Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 236 (2009) 101–115
www.elsevier.com/locate/entcs

Testing-based Process for Evaluating Component Replaceability

Andres Flores[1]

GIISCo Research Group
Department of Computer Science
University of Comahue
Neuquen, Argentina

Macario Polo[2]

Alarcos Research Group
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Ciudad Real, Spain

Abstract

Replacing pieces of component-based systems carries a serious risk on the expected stability. Substitutability of components must then be carefully identified. With this intent, this paper presents a process to evaluate replacement components by complementing the conventional compatibility analysis with component testing criteria. Functions of data transformation encapsulated by components (i.e. their behaviour) are analysed according to the Observability testing metric. For a component under substitution, a Component Behaviour Test Suite is built to be later applied on candidate replacement components. This approach is also known as Back-to-Back testing. The whole process is currently supported through the tool *testooj*, which is focused on testing Java components.

Keywords: component-based software engineering, substitutability, upgrade, component testing

1 Introduction

Maintenance of component-based systems involves replacing existing pieces with upgrades or new components. This implies a serious risk on the stability of functioning systems [15,27]. Substitutability is then an important challenge due to the evolutive nature of software and the impact of changes. Whether there can be a certain control on versions of a component, under successive releases changes may spread across most of the codified functions and structures producing a massive

[1] Email: aflores@uncoma.edu.ar
[2] Email: macario.polo@uclm.es

doi:10.1016/j.entcs.2009.03.017

difference with respect to the original component. This is even harder when components are acquired from different vendors, where system integrators cannot control deployment, and cannot be sure if the same environment (e.g. compiler, and compiling options) were used on components that are supposed to be alike. This even applies to successive releases [20,4].

The main concern for an integrator is therefore identifying if new releases or new acquired components can safely replace pieces from a component-based system already deployed and in-use. With that intent, this paper presents a Process for Evaluating Component Replaceability. The proposal complements the conventional compatibility analysis by means of black box testing coverage criteria. The central idea is to observe the operational behaviour of a component (i.e. its output as a function of its input), which is reflected by the *observability* testing metric [10,16].

To address this approach, specific testing coverage criteria have been selected in order to design an adequate Test Suite (TS) as a representation of behaviour for components, viz. a Component Behaviour Test Suite. Such TS is developed for the piece under substitution, to be later exercised on candidate replacements to observe behaviour equivalence.

Automation of the whole process is currently supported for the Java framework through a tool, *testooj* [26], from where Test Case generation is done rigorously through automated steps and conditions. The tool additionally integrates well-known testing frameworks like JUnit and MuJava [17,22], from where the Component Behaviour TS is easily validated in the development phase and later effectively executed against candidate components to analyse compatibility. A .Net version of the same tool has been partially implemented as well, and will be updated to include support for the remainder phases of the process.

The paper is organised as follows. Section 2 presents an overview of the whole approach. Section 3 describes aspects of the Component Behaviour TS. Section 4 presents the evaluation of Interface Compatibility which is done at a syntactic level before the testing-based evaluation. Section 5 describes the Testing-based Behaviour Compatibility analysis. Section 6 presents results of an experiment. Section 7 presents some related work. Conclusions and future work are presented afterwards.

2 Process for Evaluating Replaceability

Our proposal consists of three main phases, which are depicted in Figure 1. Being an original component C and a candidate replacement K, the whole process involves the following:

1st Phase. A TS is generated with the purpose to represent behavioural aspects of a component C. This TS complies with certain criteria which help describing different facets of interactions of component C with others components into a software system. Notice that the goal of such TS is not to find faults but to represent behaviour. This will be fully explained in Section 3.

2nd Phase. Interfaces offered by C and the candidate K are compared syntactically. At this stage, there can be compatibility even though services from C and K have

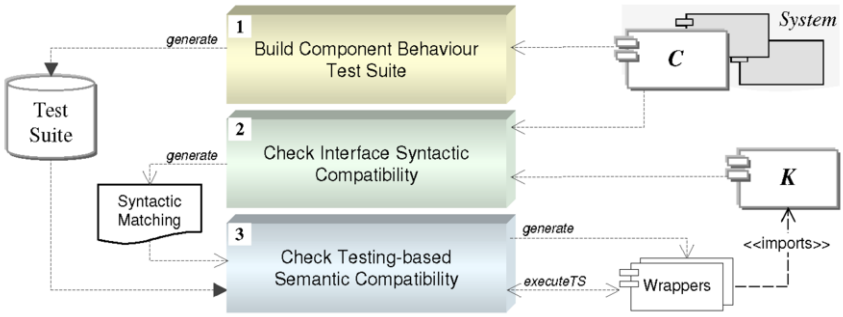


Fig. 1. Testing-based Process for Evaluating Replaceability

different names, different order in the parameters, etc. The outcome of this phase is a matching list where each service from C may have a correspondence with one or more services from K. See details in Section 4.

