires services from the environment, the following specifications are further needed [6]:

- the contract of the required interface of the component, and
- the specification of the services provided to the component by other components, which are either parts of the application or other existing components to be used in the application.

* Corresponding address: International Institute for Software Technology, UNU-IIST, P.O. Box 3058, Macao.
E-mail addresses: bl@iist.unu.edu (B. Lei), lxd@nju.edu.cn (X. Li), lzm@iist.unu.edu, z.liu@iist.unu.edu (Z. Liu), cm@iist.unu.edu (C. Morisset), vs@iist.unu.edu (V. Stolz).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.02.005

A large number of techniques has been developed for effectively checking the fitness of a component. In particular, we have developed a theory of component-based design, called rCOS, based on the notion of *refinement of components and objects* [6,23]. In rCOS, an *interface of a component* is a declaration of a number of data variables called *fields*, whose types can be classes in object-oriented programming as well as simple data types, and a set of method signatures, each having a method name and a list of typed parameters. A *contract of the interface* is a specification of the *data functionality* of the methods and the *protocol* in which the client of the component calls the methods. The functionality of a method is specified by a precondition and a postcondition in the form $p \vdash R$. The protocol is a set of traces over the signatures of the interface that can be represented by a regular expression.[1] This thus generalises the idea of *assumption-commitment* (and *rely-guarantee*) in a way that the preconditions and the protocol specify the *assumption on the environment* and the postconditions describes what the component *commits* to do when the environment follows the assumption.

Symmetrically, the requirements of the component by the application is also specified in terms of the data functionality of the methods, and the protocol in which it calls the methods. The compatibility is defined as such that the functionality of the methods of a component *refines* the functionality of the requirements, and the protocol of the application is a subset of the protocol of the component.

Consider a component for example, called `CurrencyConverter`, with an interface `CurConv` that declares a field `rate`, and the methods `setRate(double r)` to set an exchange rate, `convert(double amount;double result)` to convert using the exchange rate, and `convertInverse(double amount;double result)` to convert using the inverse of the exchange rate. To ensure a rate is set before other operations, we assume the protocol $(setRate \cdot (convert + convertInverse)^*)^*$. This component can be used in many systems, such as a banking system, a travel agency, or a price comparator. The designer of such a system has to be sure that the contract of `CurConv` meets the functional requirements of the system.

Checking the correctness of a component consists of verifying that the implementation meets the specification. For instance, the `CurrencyConverter` is correct only if, under the assumption that `rate` is different from zero, the method `convertInverse(double amount;double result)` indeed sets `result` to `amount / rate`. The behaviour of `convertInverse` is not specified when the `rate` is equal to zero. There are several techniques to ensure the correctness of a component, which can be either used by the developer of the component (as a guarantee to the user) or by its user (to check the claim of the developer). If the developer can use techniques like theorem-proving or model-checking, they require the source code since the construction of an abstract model from binary code is hard. However, the source code is usually not available to the user, who only has access to the contract, and therefore she cannot use such techniques. A possible solution is to certify the development, using for instance the Common Criteria [5], but the certification process is costly, in terms of time and money, and there is a lack of automatic tools for certification. The Proof-Carrying Code [27] approach also addresses this issue, but is hard to apply to large systems. Therefore, it seems that testing [1,11,13,18] is the choice left for effectively determining the correctness of a component, as it only needs the contract and executable code to verify a component. More examples of correctness testing for components are given in Section 6.

Another important property for the integration of a component within an application is its *robustness*, and that is the main subject of this paper. A component is robust when it never *fails* or *crashes*, whatever the input is. Indeed, the failure of a single component may cause the failure of the entire system, as it happened with the Ariane flight 501, where an arithmetic overflow of a simple piece of software calibrated for Ariane 4 led to the change of direction of the launcher, and therefore to its crash [10]. This piece of software was correct, since it was delivering the correct output for an input in the correct range, however the range was different with Ariane 5, and so it was called with an input violating the precondition. In general, nothing is known about the behaviour of a correct component if a user calls a method with an input that violates the precondition. For example in `CurrencyConverter`, one can correctly implement the component by writing `setRate(r){rate:=r}`. However, if during the execution of the program, the application makes a call to `setRate(0)`, the execution may eventually *crash* when `convertInverse` is invoked and an *exception may be raised*. Such an implementation is said to be *not robust*. Indeed, as recently reported in the Common Weakness Enumeration (CWE) Initiative, most of the 25 most dangerous programming errors[2] are not discovered by conformance testing. Instead, these errors are just assumed not to happen. These errors include, for instance, integer overflows, division by zero, and incorrect conversion between numeric types (listed in CWE-682).

The objective of the work presented here is to generate test cases to detect robustness failures. We assume that a component is correct, so if the precondition of a method is satisfied, then the post-condition is also satisfied after the execution and therefore nothing wrong can happen. We also assume that the component fits the application, so when the protocol is respected, the application cannot deadlock. Since we want to detect when things can go wrong, we therefore focus on test cases either violating the precondition or the protocol.

In this paper, we present a methodology such that given an rCOS model, which provides the specification of both the data functionality and the protocol of a component, robustness test cases are generated. For a test run, we produce a prefix of valid method invocations with arguments respecting both the protocol and the preconditions, and then execute the method calls with invalid arguments followed by a suffix of valid calls again, or we execute a single method call against the protocol with random arguments. The component is tagged as not robust when a crash is observed. We have implemented a prototype tool called *Robut* that takes UML-based rCOS component specifications as input and produces test scripts for Java classes.

[1] In general, it does not have to be a regular language, but regular is used here for convenience so that we can easily check the compatibility.
[2] http://cwe.mitre.org/top25/.

The next section introduces the rCOS method for component based design, focusing on the notion of contracts and implementation of components, and defining what we mean when we say an implementation is correct. Robust components are defined in Section 3 and we discuss the causes of non-robustness so as to motivate our method. Section 4 describes the methodology of the protocol based testing framework. In Section 5 we outline the implementation of the prototype tool and give experimental results for the evaluation of our tool. We discuss related work, summarise our work, and give a plan of future work in Section 6.

2. Component contracts

An essential concept in rCOS is the one of *components*. A component is modelled with different details at different stages of the software development's life cycle and for different purposes. A *component implementation* is a piece of program, and a *contract* of a component is for the requirements specification and design. A contract is also used as a *publication* for component usage and assembly.

In this section we summarise the definitions of the main models without going into technical details. Such details can be found in our earlier publications [15,6]. We define the notion of robust component implementation, that motivates our robustness testing method. We first introduce the semantic root of the rCOS theory so as to discuss the definitions precisely.

2.1. UTP as root of semantic theory

In principle, different components can be implemented in different programming languages. We thus need a semantic model for "unifying theories" of different programming languages, and thanks to Hoare and He, we use the well studied UTP [16] and its extension by He, Liu and Li to object-oriented programming [15]. The essential theme of UTP that helps rCOS is that *the semantics* $[\![P]\!]$ *of a program* P *(or a statement) in any programming language can be defined as a predicate*, called a *design*. A *design* is specified as a pair of pre- and post-conditions, denoted as $p(x) \vdash R(x, x')$, of the *observables* x and x' of the program. Its meaning is defined by the predicate $p(x) \wedge ok \Rightarrow R(x, x') \wedge ok'$, that asserts if the program executes from a state where the *initial values* x satisfies $p(x)$ the programs will terminate in a state where the final values x' satisfies the relation $R(x, x')$ with the initial values x. Observables include program variables and parameters of methods or procedures. The Boolean variables ok and ok' represents the observation of termination of the execution preceding the execution of P (i.e. ok is $true$) and the termination of the execution of P (i.e. ok' is $true$), respectively.

2.2. Interfaces and their contracts

A component is used through its *interface* according to a *contract* that specifies the *assumption* about the behaviour of its environment and the *commitment* of the functionality it provides.

Interfaces. In rCOS, an interface is a *syntactic notion*, and each *interface* I is a declaration of a set $I.fields$ of typed variables of the form $x : T$, called *fields*, and a set $I.methods$ of method signatures of the form $m(x : T; y : V)$, where $x : T$ and $y : V$ are the input and output parameters with their types. The interface of a component is used for syntactic type checking, i.e., if the component fits in an application. It does not provide any behavioural information.

Contracts. The contract of an interface of a component is a black box behavioural specification of what is needed for the component to be designed and used in building and maintaining a software system. The information needed depends on the application of the component. For example, for a sequential program, the specification of the static data functionality of the interface methods is enough, but information about the worst execution time of methods is needed for a real-time application, and for interactive systems we also need the interaction protocol in which the environment interacts with the component. For the treatment of different applications, the intention of rCOS is to support incremental modelling and separation of concerns. In this paper, we take a simplified version of rCOS contracts, that only consider data functionally and interaction protocols.

