Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 260 (2010) 173–188
www.elsevier.com/locate/entcs

State Based Robustness Testing for Components

Bin Lei[a,b,1], Zhiming Liu[b,2], Charles Morisset[b,3], Xuandong Li[a,4]

[a] *Department of Computer Science and Technology*
Nanjing University
Nanjing, China

[b] *International Institute of Software Technology*
United Nation University
Macao, China

Abstract

Component based development allows to build software upon existing components and promises to improve software reuse and reduce costs. To gain reliability of a component based system, verification technologies such as testing can be applied to check underlying components and their composition. Conformance testing checks the consistency between the behavior and component specifications. On the other hand, robustness testing detects vulnerability of software with unexpected input or stressful environment. Existing robustness testing tools aim to crash components with preset values of different data types. But they do not take into account component states, which are vital to the detecting robustness problem of a component. We propose a state machine based approach to detect robustness problems of components. Firstly, a set of paths is generated to cover transitions of the state machine. Test inputs which follow the paths achieve high coverage of the system states and examine more transitions than stateless API testing. Secondly, invalid inputs and inopportune method calls are fed to the component in different states to test the robustness. When unexpected exceptions arise in the test runs, robustness failures are reported. We do a case study on a component from an open source software and it results in positive results.

Keywords: Robustness Testing, Component, State machine, rCOS

1 Introduction

Rather than implementing a system from scratch, component-based software development (CBSD) reuses black-box components which encapsulate internal implementations and provide functionalities through their interfaces. Various component

[1] Email:bl@iist.unu.edu
[2] Email:lzm@iist.unu.edu
[3] Email:morisset@iist.unu.edu
[4] Email:lxd@nju.edu.cn

doi:10.1016/j.entcs.2009.12.037

architectures such as EJB [20] and COM [16] are developed by different companies. Meanwhile, component models such as rCOS [3] formalize both the syntax and semantics of component systems to support design and verification.

Component-based design brings new challenges to testing. A component specifies its provided services in contracts in the form of $pre \vdash post$. When a service is called while the *pre-conditions* hold, it terminates and the *post-conditions* hold. Components are designed without being bundled to any particular client code. When they are reused in new environments, hostile usages such as illegal parameters are hard to avoid. We name usages which breaks the pre-conditions or violates the designs as *invalid inputs*. A fragile component might fail, or even crash the whole system due to such inputs. A component is *robust* when it can handle contract-breaking usages. Although being *robust* is non-functional, it is still important to test *robustness* for components, especially when they are reused in mission-critical system.

Take the following method from a calculator component for example:

$$Method : divide(x : int, y : int) : float \quad Contract : y \neq 0 \Rightarrow return = x/y$$

The contract specifies that when $y \neq 0$, the method returns the value x/y. A functional tester generates test cases in which y is not equal to 0, and checks whether the method returns result x/y as expected. However, functional testing does not check when $y = 0$, whether the component behaves reasonably *robust*. A careless programmer might ignore the *divided by zero* problem and implement a C program which terminates the call sites with a 'Floating point exception'. The tiny mistake becomes deadly when the component is reused in a mission critical system. The designer might want to fix this by refining *Contract* to *Contract'*: to return 0 when the divider is 0. In this case, functional testing covers testing of *robustness*.

$$Contract' : (y \neq 0 \Rightarrow return = x/y) \wedge (y = 0 \Rightarrow return = 0)$$

However, the component designers usually work on implementing functional requirements. It is impossible to require them to define all invalid inputs, which could be a very large subset of the complete input domain. Hence, testing for robustness can improve reliability of components.

Existing robustness testing tools such as Ballista [14] and JCrasher [5] test APIs with random and preset inputs to check whether the programs crash or not. The authors did empirical studies on Unix shell programs and Java classes respectively. They detected interesting robustness defects.

However, it is not easy to directly adapt them to test component software. The tools are designed to test APIs without taking states into account. A test case of Ballista repeats calling a single method with different parameter combinations. A JCrasher test case initializes an instance, calls a method, and restores the original state before it calls the next one. Nevertheless, components are often complex enough to have different states. Like any other bugs in software, a robustness defect is likely to be hidden unless the component under test is at a certain state, or has been through a sequence of state transitions. As a result, taking states into account can improve test efficiency for robustness testing.

On the other hand, many verification/testing techniques based on Finite State Machine (FSM) have been proposed [1][7][9][22] for traditional software as well as component software. They define coverage criteria and algorithms to generate test cases for both unit testing and integration testing. But they focus on verifying whether functional specifications are correctly implemented. To our knowledge, it is new to use state model to test robustness.