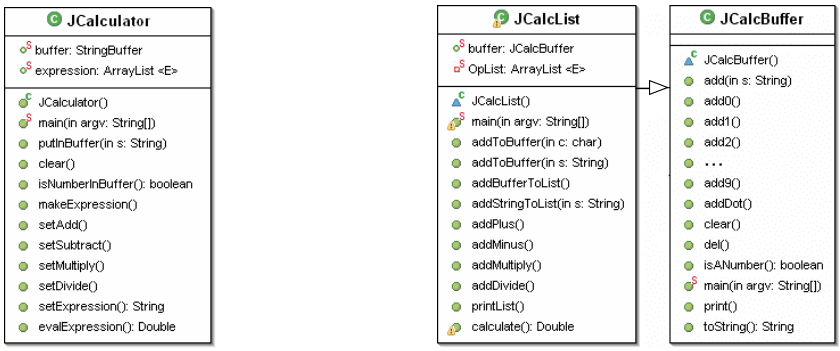
3rd Phase. Component K which has passed the interface compatibility must be evaluated at a semantic level. This implies to execute the TS built for C (in the first phase) against K. If the previous phase discovers a complete inclusion of C interface into the K component, then the TS is directly applied on K. Otherwise, there is a need to identify the true service correspondences from the list obtained in the second phase. Hence, from that list a set of wrappers (W) is generated for K. Then, each $w \in W$ is taken at a time as the target class under test by running the TS from C. After the whole set W has been tested then results from each execution are analysed to conclude if a compatibility has been found. This may also imply that a wrapper $w \in W$ could be selected as the most suitable to allow tailoring K to be integrated into the system as a replacement for C. Full details are given in Section 5

The approach can also be understood from the point of view of the technique called Back-to-Back testing, which makes use of a reference implementation for a component (i.e. C) to generate a TS to exercise both a unit under test (i.e. K) and the reference implementation. Then results from the reference component help to judge the correctness of the unit under test [8].

Next sections provide detailed information of each phase of the process, which will be illustrated by means of a case study presented as follows.

Case Study

The approach proposed is illustrated by a small case study. A Java calculator, `JCalc`, which can be downloaded from *http://sourceforge.net*, and whose main classes are shown in Figure 2(b). A new component called `JCalculator` has been created as a variation from `JCalc`, as can be seen in Figure 2(a). For illustrative purposes, `JCalculator` will be considered as the original component, which makes `JCalc` become a candidate replacement. The following sections explain how the Process to Evaluate Replaceability is applied to give a conclusive decision on compatibility between `JCalculator` and `JCalc`.



(a) JCalculator (b) JCalc

Fig. 2. (a) Original component C – (b) Candidate replacement K

3 Component Behaviour Test Suite

Given a component C and a candidate replacement K, each one accepts a certain input on their services, from where an internal transformation function returns a specific output. Such correspondence input-output is called *functional mapping* and is particularly reflected by the observability testing metric [10,16]. Analysing functional mappings can be used to expose a potential compatibility between components – as discussed in [1,4]. Although a deep analysis could be extensive, focusing on certain aspects and representative data result more efficient and is also highly effective. This is basically addressed through a specific selection of testing coverage criteria in order to build a TS as a behavioural representation of components.

The goal of this TS is to check that a candidate component K coincides on behaviour with a given original component C. Therefore, each test case in TS will consist of a set of calls to services of C, from where the testing results are saved in a repository for determining acceptance or refusal when the TS is applied against component K. Following we list some relevant component coverage notions, to then explain the strategy for their implementation on the approach.

- *all–methods* [12]. It is required that every method (or service) from a component interface must be invoked at least once. This is called *all-interfaces* in [16,29].
- *all–events* [16]. An event is an incident where the effect is the invocation of an interface. Events can be synchronous (e.g. direct calls to services) or asynchronous (e.g. triggering exceptions) [28]. It is required that every event must be covered by some test. Thus this criterion covers *all-exceptions* described in [12,29].
- *all–context-dependence* [16]. Events can have sequential dependencies on each other causing distinct behaviours according to the order in which they (i.e. services or exceptions) are called. The criterion requires to traverse each operational sequence at least once.

Two cases apply for the last criterion: *intra-* or *inter-component* dependence. That is, dependence either to events inside the same component or to external events (from other components). Inter-component context dependence requires to design tests with a client and a server component [30].

Our Component Behaviour TS regards intra-component dependence to ease evaluating components without extra environmental requisites. Yet there could be a concern with respect to exploring actual inter-operation between client-server components. Though, expressing potential sequences of events expose likely interactions with any client component. Therefore, the approach is still effective and also general enough to wider its applicability.