Definition 1. A **contract** $C = (I, \theta, \mathcal{S}, \mathcal{P})$ defines for an interface I, denoted by $C.IF$

- an *initial condition* θ, denoted by $C.init$, that specifies the allowable initial states of the intended component;
- a specification function $\mathcal{S}$, denoted by $C.spec$, specifying the data functionality by giving each method $m \in C.IF.methods$ a design $p(x) \vdash R(x, x')$, denoted by $C.spec(m)$, where x represents the fields of the interface and input parameters of m, and the primed version x' denotes the values of the fields and output parameters after the execution of the method;
- a protocol $\mathcal{P}$, denoted by $C.prot$, that is a set of finite traces, *i.e.* a set of finite sequences of method names in $C.IF.methods$, specifying the *interaction protocol* in which the environment may use the services provided by the component.

Note that the notion of a contract specifies three views of the component: (1) the data functionality view is modelled by *designs* $C.spec(m)$; (2) the interactive view by the protocol, which can be represented as a *state machine diagram*; and (3) the *data structure view* that defines the data- and class structures, which can be represented by a UML *class diagram*. When the protocol of the contract is a regular expression, it can be modelled by a finite state machine, and this is what we assume in this paper.

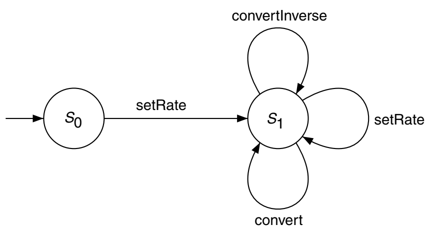

Fig. 1. State machine of `CurrencyConverter`.

Example 1. Consider the example of `CurrencyConverter`. The contract of its provided interface can be specified in rCOS as follows.

```
interface CurConv
  public double rate;
  public setRate(double val)
        {true ⊢ rate' = val};
  public convert(double amount; double result)
        {true ⊢ result' = amount ∗ rate};
  public convertInverse(double amount; double result)
        {rate > 0 ⊢ result' = amount / rate};
  protocol : (setRate.(convert + convertInverse)∗)∗
```

Notice that rCOS uses input/output parameters separated by a semicolon in method signatures.

We now formally define the model of finite state machines that are used to represent interaction protocols of components.

Definition 2. A **state machine** is a tuple $M = (S, s_0, \Sigma, \mathcal{T})$ where

- S is a finite set of states, and $s_0 \in S$ is the initial state;
- Σ is a finite set of input symbols;
- $\mathcal{T} \subseteq S \times \Sigma \times S$ is the transition relation.

Fig. 1 shows a state machine of the protocol of component `CurrencyConverter`, in which the input symbols are the method names of the component's interface.

2.3. Component implementation

At the source code level, components have a *provided interface K.pIF*, possibly a *required interface K.rIF*, and a piece of program code $K.code(m)$ for each method $m()$ in the provided interface. The required methods in *K.rIF* are those methods called in the code $K.code(m)$ of the methods m of the provided *K.pIF*, but they are neither declared in the provided interface *K.pIF* nor defined as internal methods of K. A component K is *closed* if it does not require methods from outside, i.e. it has no required interface. Otherwise K is called *open*. Note that a component K is developed and programmed to implement a contract of its provided interface *K.pIF*. To simplify the discussion, we avoid the problem of how the interaction protocol of the contract is realised in the implementation by simply assuming a known interaction protocol for the component's provided interface, denoted by *K.pProt*. It is not difficult to understand that given this protocol, the protocol *K.rProt* of the required interface *K.rIF* can be calculated from the code of the provided methods. The algorithm for this calculation is given in [22].

Example 2. Consider the example of `CurrencyConverter`. The contract is implemented by the following class.

```
class CurConv
  public double rate;
  public setRate(double val)
        {rate :=val };
  public convert(double amount; double result)
        {result :=amount ∗ rate};
  public convertInverse(double amount; double result)
        {if (rate = 0) then abort else result:= amount / rate};
```

With this implementation of the `CurrencyConverter`, the sequence of execution `setRate(0);convertInverse(1;x)` will abort, which is usually executed in Java or C++ by throwing an exception, and it is then the responsibility of the application using `CurrencyConverter` to catch it and to treat it. However, if such a treatment is not enforced, as it was the case during the Ariane flight 501, the failure of this single method can lead to the failure of the whole system. This situation is a *robustness* failure, addressed in the next section.

Table 1
Robustness degrees.

	Before execution		After execution		
Category	inv	pre	inv	post	exceptions
Correct	True	True	True	True	No
Incorrect	True	True	True	False	No
Crashing	True	True	?	?	Yes
Invariant preserving	True	False	True	?	No
Invariant breaking	True	True	False	?	No
Invalid input resistant	?	False	?	?	No
Invalid input crashing	?	False	?	?	Yes
Broken invariant resistant	False	?	?	?	No
Broken invariant crashing	False	?	?	?	Yes

3. Robustness testing

It is generally accepted that a component is *robust* if it can be executed without crashing, even when it is used inappropriately. IEEE defines robustness as the degree to which a system or component can function correctly in the presence of *invalid inputs* or *stressful environmental conditions* [17]. However, these notions can be ambiguous, and in order to clarify the context, we introduce in Table 1 different properties for a method of a component, according to the values of the class invariant and the postcondition, and the raising of exceptions after execution, depending on the values of the invariant and the precondition before execution. The question mark stands for either true or false.

The scope of this paper is to detect the exceptions thrown by a method, in order either to fix the component (or to ask the developer to fix it) or to enforce an exception handling mechanism. Thus, we call a component *robust* if none of its methods is *crashing*, *invalid input crashing* or *broken invariant crashing*. The main reasons for a component to be non robust are the ignorance of preconditions and incomplete specifications, as we see in the next section. We then give the definition of robustness test cases and robustness alarms, implemented here as exceptions.

3.1. Causes of robustness failures

The development of a component usually starts from its requirements specification, that is the contract, formally called *design by contract* [25]. It goes through a number of steps that *refine* the contract step by step into a program, either formally or informally. It is important to note that a formal development process by refinement should in principle not introduce more robustness failures or correctness failures than an informal development. A step of refinement of a method specification $p \vdash R$ is essentially to *weaken its precondition and strengthen its postcondition*. In the final implementation, the refinement is implemented as a piece of code P, and the *abstract semantics* $[\![P]\!]$ of P has $true$ as the precondition. However, the notions of correctness and refinement do not impose any constraint on how to implement the cases when the precondition is violated, because $p \vdash R$ does not specify what to do if p is violated.

For example, the specification $x \neq 0 \vdash y' = y/x$ can be *correctly implemented* by any of the programs:

P1. `if x=0 then abort else y:=y/x`, $[\![P1]\!] = true \vdash (x = 0 \vee (x \neq 0 \wedge y' = y/x))$;
P2. `y:=y/x`, $[\![P2]\!] = true \vdash y' = y/x$; and
P3. `if x=0 then skip else y:=y/x`,
$[\![P3]\!] = true \vdash (x = 0 \wedge y' = y) \vee (x \neq 0 \wedge y' = y/x)$.

Obviously, only program P3 is robust in any realistic implementation of the programming language. The question whether P3 is actually the "best" implementation is complex. Indeed, one might argue that it is better to have a program failing in an explicit way (P1 and P2) than a program failing silently (P3), especially to detect errors. However, P1 and P2 are merely delegating the error management to the caller, who then has the choice between failing explicitly or silently, since there is no possible valid answer for dividing by zero. Of course, a possible way to address this issue is to define a *complete* specification, i.e. describing the behaviour of the method for any input. For instance, the previous specification can be replaced by $true \vdash (x \neq 0 \Rightarrow y' = y/x) \wedge (x = 0 \Rightarrow raiseFlag())$, where $raiseFlag()$ is a method setting a flag to true, hence indicating that the result is wrong. The caller may then choose to ignore or not this flag, effectively turning any ill-defined operation into a skip statement.

Another source of runtime errors is an incomplete contract specification. A designer defining a component contract can easily miss the preconditions about division by zero, passing a null reference or pointer in a method invocation, an empty list, etc. The constraints of the platform for the implementation language, such as ranges for numeric types, are also usually forgotten, unless for specific domains like embedded systems or fault-tolerant systems. For instance, even though a language defines several types of integer like *long* or *short*, they are usually considered as numbers in $\mathbb{Z}$ in the semantics, and therefore without boundaries. However, multiplying two integers may lead to an inconsistent result and therefore cause runtime errors during the execution of a program. It is important to note that identification of these causes of runtime errors can be accumulated with the improvement of the knowledge of the implementation of the programming language and the growing experience in using the language. For a set of identified *language specific causes*, the precondition of a specification $p \vdash R$ can

Table 2
Error and exceptions.

Error	Java exception	C
Division by zero	ArithmeticException (for integers)	Floating Point Exception (signal)
Integer overflow	Not detected	Not detected
Null dereference	NullPointerException	Segmentation fault (signal)
Array out of bounds access	IndexOutOfBounds Exception	Not detected or eventual crash

be (almost) automatically strengthened with conditions about those causes. For example, the specification $true \vdash y' = y/x$ of P2 that misses the condition of division by zero can be transformed to $x \neq 0 \vdash y' = y/x$, and $true \vdash y' = y * x$ can be changed to

$$(MIN\_INT < x \wedge MIN\_INT < y \wedge |x| * |y| < MAX\_INT) \vdash y' = y * x.$$

The conclusion we draw from the above discussion is that, unlike conformance testing, robustness testing is mainly concerned with the execution results on an input that violates the contract.