We combine these two distinct areas, and present an efficient robustness testing framework based on state machine. It drives the component along the paths of the state machine with a call sequence with valid inputs. At the end, invalid test inputs or method calls are used to try to *crash* the component. There are some unique features of this testing framework:

- The characteristic of component robustness is defined based on a semantics theory of component model. The definition guides the generation of test cases. It also helps to analyze the testing result.
- By method calls with valid inputs along the state machine, state transitions of a component are more sufficiently explored to detect more robustness defects.

The approach is presented based on Refinement of Component and Object Systems (rCOS) [3]. A prototype tool named *Robut* is implemented as an Eclipse plugin, and ported as a feature of rCOS tool suite.

The rest of the paper is organized as follows. The next section introduces components and their robustness problem. Section 3 describes the methodology of the state machine based testing framework. Experimental results and evaluation of the prototype tool are presented in Section 4. Section 5 surveys literature works. Finally we conclude the paper and describe the future work.

2 Components and Their Robustness

2.1 Interface, contract and component

The following theories from [3][10] form the foundations of our understanding towards components.

Different from object technologies in which each object represents exactly one software entity, *component* is an abstract concept. A component is syntactically a Java class in EJB [20]; while in service oriented architecture, a component is published as a service in description languages such as WSDL [12]. Despite the diversity, the following ideas for a component are widely accepted [15]: it can be reused by other software elements; it is provided with explicit usage description. Semantically, a component is a software unit with *provided interfaces* and *required interfaces*. Fig.1 is a circular buffer component, with a provided interface *CB_IF*.

Definition 2.1 An interface $I = (FDec, MDec)$, where *FDec* is a set of fields; and *MDec* declares a set of methods.

In *CB_IF.FDec*, *size* is a read-only field; in *CB_IF.MDec*, method *write* has three parameters. *off:int* is an input parameter of type *int*. The last *int* is the return

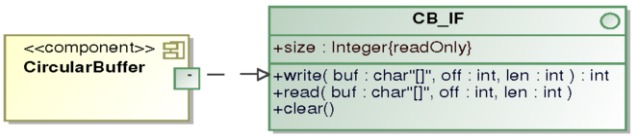

Fig. 1. Circular Buffer Component with Interface

type.

Definition 2.2 A *component* is a tuple of *C = (I, Init, MDec, MCode, PriMDec, PriMCode, InMDec)*, where

- *I* is an interface listing the provided methods, and *Init* is an initialization command;
- *MCode* (*PriMCode*) maps a public (private) method *m* in *MDec* (*PriMDec*) to a guarded command $g_m \rightarrow c_m$;
- *InMDec* is a set of required interfaces.

Definition 2.3 A *contract* is a tuple *Ctr = (I, Init, Spec, Prot)*, in which

- *I* is the interface and *Init* is the initial condition;
- *Spec* specify each method with a guarded design *g*&*D* [10];
- *Prot* is the interaction protocol the component users should follow.

A design *D* specifies a functionality provided by the interface. It is defined as $D = (\alpha, p \vdash R)$, in which α is the set of program variables; $p \vdash R$ is a predicate $ok \wedge p(x) \Rightarrow ok' \wedge R(x, x')$, which means if the program is activated in a state *ok* and the precondition *p(x)* holds, it will terminate in state *ok'* where post condition *R* holds. *R(x, x')* is a relationship of post-state *x'* and initial state *x*.

Semantics of *reactive programs* is defined by the notion of *reactive designs* with an additional Boolean observable *wait* that denotes suspension of a program. Design *D* is *reactive* if it is a fixed point of $\mathcal{H}$, i.e. $\mathcal{H}(P) = P$, where

$$\mathcal{H}(p \vdash R) =_{df} ((wait \vee p) \vdash wait') \lhd wait \rhd R$$

$$P \lhd b \rhd Q =_{df} (b \wedge P) \vee (\neg b \wedge Q)$$

We use a *guarded design g*& *D*, where *D* is a design, to specify the reactive behavior $\mathcal{H}(D) \lhd g \rhd (true \vdash wait')$, meaning that if the guard *g* is false, the program stays suspended, and if it is true, the result is $\mathcal{H}(D)$. A reactive design ensures that a synchronization of a method invocation by the environment and the execution of the method only occur when the guard is *true* and *wait* is *false*. The domain of reactive designs enjoys the same closure properties as that of sequential designs, and refinement is also defined as logical implication [10].