In order to describe sequential dependencies of events, we make use of *regular expressions* (RegEx), where the alphabet is comprised of signatures from components services. This helps to describe a general pattern referred to as the "*protocol of use*" for a component interface [18,24]. Specific coverage criteria for RegEx have been proposed in [21], from where the relation with the component coverage criteria presented above was analysed in a previous work [9]. Figure 3 shows such a relation to clear why RegEx are an adequate implementation strategy on this approach. Since operational sequences can also be derived from Finite State Machines (FSM) [3,24,18] and FSMs can be actually represented by RegEx, then equivalence or subsumes relations can be found on criteria from both notations. Such a relation is also explained in [9] and is depicted on Figure 3.

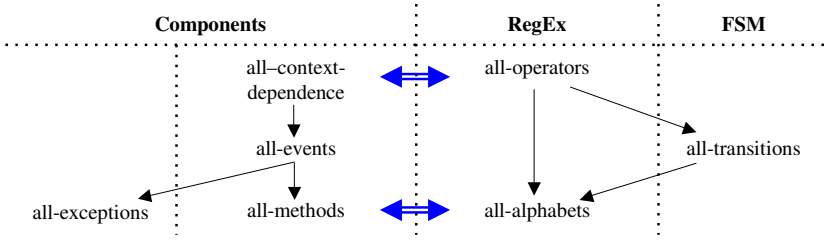


Fig. 3. Subsumes relation among testing criteria

The reflection mechanism of the Java framework allows to extract elements from a component interface to be able to automate Test Case generation. Thus, we may count with service signatures for the alphabet of RegEx. Also, exceptions extracted from services help to strengthen the representation of components behaviour, which then are used to satisfy the *all–exceptions* criterion. In this way, the RegEx based approach is properly complemented to achieve the *all–context-dependence* criterion. In fact, some exceptions require the component being in a specific state only reachable after previous executions of other events (e.g. invocations to certain services), therefore operational sequences (context-dependence) are usually the only strategy to get a proper coverage.

For our Java calculator case study (cf. Section 2), it is following explained how the procedure to build the Component Behaviour TS is carried out, and how it deals with the analysis concerning coverage criteria.

Test Suite for JCalculator

To build a Component Behaviour TS for `JCalculator`, some steps supported by the *testooj* tool [26] must be done, as depicted on Figure 4. Test cases can be generated in two formats: JUnit and MuJava [17,22]. Initially the TS is generated

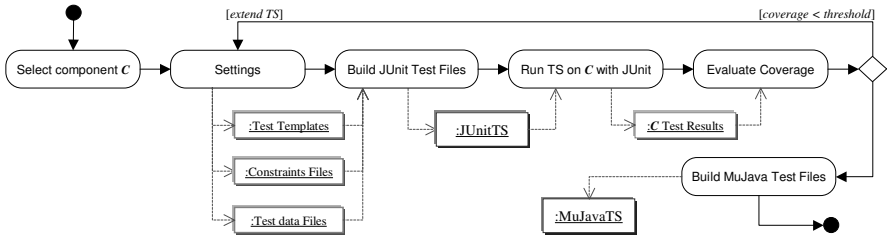


Fig. 4. Generation of Component Behaviour Test Suite

on JUnit format to be validated by its execution against the original component (`JCalculator`). The need is to get 100% successful results since the TS is designed to have configurations of test cases that either do not fail or raise controlled exceptions. Only thus, the TS achieves the goal of representing behavioural facets of the original component. After the TS has been properly validated, then a TS on MuJava format will be derived to be used on the third phase of the process, to ease the involved analysis tasks – this is fully explained in Section 5.

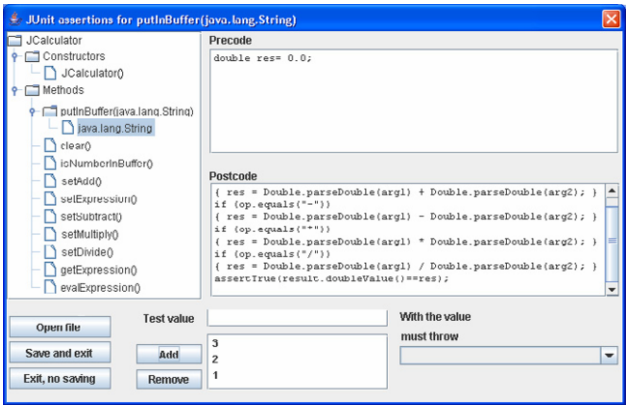
One of the initial steps for building the TS, according to Figure 4, implies some settings like the *protocol of use* (i.e. a RegEx). For `JCalculator` this can be as follows.

```
JCalculator putInBuffer [(setAdd | setSubtract | setMultiply | setDivide)
                         putInBuffer]+ setExpression evalExpression
```

The *testooj* tool makes use of the *java.util.regex.Pattern* class to get a set of *test templates* describing operational sequences. They are generated according to the expected length for expressions (sequences) derived from the RegEx. In this case, the minimum length would be 8 to generate 20 templates to cover the *all–operators* criterion.