3.2. Robustness test case

Since we assume that the component under test is correct, it cannot crash under correct use, so we focus here on the last two properties. While it may still be possible to call a method when its precondition is violated by selecting the arguments as described in Section 4.2, it is more complicated to construct a setting for a method call where the invariant is broken: usually it is not possible to attain a state where the invariant does not hold, since the methods are assumed to be correct, and thus preserve the invariant.

To increase the chances to detect a failure, we call several methods in a row with violated preconditions. Indeed, the behaviour of a method with an invalid precondition is not known, and thus it may or may not preserve the invariant. Therefore, by calling several methods, we augment the chances to eventually break the invariant, although it cannot be guaranteed that we actually break it. We now define our terminology of component-based robustness testing, including *robustness testing cases* and *robustness testing failure alarms*. Since these test cases are generated for an rCOS model, the contract, and therefore the preconditions of the methods and the protocol are available.

Definition 3. Given a closed component K and its contract C, assume a set **LSE** (for language specific error) of "dangerous" input values (e.g. the *null* pointer). An input v of a method invocation of a method m in $K.pIF$ is **hostile**, if v violates the precondition of $C.spec(m)$, or $v \in LSE$.

An input v to a method invocation $m(v)$ is *valid* if it is not hostile. The set LSE contains empirical values, which can potentially lead to errors even if they do not violate the precondition of a method. We include such values in the case where they have not been covered by conformance testing. Examples of LSE are given in Section 4.2.

Definition 4. Given a closed component K and its contract C, a **robustness test case** is a sequence π of method invocations $\pi = m_1(v_1), \ldots, m_n(v_n)$ satisfying one the two following properties.

- For some $i, j = 1, \ldots, n,\ i \leq j$, $v_i, \ldots, v_j$ are hostile arguments of the respective invocations of $m_i, \ldots, m_j$, and π is in the protocol $C.prot$ with valid arguments $v_1, \ldots, v_{i-1}, v_{j+1}, \ldots, v_n$.
- For every $i = 1, \ldots, n$, v_i is a valid argument of the invocation of m_i and the sub-sequence $m_1, \ldots, m_{n-1}$ is in $C.prot$ but π is not, i.e. the method invocation m_n does not respect the protocol of the component. Such an invocation is called **inopportune** in the following.

Component K **passes** the robustness test case, if the execution of the sequence of the above method invocations terminates normally, otherwise K **fails** the robustness test case, and a **robustness alarm** is reported.

Different mechanisms are used to implement robustness alarms, mostly according to the language. The prototype of our testing tool is designed for testing Java components, and therefore a robustness alarm is implemented as throwing an exception, as we see in the next section.

3.3. Exceptions and robustness

In modern programming languages like Java, programs usually do not crash immediately, but trigger an exception (Java, C++) or signal (C/OS-level) that still allows the consumer to handle this situation. Common examples are (see Table 2) the *NullPointerException* in Java (segmentation fault signal), or the *DivisionByZeroException* (floating point exception signal). In the following, we will discuss the relation between exceptions and robustness in detail, as they have also become a major programming feature that we need to consider.

Java distinguishes checked and unchecked exceptions. Unchecked exceptions are usually triggered by the Java Virtual Machine in the cases we previously discussed, like the *NullPointerException*. The programmer is not required to handle those exceptions. If they remain unhandled, they are propagated to the top-level of the call stack where the runtime system will then terminate the program with a corresponding error message.

Checked exceptions are user-defined exceptions, that are also enforced by the type checker: they have to be declared on the method's signature, and if a method which may throw a checked exception is called, it has to be handled in one of two ways by the programmer: either she chooses to ignore it, which means that she allows it to propagate. In that case, she adds the exception to her method signature. Or, she handles it using a `catch`-statement, in which case she can inspect the exception and take some action. She may decide to re-throw this, or another, exception, or initiate some kind of error handling.

Exceptions have become a standard feature of "programming contracts". Very often they are precisely documented ("if passed a null argument, the method will throw an *IllegalArgumentException*"). Both checked and unchecked exceptions are extensively used to deliberately indicate error conditions. Outside the scope of robust component based software development, contemporary software development uses exceptions as a feature of communication error conditions out-of-band (with respect to plain return values). It may then not be appropriate to take an occurrence of a checked exception as a robustness failure, since the compiler will force the application programmer to handle this exception in any of the two ways discussed above. We should not further distinguish between checked and unchecked exceptions, but differentiate between declared and undeclared *unchecked* exceptions, depending on whether an exception and its cause are precisely documented.

In the context of robustness, any declared unchecked exceptions are a strong indication that the component is not robust, since some inputs may trigger them. The burden is on the consumer of the provided service, and we should not make any assumptions on how/if this exception will be handled.

4. Method

The most important task for software testers is to detect as many defects as possible, usually within a given amount of resource constraints like time/number of test cases. Redundant test cases, i.e. tests that have identical cause, should thus be avoided. In the context of model-based testing, model coverage is a major factor to improve the efficiency of defect detection.

Given a component contract consisting of the class diagram with the preconditions of the functionality specifications for the public methods of the component, and its protocol in the form of a state machine, we generate a set of test cases. We first generate a set of paths, and then for each path, the corresponding sequence of Java statements is generated into a file, that can be executed and monitored to detect if exceptions are raised.

4.1. Path generation

We first give the notation used for method calls. Given a method m, $dom(m)$ stands for the domain of m, that is, the set of all possible input. Moreover, $dom_v(m)$ stands for the *valid* domain of m, that is, the subset of $dom(m)$ with inputs that satisfy the pre-condition of $dom(m)$, and $dom_h(m)$ stands for the *hostile* domain of m, that is, the subset of $dom(m)$ with inputs that either violate the pre-condition of $dom(m)$ or belong to LSE. Note that since LSE is defined regardless of any method, an input can be both in $dom_v(m)$ and in LSE, it follows that for a given method m the intersection of $dom_v(m)$ and $dom_h(m)$ is not necessarily empty.

We then formally introduce the notion of paths based on the notion of state machines given in Definition 2. Given a state machine $M = (S, s_0, \Sigma, \mathcal{T})$, a path is defined as: $s_0 \overset{m_1(x)}{\rightarrow} s_1 \overset{m_2(x)}{\rightarrow} \cdots \overset{m_n(x)}{\rightarrow} s_n$, where for all i, $s_i \in S \cup \{\perp\}$, $m_i \in \Sigma$ and $(s_i, m_{i+1}, s_{i+1}) \in \mathcal{T}$. Here, as we see below, $\perp$ is a special sink state used for inopportune invocations, from which no transition can be taken.

A single state is also considered as a sub-path, of length 0. We write Π for the set of all possible sub-paths and in the following; the method $last : \Pi \rightarrow S$ returns the last state of a sub-path. The partial function $\bullet : \Pi \times \Pi \rightarrow \Pi$ for concatenating some pairs of sub-paths is defined as:

$$(s_0 \overset{m_1(x)}{\rightarrow} \cdots \overset{m_i(x)}{\rightarrow} s_i) \bullet (s_i \overset{m_{i+1}(x)}{\rightarrow} \cdots \overset{m_n(x)}{\rightarrow} s_n) = s_0 \overset{m_1(x)}{\rightarrow} \cdots \overset{m_i(x)}{\rightarrow} s_i \overset{m_{i+1}(x)}{\rightarrow} \cdots \overset{m_n(x)}{\rightarrow} s_n$$

Notice that $p \bullet q$ is only defined when the last state of p is equal to the first state of q. We also introduce the function $\psi : \Pi \times \mathcal{T} \rightarrow \mathbb{B}$, which checks whether a path contains a given transition or not, regardless of the parameters used for the corresponding method call, and which is formally defined as:

$$\forall p \in \Pi \ \forall (s_1, m, s_2) \in \mathcal{T} \cdot \begin{pmatrix} & \psi(p, (s_1, m, s_2)) \\ \Leftrightarrow & \exists q_1, q_2 \in \Pi \ \exists x \in dom(m) \cdot p = (q_1 \bullet (s_1 \overset{m(x)}{\rightarrow} s_2) \bullet q_2) \end{pmatrix}$$

The algorithm presented below generates two kinds of paths, as depicted by Fig. 2. A path of the first kind starts with a sequence of valid method calls following the state machine. It then proceeds with a sub-path containing at least one method call with hostile input, followed a sub-path of valid method calls. A path of the other kind starts with valid method calls and then finishes with a single *inopportune method call*, i.e. a call that is not allowed by the state machine.

$$s_0 \underbrace{\overset{m_1}{\rightarrow} \cdots \overset{m_i}{\rightarrow}}_{\text{valid}*} s_i \underbrace{\overset{m_{i+1}}{\rightarrow} \cdots \overset{m_j}{\rightarrow}}_{\text{hostile}+} s_j \underbrace{\overset{m_{j+1}}{\rightarrow} \cdots \overset{m_k}{\rightarrow}}_{\text{valid}*} s_k$$

$$s_0 \underbrace{\overset{m_1}{\rightarrow} \cdots \overset{m_i}{\rightarrow}}_{\text{valid}*} s_i \quad \underbrace{\xrightarrow{m_1}}_{\text{inopportune}} \quad \perp$$

Fig. 2. Structure of generated paths.

