TESTING-BASED COMPONENT ASSESSMENT FOR SUBSTITUTABILITY

Andres Flores[1] and Macario Polo Usaola[2]
[1]*GIISCo Group, Departamento de Ciencias de la Computación*
Universidad Nacional del Comahue, Neuquen, Argentina

[2]*Alarcos Group, Escuela Superior de Informática Universidad de Castilla-La Mancha, Ciudad Real, Spain*

Keywords: Component-based Software Engineering, Substitutability, Upgrade, Black-Box Testing.

Abstract: Updating systems assembled from components demand a careful treatment due to stability risks. Replacement components must be properly evaluated to identify the required similar behaviour. Our proposal complements the regular compatibility analysis with black-box testing criteria to reinforce reliability. The aim is to analyze functions of data transformation encapsulated on components, i.e. their behaviour. This complies with the *observability* testing metric. A Component Behaviour Test Suite is built concerning the integration level to be later applied on candidate upgrades. The approach is supported through a tool developed on our group, *testooj*, which is focused on Java components.

1 INTRODUCTION

Maintenance of systems assembled from components (i.e. component systems) involves updates by replacing existing pieces with upgrades or totally new components. This entails a highly risky situation, where functioning systems stability can be seriously undermined (Heineman and Council, 2001; Warboys et al., 2005; Cechich et al., 2003; Jaffar-Ur Rehman et al., 2007). Any upgraded component must be carefully managed, even those acquired from the same provider. Missing behaviour is usually the main concern, but unexpected functions may certainly bring side effects to the system.

Our main concern is to maintain the integrity of a component system, with the assumption of the usual unavailability of component internal aspects (e.g. source code), and being interfaces the only accessible information for the required compatibility analysis. Our approach is focused on Java components, which includes introspection facilities. This is used to syntactically compare interfaces from a component and its upgrade in order to recognize service correspondences and mismatch cases (Flores and Polo, 2007). Such results become input for a subsequent phase, which complements the regular compatibility approach by means of black box testing criteria. This enforcement is based on the *observability* testing metric (Freedman, 1991; Jaffar-Ur Rehman et al., 2007) which observes the component operational behavior according to its output, as a function of its input. Analyzing the expected input and output data, and how data is transformed into another, provides a reliable way to compare behaviour from components – i.e. to achieve semantic analysis.

Specific testing coverage criteria have been selected in order to design an adequate Test Suite TS as a representation of behavior for components, i.e. a Component Behaviour Test Suite. Such TS is previously developed for each component of an enclosing system, to be later exercised on candidate upgrades to observe behavior equivalence. Generation of Test Cases is achieved through a tool support, *testooj* (Polo et al., 2007), which includes an effective and simplified Meta-Model based on the OMG's UML Testing Profile (OMG, 2005). The tool additionally integrates well-known testing frameworks like JUnit and MuJava. Few extensions to the tool have allowed to be able to apply the Component Behaviour Test Suite on upgrades for compatibility.

The reminder of the paper is organised as follows. Section 2 presents an overview of the whole approach. Section 3 describes aspects concerning the Component Behaviour TS. Section 4 presents the Syntactic Evaluation phase. Section 5 describes the Test-based Semantic Evaluation. Section 6 presents some related work. Conclusions and future work are presented afterwards.

Flores A. and Polo Usaola M. (2008).
TESTING-BASED COMPONENT ASSESSMENT FOR SUBSTITUTABILITY.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - ISAS*, pages 386-393
DOI: 10.5220/0001714103860393
Copyright © SciTePress

2 TEST-BASED COMPATIBILITY ASSESSMENT

The *observability* testing metric (Freedman, 1991; Jaffar-Ur Rehman et al., 2007) is focused on analyzing data transformations from input to output, which helps to understand the functional mapping performed by a component and therefore its behaviour. This may be used to expose a potential compatibility between components – as discussed in (Alexander and Blackburn, 1999; Cechich and Piattini, 2007).

Whilst exploring functional mappings could be extensive, focusing on specific aspects and representative data might effectively accomplish the goal, which could be conveniently addressed through a specific selection of testing criteria.

Based on the previous discussions our proposal implies three main phases to undertake component assessment for substitutability. The approach is depicted in Figure 1. Being *C* an original component and *K* a possible substitute component, the whole process involves the following:

1st Phase. A test suite TS is generated as a behavioural representation of a component *C*. This TS complies with specific criteria which help describing possible interactions of *C* with other components inside a software system. For each component into a system its corresponding TS is built with the only goal to represent behaviour, not finding faults. This will be fully explained in Section 3.