The next step involves the setting of *test values*. In order to load *test values* for service parameters, a previous analysis must be carried out on selecting a representative set of test data, in which techniques like Equivalence Partitioning, and Boundary Value Analysis [3,23] (among others), could be very helpful. Figure 5(a) shows how the data (`1,2,3`) were loaded for the only parameter of `putInBuffer` service. They will be used in pairs according to the *protocol of use* (i.e. one value before and after a call to a math service). Figure 5(a) also shows how to edit *constraints* (assertions) in the pre-/post-code areas, that are later inserted before and after the call to a corresponding selected service. Some reserved words are provided to manipulate the called services. The word *obtained* represents the instance of the component under test (CUT). Arguments for parameters are referenced with *argX* – e.g. *arg1* and *arg2* for the two calls to `putInBuffer` (Figure 5(a)). At the right bottom of Figure 5(a) can also be seen how for an *exception* of some service can be set that it must be thrown with a specific test value. However, no exceptions were modelled for `JCalculator` in this case study.

Test cases on JUnit format require to include an oracle, for which operations of the *Assert* class from the JUnit framework help to check the state of the CUT. In the postcode of `evalExpression` service was used *assertTrue* – as shown on Fig-



(a) Constraints, Exceptions and Test Values

```
public String testTS_1_1() {
 try {
   JCalculator obtained=null;
   obtained =new JCalculator();
   java.lang.String arg1=(java.lang.String) "3";
   obtained.putInBuffer(arg1);
   obtained.setAdd();
   java.lang.String arg2=(java.lang.String) "3";
   obtained.putInBuffer(arg2);
   obtained.setExpression();
   java.lang.Double result=obtained.evalExpression();
   return result.toString();  }
 catch (Exception e) {
   return e.toString();  }
}
```

(b) Test Case on MuJava

Fig. 5. (a) Settings in *testooj* – (b) Test Cases for JCalculator

ure 5(a). After this, *test values* are used in combinations with the 20 *test templates* (operational sequences) and *constraints* files (pre/post-code). Four algorithms are provided by *testooj* to produce such combinations: *each choice* [2], *antirandom* [19], *pairwise* [5], and *all combinations* [14]. The last algorithm was applied in this case study.

Each combination becomes a test case, in the form of a testing method inside a test driver file. For `JCalculator`, 468 test cases were generated into a class called `JUnitJCalculator`. After this the TS is validated against `JCalculator`, where *testooj* launches the JUnit tool and iterating through the test cases. They are evaluated according to the included *Assert* operation, thus producing a binary result: either success or failure. The java class `JUnitJCalculator` represents the Component Behaviour TS for `JCalculator`, i.e. the goal to be accomplished on this initial phase. Then a version of the TS on MuJava format can be derived to be used on the third phase of the process. The `MuJavaJCalculator` class was generated with minimal variations: neither pre/post-code is necessary nor oracle (since now the methods return a String). Figure 5(b) shows the test method `testTS_1_1` on MuJava format, which exercises the `setAdd` math service with the test value 3 on both arguments.

In the following section the second phase of the process is explained, which applies when a candidate replacement component must be integrated into the system.

4 Interface Compatibility

This phase takes place when a component K is considered as a potential replacement for a given component C into a system. This particular evaluation is focused on component interfaces, which are compared at a syntactic level. Four levels are defined for services when comparing interfaces syntactically:

(i) *Exact Match.* Two services under comparison must have identical signature. This includes service name, return type, and for both parameters and excep-

tions: amount, type and order.

(ii) *Near-Exact Match.* Similar to level (i), though on parameters and exceptions the order into the list is relaxed. For service names substring equivalence is taken into account.

(iii) *Soft-Match.* Two cases may apply: (1) similar to (ii), though the service name is ignored and for exceptions it is relaxed to mutual existence only; (2) implies subtyping equivalence for return and parameters, and either equality or equivalence for names, and for exceptions like on (i).

(iv) *Near-Soft Match.* Similar to case (1) of (iii), though considering subtyping equivalence for return and parameters at this level.

Data type equivalence concerns the subsumes relationship or subtyping (written $<:$) [31,13], which is implemented for built-in types in this approach according to the *direct* subtyping (written $<_1$) of the Java language [13]. Therefore types on services from K must have at least as much precision as types on C. For instance for an *int* type on C, the corresponding type on K cannot be lower on precision like *short* or *byte* (among numerical types).