The coverage criterion of this algorithm is *All-Transitions* [3], which ensures that all reachable states and transitions of the state machine are covered. Moreover, the transitions are covered by valid method calls. In other words, for any transition of the state machine, there exists a generated path containing this transition in the initial sub-path of valid method calls. This sub-path is generated using the method $cover : S \rightarrow \wp(\Pi) \rightarrow \wp(\Pi)$ defined for any state s by $cover(s)(\emptyset) = \{s\}$ and for any state s and for any non empty set of paths X by

$$cover(s)(X) = X \cup \left\{ \begin{array}{l} p \in \Pi \mid \exists q \in X\ \exists m \in \Sigma\ \exists x \in dom_v(m) \cdot \\ \qquad\quad \exists s_1 \in S \cdot ((last(q), m, s_1) \in \mathcal{T} \\ \qquad\quad \wedge\ (\forall q' \in X \cdot \neg \psi(q', (last(q), m, s_1))) \\ \qquad\quad \wedge\ p = (q \overset{m(x)}{\rightarrow} s_1)) \end{array} \right\}$$

This method appends to every sub-path in a set X a possible transition according to the state machine, if and only if this transition has not been used before. The argument s is used to initiate the generation, and is the first state of every generated sub-path. Note that if a transition can be reached through different paths, it is only chosen once.

The sub-path following the initial sub-path of valid method calls is either a sub-path containing at least one hostile method call or a sub-path with exactly one inopportune method call. For the first case, we define the method $genHostile : \wp(\Pi) \rightarrow \wp(\Pi)$ by for any set of sub-paths X:

$$genHostile(X) = \left\{ \begin{array}{l} p \in \Pi \mid \exists q \in X\ \exists m \in \Sigma\ \exists s \in S\ \forall x \in dom_h(m) \cdot \\ \qquad\quad ((last(q), m, s) \in \mathcal{T} \wedge p = (q \overset{m(x)}{\rightarrow} s)) \end{array} \right\}$$

To generate the inopportune method calls, the function $gen_\perp : \wp(\Pi) \rightarrow \wp(\Pi)$ is defined for any set of sub-paths X by:

$$gen_\perp(X) = \left\{ \begin{array}{l} p \in \Pi \mid \exists q \in X\ \exists m \in \Sigma\ \exists s \in S\ \exists x \in dom_v(m) \cdot \\ \qquad\quad (\neg((last(q), m, s) \in \mathcal{T}) \wedge p = (q \overset{m(x)}{\rightarrow} \perp)) \end{array} \right\}$$

The last sub-path contains valid method calls, covering the state machine transitions. This sub-path is generated using the function $genValid : \wp(\Pi) \rightarrow \wp(\Pi)$, which first covers the whole state machine, and then removes all the useless sub-paths, and which is defined for any set of sub-paths X by

$$genValid(X) = \left\{ p \in \Pi \mid \exists q_1 \in X\ \exists q_2 \in \mathbf{lfp}(cover(last(p)))_\downarrow \cdot p = q_1 \bullet q_2 \right\}$$

where for any set A, the operator $\downarrow$ removes all the redundant sub-paths, $A_\downarrow = \{x \in A \mid \nexists y \in A \cdot x \subset y\}$, and where **lfp** stands for the least fixpoint. This concatenation is always possible since by definition of *cover*, every sub-path in $\mathbf{lfp}(cover(last(p)))_\downarrow$ starts with $last(p)$. Although this operation increases the complexity of the test case generation algorithm, it decreases the number of generated test cases.

The set Π_v of valid paths covering all the transitions starting from the initial state is obtained by computing the least fixpoint of $cover(s_0)(X) = X$, denoted as $\mathbf{lfp}(cover(s_0))$, using the inclusion as ordering relation. For instance, for the `CurrencyConverter`, the generated set is:

$$\Pi_v = \left\{ \begin{array}{l} S_0,\ S_0 \overset{setRate}{\rightarrow} S_1,\ S_0 \overset{setRate}{\rightarrow} S_1 \overset{convertInverse}{\rightarrow} S_1, \\ S_0 \overset{setRate}{\rightarrow} S_1 \overset{convert}{\rightarrow} S_1,\ S_0 \overset{setRate}{\rightarrow} S_1 \overset{convert}{\rightarrow} S_1 \overset{setRate}{\rightarrow} S_1 \end{array} \right\}$$

The cardinality of Π_v is equal to the number of transitions N_T plus one, since every path added contains a transition which was not taken yet, and the empty path is also contained in this set. The set Π_g of all generated paths is then given as:

$$\Pi_g = genValid(genHostile^n(\Pi_v)) \cup gen_\perp(\Pi_v)$$

where $n > 0$ represents the length of the sub-path of hostile method calls. Note that since the method *genHostile* universally quantifies over dom_h, the set returned by this function is potentially very large. In practice, we arbitrarily bound the cardinality of this set by a number N_H. Let N_m be the cardinality of Σ, the method $gen_\perp$ creates for each path N_m new paths. The method *genValid* generates at most N_T new paths for each path, since it removes all the sub-paths. We can conclude that:

$$card(\Pi_g) \leq N_T^2 \star N_H^n + N_T \star N_m$$

The completeness of this tool is limited by the test selection criterion. To find more defects, other coverage criteria might help by producing more interesting test sequences. We plan to investigate in them for more efficient testing based on dynamic software models.

Although our approach cannot fully cover the state space of the component under test, the proposed methodology is sound in that each exception indicates a robustness defect inside the software.

4.2. Parameter generation for object oriented programs

The oracle that generates the parameters for method invocations must be able to handle both primitive values and objects. While choosing primitive values is relatively straightforward, we would also like to create non-trivial objects, or, more specifically, object trees. We first define the set *LSE* as the union of

- $lse.int \mathrel{\widehat{=}} \{MIN\_INT, -1, 0, 1, MAX\_INT\}$, and
- $lse.string \mathrel{\widehat{=}} \{null, empty\_string, very\ long\ string\}$,
- $lse.\tau \mathrel{\widehat{=}} \{null\}$ where τ is any class type.

We choose the parameters for an invalid method invocation (phase 2) as follows.

- For primitive parameters, the pool of language specific error values LSE is predefined as above.
- For primitive parameters constrained by preconditions, an oracle selects random values violating the predicates.
- For non-primitive parameters, the pool contains *null* and 1-depth instances constructed with either a *null* reference value or an invalid primitive value for fields.

In rCOS, *designs* give the method specifications, and may contain sequential composition of pre/post-conditions. As a simplification, but without loss of generality, we only assume a single precondition is given for each method. Our prototype tool analyses the preconditions as discussed in the previous section and uses them for effective random generation. For instance, in the `CurrencyConverter` example in the introduction, the precondition is $y \neq 0$, so the random generator picks a random value in [MIN_INT, 0) and (0, MAX_INT] when asked for a valid value. Note that static type checking and refinement typing [9] may detect some of the invalid inputs.

For *valid method calls*, the parameter objects/values are generated recursively. Given a type T in the parameter list, a method $generate(T)$ gets a concrete value.

- If T is primitive, return a random value of T satisfying the precondition;
- If T is non-primitive and method $T(T_1, T_2, \ldots T_n)$ is its constructor, return $T(generate(T_1), generate(T_2), \ldots, generate(T_n))$.

In the test case setup, a list of statements constructing $T_1, T_2, \ldots, T_n$ respectively is inserted before calling T's constructor. For simplicity, when T has multiple public constructors, we choose a random one for the test generation. We limit the recursion depth to less than five. When construction reaches the maximum depth, a *null* reference is returned. This limit is configurable and based on our experience.

4.3. Analysis of preconditions

Apart from the currency converter we used in the motivation, we consider the implementation of the Cashdesk component[3] from the CoCoME project [29] as modeled in the rCOS modeler [7]. This component is somewhat more interesting, since the preconditions go beyond primitive types, and the state machine is more complex. Fig. 3 shows the class diagram, with a provided interface for the *Cashdesk*. Fig. 4 is the state machine of the Cashdesk contract.

The following is a path ρ along the state machine. Note the constructor invocation on the first transition:

$$\begin{aligned}\rho \;=\; & S_0 \overset{new}{\hookrightarrow} Init \xrightarrow{enableExpress} ExMode \xrightarrow{startSale} NCE \\ & \xrightarrow{enterItem} NCE' \xrightarrow{finishSale} CE \xrightarrow{cashPay} CE\end{aligned}$$

We also give the non-trivial preconditions of the methods:

public enterItem(Barcode code, *int* qty)
  pre: store.catalog.find(code) $\neq$ **null** $\wedge$ qty > 0 $\wedge$ sale $\neq$ **null**

public cardPay(Card c)
  pre: sale $\neq$ **null** $\wedge$ c $\neq$ **null** $\wedge$ bank.authorize(c, sale.total)

public cashPay(double amount; double change)
  pre: sale $\neq$ **null** $\wedge$ amount $\geq$ sale.total

Since we do not have the other required components of the system like the bank and the store available, we use dummy or so-called “mock” implementations: we provide a store component that assumes that only even barcodes exist in the catalogue, and that odd barcodes do not exist, so that the precondition can occasionally fail. This is an example of *closing* the component before testing, albeit not to a fully functional system. Similarly, we model a credit card of type `Card` as a class which successfully authorises, if the card number is even in `bank.authorize()`.