A state machine diagram describes *guarded designs* of a component with its state transitions. The semantics is, for the component in a certain state, only services specified in the form of the outgoing edges are available.

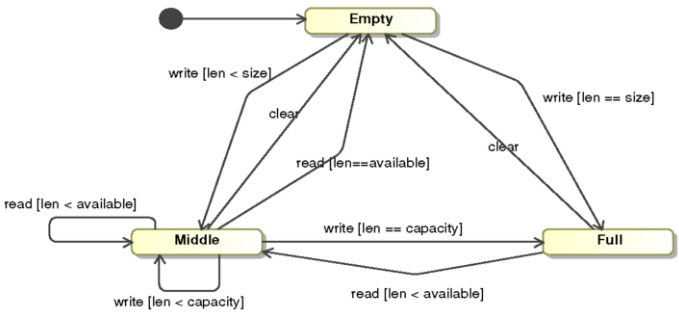

Fig. 2. Circular Buffer Statemachine

Definition 2.4 A *state machine* is a tuple $SM = (S, S_0, \Sigma, Guard, Tran)$ where

- S is a finite set of states, $S_0 \in S$ is the initial state, and $Guard$ is a finite set of guard conditions;
- Σ is the finite set of input symbols, which are public method calls in the interface, i.e. $\Sigma \subseteq SM.I.MDec$;
- $Tran \subset S \times \Sigma \times Guard \times S$, is a relation of transitions between states.

Fig.2 is a state machine of the circular buffer interface. *available* is the number of characters to be read from the buffer. *capacity* is the remaining vacancis for write operations. Here is a path ρ along the state machine diagram

$$\rho = Init \longrightarrow Empty \overset{write[len<size]}{\longrightarrow} Middle \overset{write[len==capacity]}{\longrightarrow} Full \overset{clear}{\longrightarrow} Empty$$

In this paper, we use paths of state machine to explore the state transitions of a component, so that robustness testing is more efficient.

2.2 Robustness

Intuitively, a component is robust if it can work properly without crashing, even when it is used inappropriately. In CBSD, the client code is supposed to assure the pre-conditions holds. Ideally, it is also required to handle errors/exceptions declared in the contract. However, what happens if the client code is not that responsible? A robust component can handle the hazardous usages, and restore itself to normal state. But a component which does not take robustness into account might behave unpredictably or even propagate the failure to its client code.

IEEE defines robustness [13] as: the degree[5] to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. We focus here on invalid inputs, which are more related to component design. From the perspective of a *contract*, *invalid method calls* are:

iv1 Method calls with inputs which are against the pre-conditions;

[5] From perspective of testing, standards for *degree* of robustness may vary according to different testing scenarios: it can be quantified by defect ratio in different overload for stress testing; or by failure frequency for server systems. We do not provide such evaluation because the testing framework aims to be a process to reveal defects.

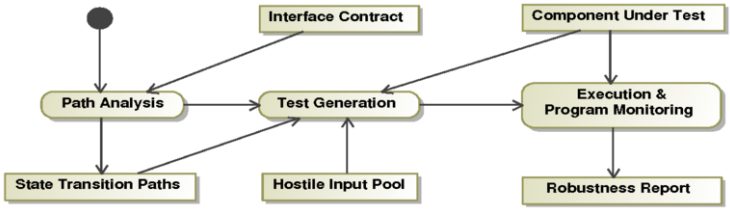

Fig. 3. Framework of Robustness Testing

iv2 Inopportune method calls, which correspond to transitions not specified on the state machine, i.e. request of services when they are not available;

Definition 2.5 For a state machine SM and state $s \in SM.S$, a method m should be called if and only if

$$\exists s' \in SM.S, g \in SM.Guard,\ g \wedge ((s, g, m, s') \in SM.Tran)$$

which means at least one of the outgoing edges is specified as $g\&m$ and g is satisfied. A method call is *inopportune* when none of the outgoing edges satisfies the above condition.

Definition 2.6 A component C is *robust* when $\forall m \in C.MDec$, its guarded command $g_m \rightarrow c_m$ implements a *robust guarded design* of the interface, i.e.

$$r(g\&D) = (\alpha, ((true \vdash post) \triangleleft pre \triangleright \neg crash) \triangleleft g \triangleright \neg crash)$$

which means, neither invalid input values (iv1) nor inopportune calls (iv2) could put the component into *crash* state.

Crash is a tester-defined behavior, in which the component becomes invalid and probably propagate the problem to its call site. It is not formally defined here for two reasons: the severity of the same software failure varies when different reliability is expected; and it is dependent to implementation platforms.