The outcome of this step is a matching list characterising each correspondence according to the four levels above. For each service s_C in C, those services from K which are compatible to s_C are added to a list. For example, let be C with three services s_{Ci}, $1 \leq i \leq 3$, and K with five services s_{Kj}, $1 \leq j \leq 5$. After the procedure the returning matching list might look as follows:

$$\{\ (s_{C1}, \{s_{K1}, s_{K2}, s_{K5}\}), (s_{C2}, \{s_{K2}, s_{K4}), (s_{C3}, \{s_{K3}\})\ \}$$

When comparing interfaces, it is enforced that every service of an original component must have a correspondence in the matching list. When a mismatch is found for any original service, the process requires a decision by an integrator. This could be either to provide a manual service matching in order to follow with the process or to stop by concluding the incompatibility of the candidate component. The analysis carried out in this phase initiates from higher matching levels and continues with the weaker ones (i.e. from *exact* to *near-soft*). It is very important to identify strong constrained matches because the outcome of this phase means a pre-analysed knowledge from components under evaluation, which is used as a basis for the next phase to finally get a conclusive result about compatibility.

In an object-oriented framework like Java, there exists a set of methods that are inherited from the *Object* class [13]. In some cases, though not often, those methods may help finding matching when some of them are conveniently overridden. Thus, the option could be to omit those methods in a first try. In case no match is found for a given component service, such *Object* methods could then be considered to observe the results of the matching procedure.

JCalculator-JCalc Interface Matching

Running the Interface Matching between `JCalculator` and `JCalc` reveals that all services from `JCalculator` have found a match – as can be can be seen on Table 1. For example, service `putInBuffer` has a *near-exact–match* with service

Exact	Near-Exact	**Soft**	Near-Soft	(Amount) **Services**
1				(7) getClass, toString, wait, etc.
1	1	20		(2) notify, notifyAll
1		21		(1) clear
	1			(1) isNumberInBuffer
	1	2		(1) putInBuffer
	1	21		(2) setMultiply, setDivide
	6	16		(1) setAdd
		1		(2) evalExpression, getExpression
		22		(2) setExpression, setSubtract

Table 1
Summary of Interface Compatibility for JCalculator-JCalc

`addToBuffer` (due to the substring equivalence) and also two *soft-matches*. Another two `JCalculator` services obtained a unique correspondence by means of a *soft-match*, where service `getExpression` has a match with the `toString` service (from the *Object* class). Moreover, four other services obtained a unique *near-exact–match* and three of them also obtained 21 *soft-matches*. Service `setAdd` obtained 6 *near-exact–matches* and 16 *soft-matches*. Service `clear` obtained an *exact-match* and 21 *soft-matches*. The remainder two obtained 22 *soft-matches*.

The matching list obtained in this phase gives the chance to discover a potential component compatibility by providing information for the next phase which involves the test-based semantic compatibility.

5 Behaviour Compatibility

This phase may not only give a differentiation from syntactic similar services, but mainly assures that interface correspondences also match at the semantic level. Thus the purpose is finding services from a candidate replacement K that expose a similar behaviour with respect to the original component C. In this approach, this implies to exercise the Component Behaviour TS, generated in the first phase of the process, against K.

The automation of this phase is based on the matching information from the Interface Compatibility analysis, which is used to build the wrapper set W for the K component. Each wrapper will be a class which can replace the C component, since it includes the same interface. A wrapper thus behaves as an adapter (i.e. an *adapter pattern* [11]) simply forwarding requests to the K component. The size of W comes from combinations of services matching. Instead of simply making a blind combination, it is possible to get a reduced amount through the previous syntactic evaluation.

The wrapping approach thus makes use of concerns from *interface mutation* [12,6] by applying operators to change service invocations and also to change parameter values. The former is done through the list of matching services. The later, by varying arguments on parameters with the same type. Nevertheless, the amount of correspondences can be reduced by taking the highest compatibility level

obtained in the Interface Compatibility. For instance, for the `clear` service from the case study which has an *exact-match*, could be initially omitted the lower compatibility levels. Thus, this service does not produce additional wrappers.

In summary, the total amount of wrappers is the result from a product of all services from C which have more than one correspondence to K services and those who have parameter correspondences. When the size of W is too high, a system integrator may decide to manually set the correspondences to build only one wrapper, based on the knowledge provided by the Interface Compatibility. For this the *testooj* tool provides ad-hoc utilities. In case no success is obtained with the generated wrapper, another correspondences could be applied, or even decide to change to an automatic building of a bigger set of wrappers.

After building wrappers, the testing step may proceed by taking each wrapper $w \in W$ as the target testing component and executing the Component Behaviour TS. Test case evaluation is done by comparing the results with those saved for component C on the first phase. Thus, each single evaluation gives a binary result: either success or failure. The percentage of successful tests for each wrapper determines its acceptance or refusal, that is either killing the wrapper (as a mutation case) or allowing it to survive. The great the number of killed wrappers the better, because it might facilitate making decisions on compatibility for the component under evaluation.