[3] This example model is available as download from http://rcos.iist.unu.edu.

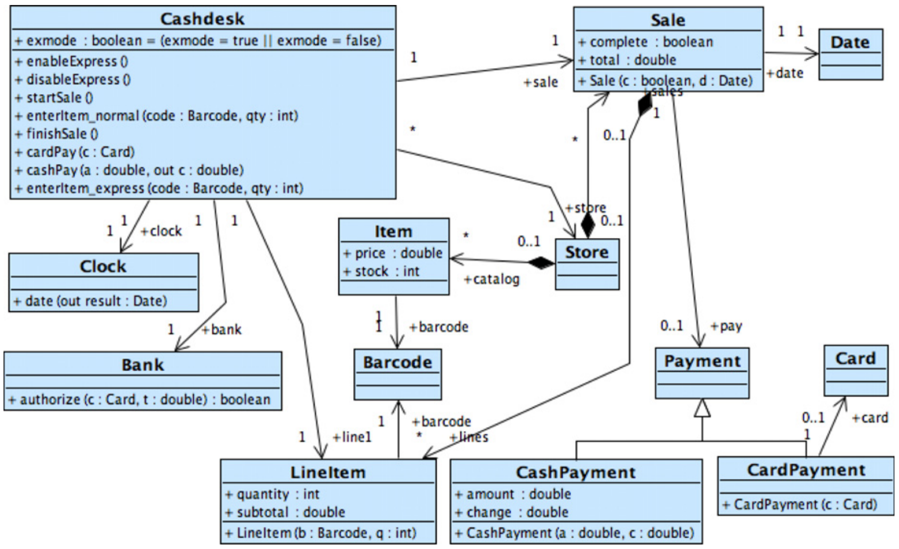

Fig. 3. Class diagram of the Cashdesk component.

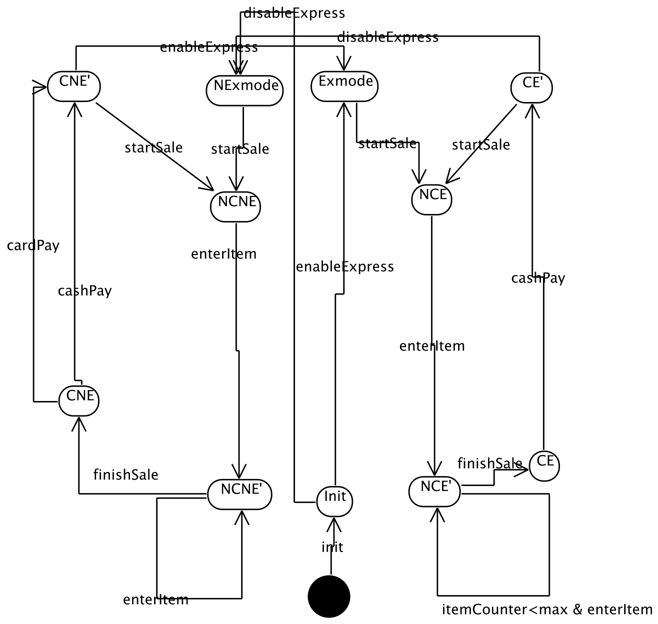

Fig. 4. State machine of the Cashdesk component.

We observe the following interesting features in the preconditions above (in increasing complexity): For the value `sale.total`, we would need to query its *current value during a test run* to chose an input value for the arguments to the payment methods. As this would effectively require us to run the component under test to generate the test cases, we filter this term from the precondition.

Additionally, we observe that a precondition may involve multiple conjuncts. In such a case, especially in robustness testing, a precondition may already be false regardless of the actual arguments, e.g. due to invalid input in a previous state. Also, different arguments may invalidate it independently, like the barcode *or* the quantity in the `enterItem` method.

In general, preconditions may also include queries that cannot be directly evaluated as a boolean function. The constraint `store.catalog.find(code)` may for example be expressed using quantification as:

$$\exists \text{ Item i} : \text{store.catalog.contains(i)} \wedge \text{i.barcode} = \text{code} \wedge \ldots$$

In that case, the oracle will have to create the intermediate objects *i* for the query, which might be difficult. In such cases, it may make sense to manually adapt the contract model to be more amenable to automated testing without sacrificing soundness.

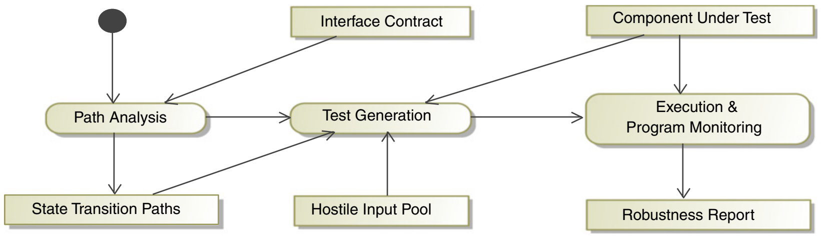

Fig. 5. Framework of robustness testing.

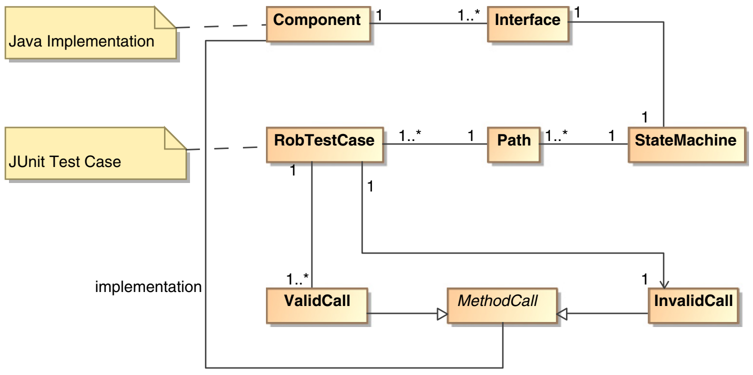

Fig. 6. Artifacts of *Robut* test case generation.

5. Tool implementation and experimental results

To validate the approach, we implement the testing methodology in a prototype tool named *Robut*. It consists of the following components: a *path analyser*, which explores a state machine to generate robustness test runs, and a *test case generator*, to compose executable test cases in the form of Java classes, and an exception classifier to produce a test report.

For a component under test, we first analyse the paths of its state machine based on transition coverage as discussed in Section 4.1. We invoke a public constructor for the class (component). A sequence of method calls with valid inputs following the state machine is invoked, to reach different states along the paths of the protocol. A sequence of method calls with hostile inputs is appended, trailed by a suffix of valid method calls. Alternatively, an inopportune method call is appended. Such a sequence forms a *robustness test cases*, see again Definition 4.

The activity diagram in Fig. 5 shows the overall work flow. Square boxes are artifacts used or produced in the testing process, while round boxes stand for the testing activities. The artifacts during test generation are summarised in the class diagram shown in Fig. 6. In the middle is the provided interface, which corresponds to the published services (methods) of the component under test. On the left is the component implemented in Java. On the right are specifications in UML diagrams. Our work focuses on generating robustness test cases based on state machines.

The prototype is built upon a set of libraries. Eclipse UML2 [12] and the rCOS modeller [23] are used to model the components and their interfaces. Java reflection [30] and Octopus' Java Metamodel [33] are used extensively to generate test inputs.

The prototype automates the test case generation and test execution. A JUnit [2] test script references a UML 2.0 model file with an rCOS contract specification, and the Java class of the component under test. A test run fails, when JUnit observes an exception.

5.1. Benchmark and test result

As shown in Table 3, we select a set of open source components to evaluate the testing framework. The Cashdesk component is from the CoCoME project [7]; the CurrencyConverter is the case we developed for testing purposes. Also, we test three container components from the Apache Commons library, the other three are input/output stream components from the GNU ClassPath project.

In the "Methods" column of the table, *x*–*y* stands for the number of public methods with preconditions versus the overall number of public methods in the component. For instance, in the Cashdesk row, 3–10 means that there are ten methods, but only three public methods with non-trivial preconditions.

Robut generates 18 paths to cover all the transitions of the Cashdesk component. Hostile inputs are generated for each state. For the *Cashdesk*, the number of invalid inputs generated for the states are shown in Table 4. Each row shows the numbers of types of test cases generated for a state from the state machine diagram. For instance, for state *NCNE*, the following test cases are generated: one invocation against the precondition, two values from the LSE set for primitive types, and seven inopportune calls. The combination of valid paths and hostile inputs forms 147 base robustness test cases. This process is

Table 3
Robustness testing benchmark.

Component	States	Transitions	Methods	LOC	Paths
Cashdesk	11	18	3–10	83	18
CurrencyConverter	3	5	3–3	30	5
ArrayStack	3	9	8–10	71	9
BinaryHeap	4	19	7–19	188	19
FastArrayList	5	33	10–85	792	33
BufferedInputStream	4	16	8–10	122	14
BufferedWriter	4	10	5–7	81	10
StringBufferInputStream	3	7	5–6	47	7

Table 4
Hostile inputs for different states of the Cashdesk component.