Crash should be refined to concrete phenomenons on different platforms. For instance, in Java, an unhandled exception is thrown to upper levels of the stack frames until it was caught in one level. Both the component and its call site are suspended. Detailed mapping from the *crash* designs to Java is given in Section 3.3.

3 Methodology

In this section we describe the details of the testing framework. For a component under test, we first analyze the paths of its state machine based on transition coverage. *Valid call sequences* are generated to reach different states along the paths. Invalid inputs and inopportune calls are appended to them, and form *robustness test cases*. The tool distinguishes the different exceptions and report robustness problems. The activity diagram in Fig.3 shows the overall work flow. Square boxes are artifacts used or produced in the testing process, while round boxes stands for the testing activities.

```
class B extends A  ...
class C{
    A a = null;
    boolean state = false;
    void init(){
        if(!state) a =new A() ;
        else a = new B() ;
    }
    public void evolve(){ state = true ; }
    public B cast(){
        if(!state) return null;
        else return (B)a ;
    }
}
```

Call sequences of a C instance:
- √ cast();
- √ init();
- √ init(); cast();
- √ evolve(); cast();
- √ evolve(); init(); cast();
- × init(); evolve(); cast();

×: *ClassCastException* thrown
√: pass

Fig. 4. Example of a Robustness Defect Deeply Hidden

3.1 Path analysis

Each robustness flaw corresponds to certain triggering conditions related to component states. For instance, in Java, a *ClassCastException* is thrown when a statement in the form of *B b = (B) a;* try to cast object *a* into an instance of class *B*, which *a* is actually not. In Fig.4, the component *C* provides several interfaces: *init()* assign a new instance to *a* according to its state; *evolve()* changes the original state variable *state* from *false* to *true*; *cast()* is a service to get *a's* reference. On the right column, a set of call sequences is used to test the component. It is identified that only the following call sequence fails:*init, evolve, cast*. Note that an API based robustness testing cannot detect it, since no exception will be thrown from the initial state no matter which single method is called.

To reveal such defects, we traverse the state machine diagram to get a certain coverage of the nodes and transitions so the state transitions can be explored deeper. Coverage criteria for state machine includes *All-Node*, *All-Transitions* [2], *Modified Condition/Decision Coverage (MCDC)* [4] and variants of them. All-Transition is selected for this testing framework. It ensures that all reachable states and transitions of the state machine are covered. Compared to other flow based criteria such as the popular *MCDC* coverage, *All-transitions* avoids analyzing the predicates and keeps the path analysis algorithm simple. Fig.5 is a Breadth-First-Search(BFS) algorithm we implemented in the prototype tool. In the result set of diagram analysis of the state machine, a path ρ is in the form of

$$Init \longrightarrow S_0 \overset{m_1[g_1]}{\longrightarrow} S_1 \cdots \overset{m_n[g_n]}{\longrightarrow} S_n$$

Each m_x corresponds to a public method declaration with signature in the component interface; while g_x stands for a guard condition which enables the transition. Guard conditions are predicates on the state variables.

```
//Input: SM = (S, S0, Σ, Guard, Tran);
//Output: PathList, a list of generated paths, each of which is a sequence
of states and transitions;
//Intermediate data: vq, a queue used to store visited states;
begin
    vq.add(S0);
    while(vq is not empty){
        Get vq's head vertex sv;
        for(each Transition t of sv's outgoing edges) {
            updateTransition(t);
            add target vertices of t to tail of vq;
        }
    }
    for(each transition t of the statemachine)
        if(t.from is reachable but t is not covered by PathList){
            find a Path p which ends at t.from;
            create a new Path p' = p.append(t.from).append(t);
            PathList.add(p);
        }
end
Function updateTransition(Transition t){
    if(t is from initial state)
        create new Path p from init;
    else{
        find a Path p which ends at t.from;
        create a new Path p' = p.append(t.from).append(t);
    }
    PathList.add(p);
}
```

Fig. 5. Algorithm of All-Transition Coverage Traverse

3.2 Stateful robustness test cases

With the paths we generate robustness test cases which aim at triggering robustness failure. A *robustness testcase* consists of the following call sequence:

- A sequence of *valid method calls*, which changes the state along the path from *Init*;
- An *invalid method calls*, either *iv1* or *iv2* as defined in Section 2.2.