Running JCalculator's TS on JCalc

In order to initiate the Behaviour Compatibility between `JCalculator` and `JCalc` it is required to build the wrappers set W according to the syntactic matching list generated in the second phase of Interface Compatibility. The highest level of compatibility has been then considered for building the wrappers on this case study. In this case the size of W is $6 * 22 * 22 = 2904$, since only three services from `JCalculator` involved a matching with more than one service from `JCalc`.

After that, the next step is to run the Component Behaviour TS saved on file `MuJavaJCalculator` on each wrapper from W in order to evaluate the semantic compatibility. For this the *testooj* tool provides with an *executor* facility, which is based on the MuJava framework. The executor takes the testing file and iterates through the wrappers list. After this a "*Result Analysis*" utility can show the wrappers that failed the tests when comparing with the original component `JCalculator`. Those failed wrappers correspond to killed mutants, since the application of the *interface mutation* technique. Figure 6 shows a summary of results where only one wrapper passed successfully the tests. This means only one wrapper may survive (as a mutation case) which ease to make decisions whether to accept or discard the candidate replacement component – i.e. `JCalc` in this case study.

In case of the wrapper with 77,77% of success, the only wrong matching involved the service `setSubtract` to service `del` from `JCalc` – instead of `addMinus` which implies the true matching. Although this corresponds to a faulty version of the target wrapper (the one with the 100%), it would give a reasonable decision on compatibility anyway, being quite easy to recognize the wrong service correspondence from

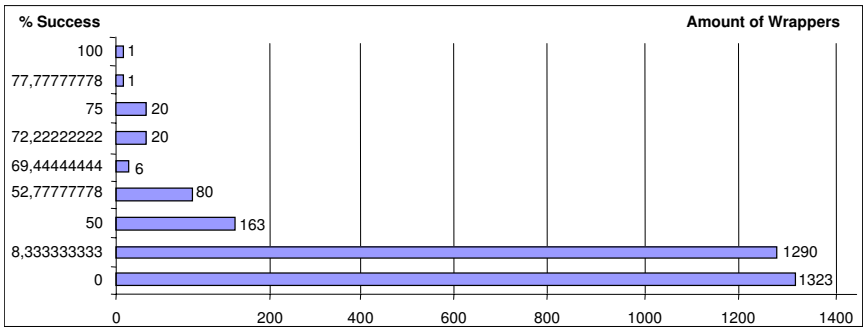


Fig. 6. Results of running JCalculator's TS on JCalc

the failed tests.

The survivor wrapper not only help discovering compatibility between `JCalculator` and `JCalc`, but it also represents the artefact a system integrator requires when tailoring the candidate component (`JCalc` in this case study) to be effectively assembled into the system.

6 Additional Experiment

Another experiment was carried out to observe the effectiveness of the process. This time, the components were download from the SIR repository [3] [7], which is a public repository intended to be used as a benchmark for testing experiments. The java package `JTopas` was selected, which provides a generic, multi-purpose tokenizer for "readable" text (e.g. source code, HTML, XML, ASCII text), to be integrated into a parser. `JTopas` is also available at *http://jtopas.sourceforge.net*. For this experiment the `PluginTokenizer` class has been selected, which represents the main functionality and makes use of the rest of the classes in the package. The whole project of `JTopas` includes 4 versions, from where *version*0 (zero) has been considered as the original component, and the remainder three as candidate replacements.

The first phase of the Process to Evaluate Replaceability was then initiated to build the Component Behaviour TS for *version*0 of `PluginTokenizer`. Together with the project available at the SIR, a test suite on JUnit format is also provided, which was used as a base for learning about the component to develop the corresponding TS. Thus, the initial step for describing the *protocol of use* (to represent operational sequences) was done to achieve an adequate TS for uncovering the required testing coverage criteria – as discussed on Section 3. As a result from this step, three *test templates* were generated.

The downloaded test suite from the project at SIR also provided with a set of *test data*, which consists of 14 HTML files to be "tokenized". These test data were then combined with the three *test templates* to generate a TS comprising 42 test cases (on JUnit format) which were saved on a file called `JUnitPluginTokenizer`. After

[3] The SIR (Software-artifact Infrastructure Repository), http://esquared.unl.edu/sir

that, the TS was run against *version*0 of `PluginTokenizer`, that is the original component, in order to validate the TS. Since results were successful the next step was to derive a version of the TS on MuJava format which was saved on a file called `MuJavaPluginTokenizer`, to be used on the third phase of the process.

The second phase was then initiated, that is the Interface Compatibility, which gave only *exact-matches* as results. In fact, *version*1 and the rest of the versions have a bigger interface than *version*0. This is easily discovered if the order candidate/replacement is inverted, giving some missmatches.