2nd Phase. Interfaces offered by *C* and a replacement *K* are compared syntactically. For this the set of services from *K* must contain the services offered by *C*. At this stage, At this stage, there can be compatibility even when services from *C* and *K* have different names, and parameter order. The outcome of this phase is a Map list where each service from *C* may have a correspondence with one or more services from *K*. Details of this phase are given in Section 4.

3rd Phase. Component *K* which has passed the interface compatibility must be evaluated at a semantic level. The TS built for *C* in the first phase, is now executed against *K* in order to find the true service correspondences from the Map list generated in the second phase.For this, the list is processed to build a set of wrappers (W) for *K*. The ultimate goal is to find a wrapper w_i to replace *C* and allow current *C*'s clients to interact with *K*'s interface. To achieve this, each wrapper $w_i \in W$ becomes the target class under test by running the TS from *C*. After the whole set W has been tested, results are analysed to reveal if can be concluded that a compatibility has been found. This also implies that at least one wrapper $w_i \in W$ can be suitable to allow tailoring *K* to be integrated into the system as a replacement for *C*.

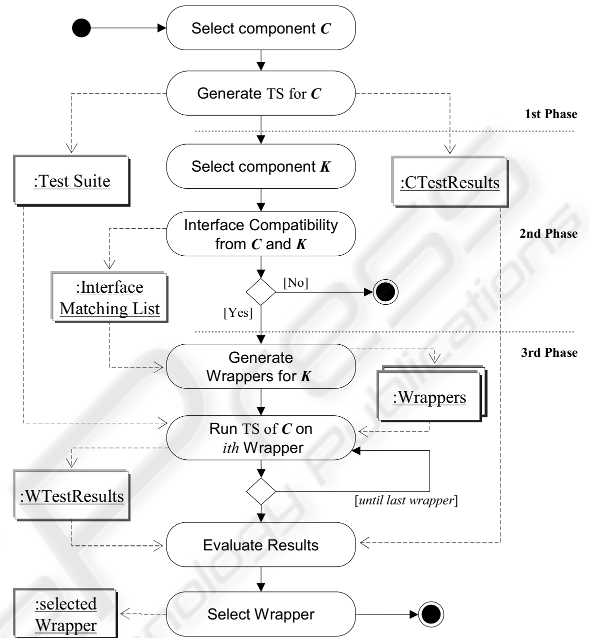


Figure 1: Test-based Compatibility Approach.

Next sections provide detailed information on each step. The application of the process is illustrated by means of the following case study.

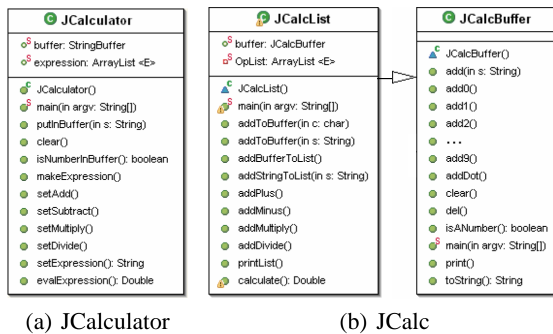
2.1 Case Study

The case study uses a Java calculator, `JCalc`, which have been downloaded from *http://sourceforge.net*, whose main classes are shown in Figure 2(b). `JCalc` has been mutated to create a component called `JCalculator`, whose interface is shown in Figure 2(a). Let us consider `JCalculator` as the original component, therefore `JCalc` becomes its candidate upgrade.

A Component Behaviour TS has been created for `JCalculator` which will be used for the semantic evaluation. Following is explained how this step proceeds.

3 COMPONENT BEHAVIOR TS

In order to build a Test Suite TS as a behavioural representation of components, specific coverage criteria



(a) JCalculator (b) JCalc

Figure 2: Original component (a), and replacement (b).

for component testing has been selected. 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 consists of a set of calls to C's services, together with an Oracle which determines its acceptance or refusal.

An overview of some relevant component coverage notions is given as follows (Wu et al., 2001; Jaffar-Ur Rehman et al., 2007). Beginning from a lower coverage the *all–interfaces* criterion focuses on individual services from each component interface, which is called *all-methods* in (Gosh and Mathur, 2001). Then *all–events* also covers other not publicly accessed events, thus covering *all-exceptions* described in (Gosh and Mathur, 2001; Wu et al., 2000). Since different sequences of events may cause distinct behaviours, each sequence should be tested covering *all–context-dependence*. Then similar to a data flow strategy, *all–content-dependence* focuses on interfaces that may change values which affect behaviour of other(s). For this two cases apply: *intra-* or *inter-component* interface dependence, where the latter requires to design tests with a client and a server component. Similarly for events in case of *all–context-dependence* coverage (Wu et al., 2000).