We implement the test framework for components implemented in Java and the following test case generation mechanism assumes such testing targets. Nevertheless, the approach can be ported to other platforms. Test inputs are generated based on the method calls on the paths. For a method call, an concrete parameter is

generated to replace each formal parameter. Instead of creating serializable objects and restore them at runtime, we generate Java statements which creates them. Our approach avoid managing any external object base. A Java metamodel is adopted from Octopus [23]. An *OJOperation* instance is constructed for each method call, and an *OJParameter* for each parameter. The objects are then translated JUnit test cases.

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;
- if T is non-primitive and method $T(T_1, T_2, ...T_n)$ is its constructor, return $T(generate(T_1), generate(T_2), ...generate(T_n))$.
  For the Java metamodel, a list of statements constructing $T_1, T_2, ..T_n$ respectively are inserted in front of the statement of calling T's constructor. When T has multiple constructors, a random one is chosen.

The recursion stops when it reaches a primitive type. We also limit the recursion depth to be less than 5, to avoid possible infinite loop and make the experiment more practical. When a construction reaches 5 levels deep an *null* reference is returned. This limit is adapted by our experiment experience.

When a parameter is bounded by pre-conditions, the random generator narrows the sample space accordingly. In rCOS, OCL constraints can be applied to notate a transition in state machine. Our prototype tool abstract OCL notations and use them for effective random generation. For instance, in the *divide* example in introduction, the pre-condition is $y \neq 0$, so the random generator pick a random value in [MIN_INT, 0) and (0, MAX_INT]. [6]

To test robustness, an *invalid method call* should be provided at the final step. For *invalid inputs*(*iv1*), parameter lists are generated the same way as we do for valid calls, but with at least one parameter replaced with an *invalid value* from a pool for corresponding type described below. Suspicious values which often cause problems are also included. Note that static type checking and refinement typing[6] could make it impossible to feed some invalid inputs.

- For primitive parameters, a pool of invalid values is constructed manually. For example, $invalid.String = \{null, empty\_string, very\ long\ string\}$; $invalid.int = \{MIN\_INT, -1, 0, 1, MAX\_INT\}$.
- For primitive parameters with OCL constraints, random values against the predicates are also included in the pool.
- For non-primitive parameters, the pool contains *null* and 1-depth instance constructed with either *null* object or invalid primitive value.

The second type of error-prone calls are *inopportune calls*(*iv2*). Based on definition 2.5, for a state s in state machine SM, and a set of transitions from s, say

[6] OCL constraints can be very complex. We take simple ones for test generation. An enhanced random generator based on complex OCL constraints can be an interesting future work.

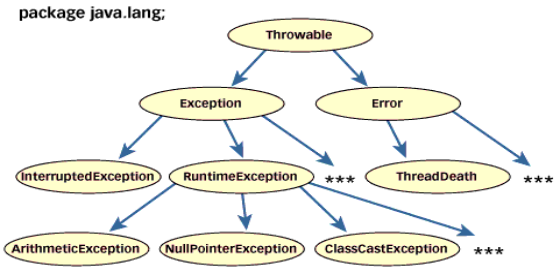

Fig. 6. Hierarchy of Java Exceptions

out, the inopportune calls have two parts:

set1 $\{true\&m \mid (m \in SM.\Sigma) \wedge (\nexists g \in SM.Guard, g\&m \in out)\}$, is the set of method calls not specified as being called from s in SM;

set2 $\{(\neg \bigvee(\Phi(m)))\&m \mid \exists g \in SM.Guard, g\&m \in out\}$, in which $\Phi(m) = \{g \mid g\&m \in out\}$, $\bigvee(\Phi(m))$ is the disjunction of elements in $\Phi(m)$, *set2* is the set of methods which are specified, but called when none of the guard conditions holds.

We implemented *set1*, the methods not specified in the outgoing edges. Implementation of *set2* requires runtime monitoring of the system variables in the guards. A enhanced version of the prototype supporting on-the-fly generation for inopportune calls is under development.

3.3 Result Analysis

We use Java's built-in exception handling mechanism to decide whether a component passes the robustness testing or not. The exceptions are classified and the component *crash* is mapped to different categories of them.

In Java, *Throwable* is the base interface for all *exceptions* and *errors* which can be thrown. Besides programmer-defined exceptions, Java has built-in exceptions such as *ArrayIndexOutOfBoundsException, NullPointerException*, .etc. The class hierarchy of Java exceptions is shown in Fig.6 and explained below:

- *java.lang.Error*. It is thrown by the virtual machine and indicates serious problems. The programmers are not supposed to *throw* or *catch* them in implementation. For example, an *OutOfMemoryError* causes hang of the thread.
- *unchecked* exceptions, subclasses of *RunTimeException*. Programmers can throw an *unchecked* exception without declaring it in the *throws* list. An *unchecked* exception might indicate a broken contract. For example, an *ArrayIndexOutOfBoundsException* is thrown when *beginIndex* is negative for *String.substring(int beginIndex, int endIndex)*. It could also be due to a bug in the component.
- *checked* exceptions, other subclasses of *Exception*. Those exceptions are supposed to be caught by the client code to indicate errors or transfer control flow.