Since only *exact-matches* have been found, then only one wrapper need to be generated. The only additional concern may involve those services whose parameter list has a size bigger than one, where equal (or equivalent) parameters might be located on a different order (into the parameter list) for different versions. However, since those components correspond to successive versions (i.e. upgrades), the initial assumption is that no such changes have been done to those services – in which there is no externally apparent change. This is even more clear, when the major changes that were observed involve the addition of extra services into the interface.

One wrapper was generated for each of the three remaining versions, from where the execution of the TS gave 100% successful results. Therefore no need of generating other wrappers is required to make a decision on compatibility for the set of upgrades.

Whether hypothetically the generated wrappers would give unsuccessful results, the next option would be among the combinations of parameters for those whose type is identical. Since 4 services involve two alike parameters and 3 others involve six alike parameters, the amount of wrappers by considering that option can actually grow to 3456. This means a major saving in effort has been achieved.

7 Related Work

Regression testing is closely related to our goals, which is explained in [25] generally try to apply reduction strategies on a TS in order to improve efficiency without losing safety – i.e. exposing expected faults on targeted pieces. This is achieved by identifying parts affected by changes on successive versions and recognising "dangerous" testing factors – e.g. paths, transitions, branches, sentences, etc. However, such reduction strategies are based on some knowledge about the changed pieces, that is, source code (white-box) or specifications (black-box). Our approach, on the other hand, assume no existence of other information but the one accessible through the reflection mechanism. Besides, candidate replacements are not assumed to be actual new versions of an original component. Therefore, no identification could be done of changed pieces, which thus expose the usefulness of our approach, which is trying to distinguish behaviour compatibility between an original component and an a priori unknown candidate replacement component.

The goals of the work in [20] are very similar to ours. The approach takes a previously generated TS to be executed against the system, from where a monitoring mechanism synthesizes models of interaction and data exchanged. From the mod-

els a reduced TS is extracted focused on a given component under substitution for efficiency purposes. Our approach gives major importance to the TS adequacy by a thorough selection of testing coverage criteria. With this in mind, our process may also accept a previously developed TS, even those designed from specific models (by applying minimal adjustments). Moreover, our approach includes an automatic procedure to work with non syntactic equivalent components, by discovering interface matching which helps to execute a TS against candidate components.

Other important related work is summarised in [16] where approaches concerning BIT (Built-in Testing), testable architectures, metadata-based, and user's specification-based testing are properly covered. In particular, for the BIT strategy it is required from developers (vendor side) to instrument components with an adequate TS which will later help to automatically check whether the component behaves in an expected way when inserted into a system (client side). For instance the approach in [8] is based on a Resolve formal specification, which is used to build assertions instrumented on components which will be verified upon the TS execution. The main difference with all those approaches concerns the underlying purpose of our proposal, which is not based on strategies to find faults for checking the correctness of a component execution. Our intent is to provide a process for component selection that can identify that a certain component may provide the required behaviour, among a set of candidate components. This is achieved through valid configurations of test cases, i.e. those that do not fail during testing. Even for exceptions the intent is to recognize their presence at specific and controlled circumstances.

8 Conclusions

The approach presented in this paper is focused on the maintenance stage where component-based systems require being updated by replacing certain components with other releases (upgrades) or completely different software units (i.e. from a different vendor). The proposal is a Process to Evaluate Replaceability which makes use of testing coverage criteria to describe components behaviour with the purpose of analysing compatibility on candidate replacement components. Therefore, this proposal integrates two aspects: evaluation of compatibility and testing tasks, which therefore reduces effort for system integrators and additionally provides a support on reliability. The *testooj* tool gives automation support for each phase of the process, which helps reducing time and effort and also reinforces control over conditions of each phase in order to achieve a rigorous approach. Since the *testooj* tool is particularly focused on Java components, the next step concerns the deployment of the corresponding upgrades on a version of such a tool which is based on the .Net framework. In this way, the approach could be additionally validated for a different component framework, thus extending the applicability of the evaluation process and providing for integrators a concrete manner to deal with component selection for replaceability.

Acknowledgement

The authors express their gratitude to the board of FME (*www.fmeurope.org*) for being honoured with a grant to participate at VODCA'08. This work has been also supported by UCLM–Indra Software Labs. (Mixed Center of R&D) and projects: CyTED–CompetiSoft (506AC0287), UNCo–ISUCSoft (04-E072), and JCCM-PRALIN (PAC-08-0121-1374).

References

[1] Alexander, R. and M. Blackburn, *Component Assessment Using Specification-Based Analysis and Testing*, Technical Report SPC-98095-CMC, Software Productivity Consortium, Herndon, US (1999).

[2] Ammann, P. and A. Offutt, *Using Formal Methods to derive Test Frames in Category-Partition Testing*, in: 9^{th} *IEEE COMPASS*, Gaithersburg, MD, USA, 1994, pp. 69–80.