State	Pre	Null	LSE	Inopportune	Overall
Init	0	0	0	6	6
NExMode	0	0	0	7	7
ExMode	0	0	0	7	7
NCNE	1	0	2	7	10
NCNE’	1	0	2	6	9
CNE/CE	2	1	1	6	10
CNE’/CE’	0	0	0	6	6
NCE	1	0	2	7	10
NE	0	0	0	6	6
NCE’	1	0	2	6	9
NE	1	0	1	7	9

Table 5
Test results for the benchmark.

Component	All-TCs (R-S-F)	Failed-TCs (R-S-F)	Exceptions (R-S-F/A)
Cashdesk	615-108-50	52-0-33	5-0-4/5
CurrencyConverter	20-30-10	5-0-0	1-0-0/1
CircularBuffer	154-60-40	24-0-8	2-0-2/2
ArrayStack	192-54-10	87-12-10	6-2-4/6
BinaryHeap	165-114-0*	70-0-0	1-0-0/1
FastArrayList	369-198-40	296-137-40	6-3-3/6
BufferedInputStream	302-84-40	56-0-17	2-0-2/2
BufferedWriter	151-66-20	51-0-19	2-0-2/2
StringBufferInputStream	93-42-30	26-0-20	2-0-3/3

iterated a configurable amount of times to increase the number of test cases with similar structure, but different (random) arguments, as it is the only tunable in the test case generation algorithm to obtain more test cases (this applies for the alternative testing approaches below as well). We test the Cashdesk component with the generated test cases.

To compare our methodology with existing approaches, we implement the state based functional testing algorithm (SF) and the robustness testing approach based on a pure random algorithm (Fuzzy Testing, Fuz [26]).

An SF test case is the first part of our state based robustness test case, i.e. a valid method call sequence with random arguments. We apply the all-transitions coverage criterion for SF as well as our approach.

A Fuz test case just instantiates a component, and then calls one of the methods with arguments violating the precondition.

Note that these different testing methodologies lead to disparate numbers of test cases: for SF, the number of test cases depends on the size of the state machine and the number of valid argument combinations. The number of Fuz test cases is limited by the number of possible method calls with invalid preconditions. The number of test cases generated for Robut has been given in Section 4.1. Since Fuz does not generate traces but only single invocations, and SF test cases are basically prefixes of Robut test cases, the number of generated test cases will increase in this order.

We test the components in Table 3 with those three approaches. The results are shown in Table 5. There are three columns for each component under test: number of all test cases, number of failed test cases, and number of unchecked exceptions which indicates failures of components. In each column, there are three sub-columns, for the results of Robut, SF and Fuz. We use R, S, and F for short, to indicate the corresponding values in these three approaches. For instance, in the Cashdesk component, in the second column, 615, 108 and 50 stand for the number of test cases the algorithms generated for Robut, SF and Fuz. For the example marked with *, Fuz does not generate any test case *at all* because of the structure of component under test —all preconditions depend on the state and are thus filtered out, so Fuz cannot come up with a single suitable test case that would violate the precondition true. This is also the reason why Fuz does not detect an error in the Cashdesk

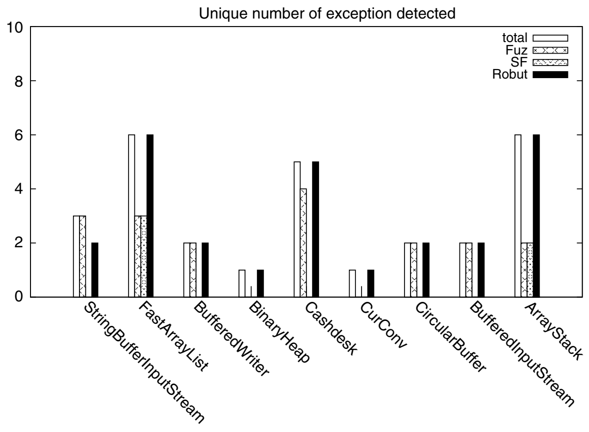

Fig. 7. Exceptions.

component, that is found by Robut: `finishSale()` will crash with a *NullPointerException* if invoked before `startSale`, e.g. at the beginning of a trace. But as this method has no arguments, Fuz is not able to test it. Another noteworthy entry exists for the Currency Converter, since due to the degenerate protocol (every action is enabled most of the time), Robut can mostly generate hostile, but not inopportune method calls, leading to less test cases than for SF.

The third column reports on the total number of unchecked exceptions captured by the test scripts. In general, the different runtime exceptions detected span those that we have mentioned before: *NullPointerExceptions, IndexOutOfBoundsExceptions*, and various application specific exceptions.

The last column is the most interesting one: it lists the *unique* detected unchecked exceptions by each approach, i.e. we do not count multiple occurrences of the same *exceptions per source code location*. The column A(All) stands for unique exceptions captured by all three approaches combined. For example, the 52 failures of the Cashdesk under Robut are due to five different unique exceptions. SF does not find any, and Fuz misses one of the exceptions found by Robut. Likewise, Robut does not find one of the exceptions (detected by Fuz due to a particular combination of arguments) in the StringBufferInputStream. Fig. 7 describes the number of detected unique exceptions of the three approaches from column four as a chart.

Robut generally detects more robustness problems compared to SF and Fuz. SF only concentrates on functional testing. It is the weakest in revealing robustness defects. Fuz's ability is in between the other two. However, compared to Robut, Fuz is not stable: for some components it detects as many robustness defects as Robut; but for some others, e.g. the CurrencyConverter or the Binary Heap, Fuz detects nothing. We conclude from the numbers that generally Robut effectively and stably detects robustness defects.

5.2. Discussion

We now discuss some practical issues when applying our methods, about the availability of models in model-based testing, and then about the problem of the size of the component understand under test.

Availability of models. Firstly, model availability is a key concern. Although it is the common problem for model-based testing, we still have to evaluate the possibility of applying our approach in reality.

For instance in Table 3, the Cashdesk and the CurrencyConverter component are developed with UML models in rCOS [23]. All the other components are taken from open-source projects which are not published with UML models. If the component under test was not designed by a rigorous model-based engineering process, the tester has to manually generate the model of the contract. That process is both time-consuming and error-prone. Fortunately, UML is gaining more popularity in both academia and industry. Platforms such as Eclipse and Rational Software Architects support design and coding in a coherent software life cycle. These widely adopted tool chains help to improve the availability of component models. On the other hand, feedback based random testing [28] and model synthesis techniques [34] are emerging to provide the means to test without explicit models. We plan to investigate them for robustness testing for future work.

Size of the component under testing. Second, the size of the resulting test suite can be very large for a component with many states and operations. In the test generation algorithm, transition coverage makes sure each transition in the state machine is covered at least once. Hence, the number of paths becomes large when the state machines are complex, see Section 4.1. The number of paths is then multiplied by the number of possible inopportune extensions for each leaf, where each extension is instantiated multiple times based on the preconditions, leading to e.g. 147 test cases for the Cashdesk. Since, depending on the precondition, there may only be a single instance of a particular path, we iterate this instantiation of an inopportune path by a configurable amount, here five, and obtain the 615 test cases reported on above. Note that the number of detected unique exceptions does not change significantly when increasing this factor to e.g. ten: usually Robut eventually picks up the missed exception, generating a "perfect" run, while the other two approaches do not find any previously missed exceptions for our test suite. This indicates that indeed the other two approaches seem to be limited with regard to detecting defects depending on control state.

In general, even with a factor to iterate test case generation, the number of test cases may still be bounded for a specific approach, as can be seen in the degenerate cases: for a component where all methods do not take arguments, the total number of Fuz test cases stays zero.

Even for Robut, distinct test cases trigger identical exceptions, so better test selection techniques could be applied to reduce the number of redundant test cases. Another possible approach is to store the runtime status of the component under testing, and to reuse it in the next test run, if possible. In this way, an execution of a sequence of method calls can be used in multiple test cases which share a single prefix of the transition path.

6. Related work and conclusions

This section summarises the related work, draws our conclusions, and discusses the future work.

6.1. Related work

There is a large body of literature in software testing, and much of it is on formal methods of model-based testing. In what follows, we discuss three categories of related work: testing of component software, state machine based testing, and robustness testing.

Component-based testing. Challenges for testing component software come from the limited knowledge of their underlying implementation. They are usually published without source code. Black box testing based on the API might still be applied to test the functionality of single components. Harrold et al. [14] propose to distinguish testing on providers and users for component testing. They design a methodology to allow component providers to encode information such as program slicing, control dependency, and data flow, which could be used for testing on the component user's side.

Concerning interactions between components, testing is either based on models such as UML Sequence Diagrams and Collaboration Diagrams [36], or on a Component Interaction Diagram (CIG) [35]. Wu et al. [35] propose to model direct and indirect interactions between different components with CIG, and define a set of coverage criteria based upon it.