The test oracle for component robustness is based on exceptions. We analyze the causes of different exceptions and refine *crash* to certain subgroups of them. The

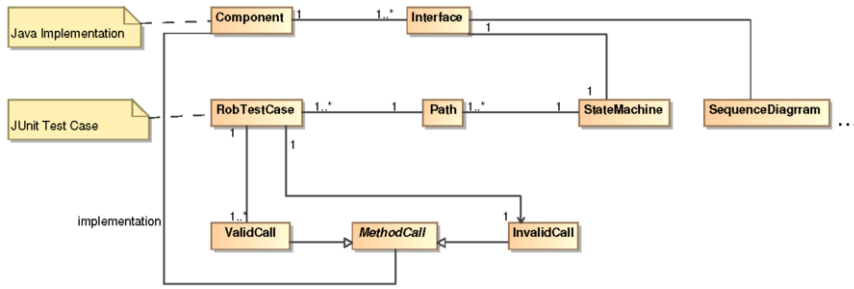

Fig. 7. Artifacts of *Robut* Test Case Generation

oracle reports failure when it spots the following phenomenons in a Java application:

- *Unchecked* exceptions thrown to test script without being declared. This will cause the client code to hang, and propagate the exception to call stacks. Instead, a *robust* component would catch the unchecked exceptions and return an information message to its client code. An *unchecked* exception might be due to either erroneous parameter from client code or the component implementation. During development, the programmer should try to expose all of them to test script. But components shipped as products are supposed to catch them and provide mechanisms for client code to understand the situation, rather than rudely terminate the process.
- User-defined *checked* exceptions, which indicates the component is in wrong state and not performing its functionality. We provide a customization point for testers to add component-defined exceptions to indicates an erroneous state of the component. This is similar as returning a positive integer in C functions to indicate certain error in the method call.

Any reported failure indicates that the component *crashes*. The stack trace of this exception pinpoints the call hierarchy how robustness defects are triggered. Corresponding test case and the path of state transitions are available for reproducing and debugging the problem.

4 Tool Implementation and Case study

4.1 Tool sketch

To validate the approach, we implement the testing methodology in a prototype tool named as *Robut*. It consists of the following components: *path analyzer*, which explore a state machine to get valid test paths; *test case generator*, to compose executable test cases; and an exception classifier to produce test report.

The artifacts during test generation are summarized in the class diagram shown in Fig.7. The center of the component framework is a provided interface, which corresponds to the published services. On the left is its component implemented in Java. On the right are specifications in UML diagrams. Our work focus on generating robustness test cases based on state machines.

The prototype is built upon a set of libraries. EMF/UML2 [8] and rCOS [3] are used to model the components and their interfaces. Dresden [11] provides OCL parser. Java reflection [21] and Octopus's Java Metamodel [23] are used extensively to generate test inputs.

4.2 Case study: Circular Buffer

As a case study, we consider implementation of a circular buffer for characters with slight adaptation from Kaffe Java virtual machine [19]. As in Fig.1 and Fig.2, the component provides a buffer allowing client code to *write*, *read* or *clear* the buffer without worrying about the underlying storage. The state machine contains 3 states besides initial state, and 9 transitions between them. The implementation of the buffer is based on two internal pointers *in*, *out*, and a character array. But client code developers only have to prepare the parameter list and call the public methods. The *checked exception*s declared by this components include: *BufferBrokenException*, which implies the buffer is corrupted; *BufferOverflowException*, thrown when the client code tries to write too many characters; and *BufferUnderflowException* which occurs under the opposite condition.