[3] Binder, R., "Testing Object Oriented Systems - Models, Patterns and Tools," Addison-Wesley, 2000.

[4] Cechich, A. and M. Piattini, *Early detection of COTS component functional suitability*, Information and Software Technology **49** (2007), pp. 108–121.

[5] Czerwonka, J., *Pairwise Testing in Real World*, in: 24^{th} *PNSQC*, 2006, pp. 419–430.

[6] Delamaro, M., J. Maldonado and A. Mathur, *Interface Mutation: An Approach for Integration Testing*, IEEE Transactions on Software Engineering **27** (2001), pp. 228–247.

[7] Do, H., S. Elbaum and G. Rothermel, *Supporting Controlled Experimentation with Testing Techniques: An infrastructure and its Potential Impact*, Empirical Software Engineering **10** (2005), pp. 405–435.

[8] Edwards, S. H., *A Framework for Practical Automated Black-box Testing of Component-based Software*, Software Testing, Verification and Reliability **11** (2001), pp. 97–111.

[9] Flores, A. and M. Polo, *Testing based Component Assessment for Substitutability*, in: 10^{th} *ICEIS* (2008).

[10] Freedman, R. S., *Testability of Software Components*, IEEE Transactions on Software Engineering **17** (1991), pp. 553–564.

[11] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[12] Gosh, S. and A. P. Mathur, *Interface Mutation*, Software Testing, Verification and Reliability **11** (2001), pp. 227–247.

[13] Gosling, J., B. Joy, G. Steele and G. Bracha, "Java^{TM} Language Specification," Sun Microsystems, Inc, Addison-Wesley, US, 2005, 3rd. edition. URL http://java.sun.com/docs/books/jls/third_edition/html%/j3TOC.html

[14] Grindal, M. et al., *Combination Testing Strategies: a survey*, Software Testing, Verification and Reliability **15** (2005), pp. 167–199.

[15] Heineman, G. and W. Council, "Component-Based Software Engineering - Putting the Pieces Together," Addison-Wesley, 2001.

[16] Jaffar-Ur Rehman, M. et al., *Testing Software Components for Integration: a Survey of Issues and Techniques*, Software Testing, Verification and Reliability **17** (2007), pp. 95–133.

[17] JUnit Home Page, *JUnit.org Resources for Test Driven Development* (2008). URL http://www.junit.org/home

[18] Kirani, S. H. and W.-T. Tsai, *Method Sequence Specification and Verification of Classes*, Journal of Object-Oriented Programming **7** (1994), pp. 28–38.

[19] Malaiya, Y., *Antirandom Testing: Getting the most out of Black-box Testing*, in: *IEEE ISSRE*, Toulouse, France, 1995, pp. 86–95.

[20] Mariani, L., S. Papagiannakis and Pezzè, *Compatibility and Regression Testing of COTS-component-based software*, in: *IEEE ICSE*, 2007, pp. 85–95.

[21] Mariani, L., M. Pezze and D. Willmor, *Generation of Integration Tests for Self-Testing Components*, in: *Workshop ITM-FORTE*, LNCS 3236 (2004), pp. 337–350.

[22] μJava Home Page, *Mutation system for Java programs* (2008). URL http://www.cs.gmu.edu/~offutt/mujava/

[23] Myers, G. J., "The Art of Software Testing," John Wiley and Sons Inc., 2004, 2nd edition.

[24] OMG, *Unified Modeling Language: Superstructure version 2.0.*, Technical report, Object Management Group, Inc. (2005). URL http://www.omg.org

[25] Orso, A. et al., *Using Component Metadata to Regression Test Component-based Software*, Software Testing, Verification and Reliability **17** (2006), pp. 61–94.

[26] Polo, M., S. Tendero and M. Piattini, *Integrating Techniques and Tools for Testing Automation*, Software Testing, Verification and Reliability **16** (2006), pp. 1–37.

[27] Warboys, B. et al., *An Active-Architecture Approach to COTS Integration*, IEEE Software (2005), pp. 20–27.

[28] Wu, Y., M. H. Chen and J. Offutt, *UML-based Integration Testing for Component-based Software*, in: 2^{nd} *ICCBSS'03* (2003), pp. 251–260.

[29] Wu, Y., D. Pan and M. H. Chen, *Techniques of Maintaining Evolving Component-based Software*, in: 16^{th} *IEEE ICSM*, San Jose, US, 2000, p. 236.

[30] Wu, Y., D. Pan and M. H. Chen, *Techniques for Testing Component-based Software*, in: 7^{th} *IEEE ICECCS*, Skovde, Sweden, 2001, pp. 222–232.

[31] Zaremski, A. M. and J. Wing, *Specification Matching of Software Components*, ACM Transactions on Software Engineering and Methodology **6** (1997).