State machine based testing. Finite State Machines (FSM) are one of the most studied models for software engineering. Drusinsky [11] proposes methods of applying state charts in monitoring and model checking of industrial reactive systems. Tretmans [32] generates test cases based on labelled transition system with input and output (IOLTS). He focuses on conformance testing of verifying the functionality consistency between implementation and abstract models. Ledru et al. [20] use a regular expression bounded on parameters and operations to express the test schema, upon which exhaustive test cases are generated. The regular expressions are equivalent to the state machine models used in our approach, except that the testers have to specify the testing schema by hand. As a major representative of FSMs, UML Statemachine diagrams are widely used in model-based testing. Bouquet et al. [4] propose UML-MBT, a subset of UML 2.0 with dynamic behaviours of the classes and transitions described in OCL. The generated test cases in [4] have similar structures in our work. They set up a set of test targets, and rely on a theorem prover to find execution paths for each target. Gallagher et al. [13] use interacting state machines with class variables to generate the combined class states and component flow graph, so as to get test cases upon them. Ali et al. [1] combine collaboration diagrams and state machines, and produce a State Collaboration Test Model for testing. These approaches extend the basic UML diagrams for integration testing, but assume the components themselves are correct.

Robustness testing. Research on robustness testing of APIs has produced several tools, mainly based on the idea of random testing. Fuzz [26] and Ballista [19] study the Unix operating system and implementations of POSIX system-level operations on different platforms. JCrasher [8] tests Java programs with pre-set values for each data type. Their advantage lies in that they do not need any additional model other than the API specification. Martin et al. [24] propose the tool WebSob to test the robustness of web services described in WSDL. The tool uses JCrasher and other tools for test generation. Their targets are the API, rather than general components with a rigorous specification of the interface and contract. Compared to our approach, a disadvantage of these tools is that the units under test are stateless. It stops them from identifying problems which only occur in certain states of a component.

6.2. Conclusions

We propose a robustness testing framework for components and implement it as a prototype tool named *Robut*. Rather than testing the robustness of the whole component system, we aim at examining a single component's ability to handle hostile method calls. The framework defines the component's robustness based on a rigorous semantics of component models. *Robut* takes a component specification in the form of an rCOS contract and its Java implementation as inputs, and generates executable JUnit test cases. The test script drives the component along paths of the interaction protocol (given as a state machine in the contract), and tries to crash it with hostile method calls. The tool is applied to a benchmark of real-world components and shows its advantages compared to traditional approaches.

This paper has been substantially revised and extended from the paper published in the proceedings of the FACS'08 workshop [21]. It is now firmly based on the formal theory of component and object software, rCOS. More cases studies and experiments have been conducted and their results are presented in this paper. We have also improved the results by not only classifying the type of the exception, but also the exact code location from the code location, leading to a more fine-grained comparison of results. The use of rCOS clarifies the concepts and illustrates the separation of concerns of a component by the model of the data functionality specified in terms of pre- and post-conditions of interface methods, the data and object definitions by a class diagram, and the model of the protocol defined by a finite state machine. Therefore, our method tests both the data's functionality and the flow of control of a component. The method for test case generation can be applied to open components as well as closed components. This is due to the distinguished feature of rCOS that separates the contract of the required interface from the contract of the provided interface of a component. For the execution of the test cases, of course a closed system is still required. Finally, rCOS provides a convenient framework for making a non-robust component robust by designing an adapter without the need to touch the code of the given component.

The test case generator and the example models are available from the rCOS website (http://rcos.iist.unu.edu) for the rCOS modeller (version 1.23).

6.3. Future work

For future work, first we want to enhance the tool and evaluate it on more components. The tool's ability to handle constraints needs improvement for efficient test generation. Second, instead of testing a single component, interesting research work could be carried out in testing the robustness of a set of integrated components, or a whole component system.

We also still lack a proper metric to compare the test cases generated by the different approaches. As the total number of tests can only be a rough indicator for the quality of the tests, other valuable statistics might be the effective number of test invocations for each method, or a combination with traditional code coverage criteria.

Also, we would like to integrate the evaluation of preconditions and invariants that depend on the component state in the case where we have the full code-generation under our control and can generate boolean queries that can be evaluated while running the test cases. Testing their robustness improves confidence in the component software on the system level. Furthermore, the classification of different types of exceptions in the context of robustness as outlined in Section 3.3 could be integrated into the tool to make it more suitable to testing traditional software that makes use of exceptions beyond the scope of system runtime errors.

Acknowledgements

We would like to thank the anonymous reviewers of the FACS'08 version and of this version for their detailed comments that helped in revising the paper to its current form. This work is supported by the research projects HighQSofD, HTTS and ARV funded by the Macao Science and Technology Development Fund, and the grants from the National Natural Science Foundation of China (No. 90718014 and 60603036), STCSM No. 08510700300, and 863 of China (No. 2006AA01Z165 and 2007AA010302).

Appendix. Detailed example test suite run

```
% Output from the test suite run.
% For each method, we list the exact exceptions NOT found by each approach,
% incl. their total frequency.
====== genjscp.Cashdesk ========

Fuz (    FOUND) = 0 4 total/checked/unchecked=50/0/33
Fuz (NOT FOUND) = 0 1

java.lang.NullPointerException @ finishSale:52 cnt. 15


SF (    FOUND) = 0 0 total/checked/unchecked=108/0/0
SF (NOT FOUND) = 0 5

java.lang.NullPointerException @ cardPay:66 cnt. 1
java.lang.NullPointerException @ cardPay:67 cnt. 7
java.lang.NullPointerException @ cashPay:74 cnt. 15
java.lang.NullPointerException @ enterItem:47 cnt. 14
java.lang.NullPointerException @ finishSale:52 cnt. 15
```

```
Robut (    FOUND) = 0 5 total/checked/unchecked=615/0/52
Robut (NOT FOUND) = 0 0



====== genjscp.ArrayStack ========

Fuz (    FOUND) = 0 2 total/checked/unchecked=10/0/10
Fuz (NOT FOUND) = 0 4

java.util.EmptyStackException @ peek:90 cnt. 15
java.util.EmptyStackException @ pop:123 cnt. 15
org.apache.commons.collections.BufferUnderflowException @ get:176 cnt. 15
org.apache.commons.collections.BufferUnderflowException @ remove:190 cnt. 15


SF (    FOUND) = 0 2 total/checked/unchecked=54/0/12
SF (NOT FOUND) = 0 4

java.lang.IndexOutOfBoundsException @ RangeCheck:546 cnt. 6
java.util.EmptyStackException @ pop:123 cnt. 15
org.apache.commons.collections.BufferUnderflowException @ get:176 cnt. 15
org.apache.commons.collections.BufferUnderflowException @ remove:190 cnt. 15


Robut (    FOUND) = 0 6 total/checked/unchecked=192/0/87
Robut (NOT FOUND) = 0 0



====== genjscp.BufferedInputStream ========

Fuz (    FOUND) = 0 2 total/checked/unchecked=40/0/17
Fuz (NOT FOUND) = 3 0
kaffe.io.IOException @ refill:352 cnt. 20
kaffe.io.IOException @ reset:307 cnt. 30
kaffe.io.IOException @ skip:326 cnt. 10



SF (    FOUND) = 0 0 total/checked/unchecked=84/0/0
SF (NOT FOUND) = 3 2
kaffe.io.IOException @ refill:352 cnt. 20
kaffe.io.IOException @ reset:307 cnt. 30
kaffe.io.IOException @ skip:326 cnt. 10

java.lang.IndexOutOfBoundsException @ read:264 cnt. 40
java.lang.NullPointerException @ read:263 cnt. 16


Robut (    FOUND) = 3 2 total/checked/unchecked=302/60/56
Robut (NOT FOUND) = 0 0



====== genjscp.BinaryHeap ========
No data for genjscp.BinaryHeap/Fuz

SF (    FOUND) = 0 0 total/checked/unchecked=114/0/0
SF (NOT FOUND) = 0 1

java.util.NoSuchElementException @ peek:249 cnt. 70


Robut (    FOUND) = 0 1 total/checked/unchecked=165/0/70
Robut (NOT FOUND) = 0 0



====== genjscp.StringBufferInputStream ========

Fuz (    FOUND) = 0 3 total/checked/unchecked=30/0/20
Fuz (NOT FOUND) = 0 0



SF (    FOUND) = 0 0 total/checked/unchecked=42/0/0
SF (NOT FOUND) = 0 3

java.lang.ArrayIndexOutOfBoundsException @ getBytes:784 cnt. 1
java.lang.ArrayIndexOutOfBoundsException @ read:143 cnt. 19
java.lang.NullPointerException @ read:142 cnt. 7


Robut (    FOUND) = 0 2 total/checked/unchecked=93/0/26
Robut (NOT FOUND) = 0 1

java.lang.ArrayIndexOutOfBoundsException @ getBytes:784 cnt. 1


====== genjscp.FastArrayList ========

Fuz (    FOUND) = 0 3 total/checked/unchecked=40/0/40
Fuz (NOT FOUND) = 0 3

java.lang.IndexOutOfBoundsException @ add:368 cnt. 103
java.lang.NullPointerException @ addAll:473 cnt. 18
java.lang.NullPointerException @ containsAll:283 cnt. 41


SF (    FOUND) = 0 3 total/checked/unchecked=198/0/137
SF (NOT FOUND) = 0 3

java.lang.ArrayIndexOutOfBoundsException @ get:323 cnt. 7
java.lang.ArrayIndexOutOfBoundsException @ remove:392 cnt. 6
java.lang.NullPointerException @ containsAll:283 cnt. 41


Robut (    FOUND) = 0 6 total/checked/unchecked=369/0/296
Robut (NOT FOUND) = 0 0



====== genjscp.BufferedWriter ========

Fuz (    FOUND) = 0 2 total/checked/unchecked=20/0/19
Fuz (NOT FOUND) = 3 0
kaffe.io.IOException @ flush:142 cnt. 10
kaffe.io.IOException @ write:175 cnt. 10
kaffe.io.IOException @ write:200 cnt. 20



SF (    FOUND) = 0 0 total/checked/unchecked=66/0/0
SF (NOT FOUND) = 3 2
kaffe.io.IOException @ flush:142 cnt. 10
kaffe.io.IOException @ write:175 cnt. 10
kaffe.io.IOException @ write:200 cnt. 20

java.lang.ArrayIndexOutOfBoundsException @ arraycopy:-2 cnt. 37
java.lang.NullPointerException @ arraycopy:-2 cnt. 14


Robut (    FOUND) = 3 2 total/checked/unchecked=151/40/51
Robut (NOT FOUND) = 0 0



====== genjscp.CurConvFlawed ========

Fuz (    FOUND) = 0 0 total/checked/unchecked=10/0/0
Fuz (NOT FOUND) = 0 1

java.lang.ArithmeticException @ convertInverse:20 cnt. 5


SF (    FOUND) = 0 0 total/checked/unchecked=30/0/0
SF (NOT FOUND) = 0 1

java.lang.ArithmeticException @ convertInverse:20 cnt. 5


Robut (    FOUND) = 0 1 total/checked/unchecked=20/0/5
Robut (NOT FOUND) = 0 0



====== genjscp.CurConv ========

Fuz (    FOUND) = 0 0 total/checked/unchecked=10/0/0
Fuz (NOT FOUND) = 0 1

java.lang.ArithmeticException @ convertInverse:29 cnt. 5


SF (    FOUND) = 0 0 total/checked/unchecked=30/0/0
SF (NOT FOUND) = 0 1

java.lang.ArithmeticException @ convertInverse:29 cnt. 5


Robut (    FOUND) = 0 1 total/checked/unchecked=20/0/5
Robut (NOT FOUND) = 0 0



====== genjscp.CircularBuffer ========

Fuz (    FOUND) = 2 2 total/checked/unchecked=40/28/8
Fuz (NOT FOUND) = 1 0
kaffe.io.BufferBrokenException @ capacity:67 cnt. 42



SF (    FOUND) = 2 0 total/checked/unchecked=60/9/0
SF (NOT FOUND) = 1 2
kaffe.io.BufferUnderflowException @ read:112 cnt. 31

java.lang.ArrayIndexOutOfBoundsException @ arraycopy:-2 cnt. 10
java.lang.NullPointerException @ arraycopy:-2 cnt. 14


Robut (    FOUND) = 3 2 total/checked/unchecked=154/99/24
Robut (NOT FOUND) = 0 0
```