Robut generates 10 paths to cover all the transitions. Invalid inputs are generated for each state. For *Circular Buffer*, the number of invalid inputs generated for the states are shown in Table 1. The combination of valid paths and invalid inputs

State	Ocl	Null	Preset	Inopportune	Overall
Empty	2	2	4	2	10
Middle	4	4	8	0	16
Full	1	1	2	1	5

Table 1
Invalid Inputs for Circular Buffer

forms 114 complete robustness test cases. We test the circular buffer with generated test cases and observed two types of *crash*, caused by a user-defined *BufferBrokenException* and unchecked exceptions such as *NullPointerException* and *ArrayIndexOutOfBoundsException*. Debugging and manual inspection of the code reveal the root defects:

We analyze the unchecked exceptions and find several code segments which optimistically assume the parameters to be valid. For example, the *NullPointerExceptions* traces back to line 100 of CircularBuffer.java, where the following statement tries to copy a null buffer, without any safety checking.

$$System.arraycopy(buffer, out, buf, off, maxreadlen);$$

The *BufferBrokenException* reveals another bug inside the component. The stack traces to line 58 in an internal method *available*(), which provides the information of how many characters are available for reading in the buffer. When the two pointers, *in* and *out*, are inverted, the method returns a value larger than the

buffer size. With the false value, the client code is able to override in existing values without getting any complaint from the component.

4.3 Evaluation

To compare with existing approaches, we also implement fuzzy robustness testing[5], and an algorithm to do functional testing with only valid inputs.

- For the former, random string and int value are generated and fed to stateless buffer which are newly initialized. The fuzzy testing detects the *NullPointerException*. However, the other exceptions are never thrown because the testing always starts on an empty buffer.
- For stateful testing, valid test inputs along test paths are used. When the path is long enough, it can spot the *invert pointer* problem. But it does not test the component's ability to bear invalid inputs, because it is aimed for conformance testing.

By combining the two approach, our prototype tool produces encouraging results for component robustness testing.

However, there are also concerns to be solved. The first concern is when a component has many states, the result test suite could be very large. Test selection techniques could be applied before testing. Secondly, model availability is also a key concern. If the component under test was not designed by rigorous model based engineering process, the tester should manually generate the model, rather than the other way around. That process is both expensive and error-prone. The good thing is, UML is gaining more popularity in both academia and industry. Platforms such as Eclipse and Rational Software Architects supports design and coding in a coherent software life cycle. The widely adopted tool chains help improve availability of component models. On the other hand, feedback based random testing[18] and model synthesis techniques[24] are emerging to provide the means to test without explicit models. We plan to investigate them for robustness testing for future work.

The tool assumes existence of Java byte code of the components, which are used by Java reflection. This limitation could hinder applying this approach to components published with interface only. It can be mitigated by manually adding information about how to get objects from certain class. This information is always available to call sites.

The prototype automates the test case generation and test execution. A JUnit test script which specifies UML2 model file name and interface class name is enough for test generation. The test oracle is implicit, in the sense that a test run fails when JUnit catch an exception. Manual effort has to be spent on customizing user-defined exceptions, which improve the tool's ability of detecting faults. Nevertheless, they are not mandatory. Porting the framework to another platform requires rewriting two parts: the mapping of *crash* to concrete program phenomenons; and test script generation module.

The user customization for *crash* breaks the soundness of our approach. Except for this, the approach is sound in that each unhandled *checked* exception indicates

a robustness defect inside the software. 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.

5 Related Work

In this section, we survey the related works, which are attributed to three categories: testing of component software, state machine based testing and robustness testing.

Challenges for testing component software comes from the limited knowledge and of their underlying implementation. They are usually published without source code and the development platform might be different from the client code. Black box testing based on the API might still be applied to test the functionality of single components. For the interactions between components, testing are either based on models such as UML Sequence Diagram and Collaboration Diagram [26], or on component interaction diagram [25]. Finite State Machine (FSM) is one of the most studied models for software engineering. Doron et al. [7] proposes methods of applying state charts in monitoring and model checking industrial reactive systems. Jan et al. [22] generate test cases based on labeled transition system with input and output(IOLTS). They focus on conformance testing of verifying the functionality consistency between implementation and abstract models.

State machine is also used for integration testing of component based software. Gallagher et al. [9] use interacting state machines with class variables to generate combined class states and component flow graph, so as to get test cases upon them. Ali et al. [1] combines collaboration diagram and state machines, and produces State Collaboration Test Model for testing. Those approaches extend basic UML diagrams for integration testing, but assume the components themselves are correct. The latter is the focus of our work.

On the other hand, research on robustness testing for API has produced several tools, mainly based on the idea of random testing. Fuzz [17] and Ballista [14] conduct study on Unix and POSIX implementation. JCrasher [5] tests Java programs with pre-set values for each data type. Their advantage lies in that they do not need additional model other than API specification. Their targets are API, rather than general components with rigorous specification of 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 occurs in certain states of a component.

6 Conclusion and Future Work

We propose a robustness testing framework for component and implement it as a prototype tool named *Robut*. Rather than testing robustness of the whole component system, we aim at examining a single component's ability to handle invalid inputs and inopportune calls. The framework defines component robustness based on rigorous semantics of component model. *Robut* takes component specification

and its Java implementation as inputs, and generates executable JUnit test cases. The test script drives the component through paths of the state machine, and try to crash it at the end with invalid inputs. The tool is applied to a real world component and has revealed several defects.

For future work, firstly we want to enhance the tool and evaluate it on more components. We plan to import open source libraries into rCOS framework and perform robustness testing on them. The tool's ability to handle constraints needs improvement for more efficient test generation. Secondly, instead of testing a single component, interesting research work could be carried out in testing robustness of a set of integrated components, or a whole component system. Composition of the components could be specified in certain modeling language. Interaction protocols are also prone to robustness defect. Testing their robustness improves confidence over component software in a system level.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China (No. 60603036 and No. 60673114) and 863 of China (No. 2006AA01Z165 and No.2007AA010302). It is also supported by HighQSofD and HTTS funded by Macao Science and Technology Fund.

References

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

[2] Binder, R. V., "Testing object-oriented systems: models, patterns, and tools," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] Chen, Z., Z. Liu, A. Ravn, V. Stolz and N. Zhan, *Refinement and verification in component-based model driven design*, Research Report 388, International Instititute for Software Technology, United Nation University, P.O.Box 3058, Macau (2007).

[4] Chilenski, J. and S. Miller, *Applicability of modified condition/decision coverage to software testing*, Software Engineering Journal **9** (Sep 1994), pp. 193–200.

[5] Csallner, C. and Y. Smaragdakis, *Jcrasher: an automatic robustness tester for java*, Softw. Pract. Exper. **34** (2004), pp. 1025–1050.

[6] Davies, R., "Practical refinement-type checking," Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2005), chair-Frank Pfenning.

[7] Drusinsky, D., "Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking," Newnes, 2006.

[8] Eclipse, *Model development tools (mdt) / uml2 tools* (2008).
URL http://www.eclipse.org/modeling/mdt/?project=uml2

[9] Gallagher, L. and J. Offutt, *Test Sequence Generation For Integration Testing Of Component Software*, The Computer Journal (2007), p. bxm093.

[10] He, J., X. Li and Z. Liu, *A theory of reactive components*, Electr. Notes Theor. Comput. Sci. **160** (2006), pp. 173–195.

[11] Hussmann, H., B. Demuth and F. Finger, *Modular architecture for a toolset supporting ocl*, Sci. Comput. Program. **44** (2002), pp. 51–69.

[12] IBM, *Web services architecture overview* (2000).
URL "http://www.ibm.com/developerworks/webservices/library/w-ovr/"

[13] IEEE, *Ieee standard glossary of software engineering terminology*, IEEE Std 610.12-1990 (10 Dec 1990).

[14] Kropp, N. P., P. J. Koopman and D. P. Siewiorek, *Automated robustness testing of off-the-shelf software components*, in: *Proceedings of the The 28th Annual International Symposium on Fault-Tolerant Computing* (1998), p. 230.

[15] Meyer, B., *The grand challenge of trusted components*, in: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (2003), pp. 660–667.

[16] Microsoft, *Com: Component object model technologies* (2008).
URL "http://www.microsoft.com/com/default.mspx"

[17] Miller, B. P., L. Fredriksen and B. So, *An empirical study of the reliability of unix utilities*, Commun. ACM **33** (1990), pp. 32–44.

[18] Pacheco, C., S. K. Lahiri, M. D. Ernst and T. Ball, *Feedback-directed random test generation*, in: *ICSE '07: Proceedings of the 29th international conference on Software Engineering* (2007), pp. 75–84.

[19] Pick, J., *Kaffe java virtual machine* (2008).
URL "http://www.kaffe.org/"

[20] SUN, *Enterprise javabeans technology* (2008).
URL "http://java.sun.com/products/ejb/"

[21] SUN, *The reflection api* (2008).
URL "http://java.sun.com/docs/books/tutorial/reflect/TOC.html"

[22] Tretmans, J., *Test Generation with Inputs, Outputs and Repetitive Quiescence*, Software—Concepts and Tools **17** (1996), pp. 103–120.

[23] Warmer, J. and A. Kleppe, *Octopus open source project* (2006).
URL "http://www.klasse.nl/index.html"

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

[25] Wu, Y., D. Pan and M.-H. Chen, *Techniques for testing component-based software*, iceccs **00** (2001), p. 0222.

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