References

[1] S. Ali, L.C. Briand, M.J. Rehman, H. Asghar, M.Z.Z. Iqbal, A. Nadeem, A state-based approach to integration testing based on UML models, Inf. Softw. Technol. 49 (11–12) (2007) 1087–1106.

[2] K. Beck, E. Gamma, Test-infected: programmers love writing tests, in: D. Deugo (Ed.), More Java Gems (SIGS Reference Library), Cambridge University Press, New York, NY, USA, 2000, pp. 357–376.

[3] R.V. Binder, Testing Object-Oriented Systems: Models, Patterns, and tools, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, M. Utting, A subset of precise UML for model-based testing, in: A-MOST '07: Proceedings of the 3rd International Workshop on Advances in Model-Based Testing, ACM, New York, NY, USA, 2007, pp. 95–104.

[5] CCRA. Common Criteria for Information Technology Security Evaluation, http://www.commoncriteriaportal.org/.

[6] X. Chen, J. He, Z. Liu, N. Zhan, A model of component-based programming, in: F. Arbab, M. Sirjani (Eds.), International Symposium on Fundamentals of Software Engineering, FSEN 2007, in: Lecture Notes in Computer Science, vol. 4767, Springer, 2007, pp. 191–206. UNU-IIST TR 350.

[7] Z. Chen, A.H. Hannousse, D.V. Hung, I. Knoll, X. Li, Y. Liu, Z. Liu, Q. Nan, J.C. Okika, A. P. Ravn, V. Stolz, L. Yang, N. Zhan, Modelling with relational calculus of object and component systems–rCOS, in: Rausch et al. [29], chapter 3.

[8] C. Csallner, Y. Smaragdakis, JCrasher: an automatic robustness tester for Java, Softw. Pract. Exper. 34 (11) (2004) 1025–1050.

[9] R. Davies, Practical refinement-type checking. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[10] M. Dowson, The Ariane 5 software failure, SIGSOFT Softw. Eng. Notes 84 (22) (1997).

[11] D. Drusinsky, Modeling and verification using UML statecharts: a working guide to reactive system design, Runtime Monitoring and Execution-based Model Checking. Newnes, 2006.

[12] Eclipse. Model Development Tools (MDT)/UML2 Tools, 2008. http://www.eclipse.org/modeling/mdt/?project=uml2.

[13] L. Gallagher, J. Offutt, Test sequence generation for integration testing of component software, Comput. J. (2007) bxm093.

[14] M. Harrold, D. Liang, S. Sinha, An approach to analyzing and testing component-based systems, in: Proceeding of First International ICSE Workshop on Testing Distributed Component-Based Systems, ACM, Los Angeles, CA, 1999.

[15] J. He, Z. Liu, X. Li, rCOS: a refinement calculus of object systems, Theoret. Comput. Sci. 365 (1–2) (2006) 109–142. UNU-IIST TR 322.

[16] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall, 1998.

[17] IEEE. Standard glossary of software engineering terminology, IEEE Std 610.12-1990, 10 Dec. 1990.

[18] J.D. Kindrick, J.A. Sauter, R.S. Matthews, Improving conformance and interoperability testing, StandardView 4 (1) (1996) 61–68.

[19] N.P. Kropp, P.J. Koopman, D.P. Siewiorek, Automated robustness testing of off-the-shelf software components, in: Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing, IEEE Computer Society, Washington, DC, USA, 1998, p. 230.

[20] Y. Ledru, L. du Bousquet, O. Maury, P. Bontron, Filtering TOBIAS combinatorial test suites, in: M. Wermelinger, T. Margaria (Eds.), FASE, in: Lecture Notes in Computer Science, vol. 2984, Springer, 2004, pp. 281–294.

[21] B. Lei, Z. Liu, C. Morisset, X. Li, State based robustness testing for components, in: Formal Aspects of Component Software, in: Electr. Notes in Theoret. Comput. Sci., vol. 260, Elsevier, 2010, pp. 173–188.

[22] Z. Liu, E. Kang, N. Zhan, Composition and refinement of components, in: Post event Proceedings of UTP08, in: Lecture Notes in Computer Science, Springer, 2009 (in press).

[23] Z. Liu, C. Morisset, V. Stolz, rCOS: theory and tools for component-based model driven development, in: M. Sirjani, F. Arbab (Eds.), 3rd. Intl. Symp. on Fundamentals of Software Engineering, FSEN 2009, in: Lecture Notes in Computer Science, vol. 5961, Springer, Keynote, 2009, UNU-IIST TR 406.

[24] E. Martin, Automated testing and response analysis of web services, Web Services, 2007, ICWS 2007, IEEE International Conference on, pp. 647–654, July 2007.

[25] B. Meyer, Applying 'Design by contract', Computer 25 (10) (1992) 40–51.

[26] B.P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of UNIX utilities, Commun. ACM 33 (12) (1990) 32–44.

[27] G.C. Necula, Proof-carrying code, in: POPL, 1997, pp. 106–119.

[28] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 75–84.

[29] A. Rausch, R. Reussner, R. Mirandola, F. Plasil (Eds.), The Common Component Modeling Example, in: Lecture Notes in Computer Science, vol. 5153, Springer, 2008.

[30] SUN, The Reflection API, 2008. http://java.sun.com/docs/books/tutorial/reflect/TOC.html.

[31] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1997.

[32] J. Tretmans, Test generation with inputs, outputs and repetitive quiescence, Softw. Concepts Tools 17 (3) (1996) 103–120.

[33] J. Warmer, A. Kleppe, Octopus open source project, 2006. http://www.klasse.nl/.

[34] J. Whaley, M.C. Martin, M.S. Lam, Automatic extraction of object-oriented component interfaces, SIGSOFT Softw. Eng. Notes 27 (4) (2002) 218–228.

[35] Y. Wu, D. Pan, M.-H. Chen, Techniques for testing component-based software, in: Engineering of Complex Computer Systems, IEEE International Conference on, pp. 222–232, 2001.

[36] W. Zheng, G. Bundell, Model-based software component testing: a UML-based approach. Computer and Information Science, 2007, ICIS 2007, 6th IEEE/ACIS International Conference on, pp. 891–899, 11–13 July 2007.