Our Component Behaviour TS concerns *intra-component dependence*, since this easy evaluating components without extra requisites. In particular, we implement *all–context-dependence* where regular expressions help describing events sequences – the alphabet is comprised of components services. Regular expressions describe a general pattern which is referred to as the "*protocol of use*" for a component interface. Since specific coverage criteria has been proposed in (Mariani et al., 2004) for regular expressions, we expose the relation with the previous component coverage criteria, which explains why regular expressions are an adequate implementation strategy on our approach.

Considering the component service invocation, i.e. *all–interfaces* criterion, a proper coverage is addressed by the *all–alphabets* criterion for regular expressions. For example, the test suite {`abc`} satisfies the alphabets coverage for the regular expression `a*b(b|c)`. In addition, the set of operators for regular expressions (e.g. '`|`', '`*`', etc) help describing every case of service sequences. Thus, the so-called *all–operators* criterion is almost equivalent to *all–context-dependence*. However, it is required to force *all–exceptions* to provide coverage for *all–events*, which is explicitly done in our approach.

Events sequences can also be defined with Finite State Machines (Binder, 2000), which indeed can be represented by regular expressions thus exhibiting subsumes relations on criteria from both notations. For example *all–transitions* criterion (called *all–edges* in (Mariani et al., 2004)) subsumes *all–alphabets*. However *all–operators* is a stronger criterion thus subsuming *all–transitions*.

By means of the *reflection mechanism* elements from Java component interfaces are collected to be able to automate Test Case generation, since they comprise the alphabet for regular expressions. Additionally, exceptions collected from services allow to enforce the representation of components behaviour, satisfying the *all–exceptions* criterion. Thus, the regular expression based approach is properly complemented to achieve the *all–context-dependence* criterion.

By means of the presented case study, is following explained how the procedure to build the Component Behaviour TS is carried out.

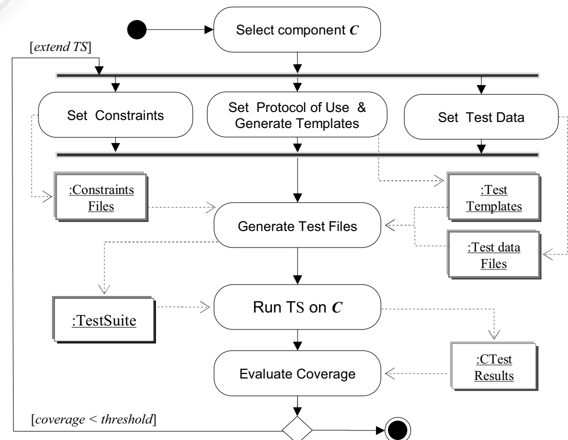


Figure 3: Generation of Component Behaviour Test Suite.

3.1 Test Suite for `JCalculator`

In order to build a Component Behaviour TS for `JCalculator`, we make use of the *testooj* tool, which

involves some steps as can be seen on Figure 3. One of the initial settings implies the protocol of use for `JCalculator`, which could be as follows.

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

Test templates describing service sequences are generated according to the expected length of expressions derived from the protocol of use. The minimum length in this case would be 8, which derives 20 templates involving four expressions with only one math service and sixteen others with an additional iteration for the `'+'` operator (to cover *all–operators*). Next settings imply constraints, exceptions and test values. Figure 4 shows the test values (1,2,3) assigned to `putInBuffer` service's parameter, which will be used in pairs according to the protocol of use – i.e. one value before and after a call to a math service. Constraints which are edited the pre and postcode areas (Fig. 4) are later inserted before and after the call to a corresponding selected service. Some reserved words are provided to manipulate some elements: *obtained* for allocating the instance of the component under test (CUT); *argX* references arguments for parameters – e.g. *arg1* and *arg2* for the two calls to `putInBuffer` (Fig. 4).

Since the correct behaviour of a component may require to throw some exceptions, they are collected from services signatures to set when they should be raised. For this, at the right bottom of Figure 4 is shown how exceptions can be selected for each service by setting that must be thrown upon a specific test value. In this case study however no exceptions were modeled for `JCalculator`.

In order to set the Oracle the *Assert* class provides some operations, which help to check the state of the CUT. For the postcode of `evalExpression` service was used *assertTrue* (Fig. 4). After this, test values can be combined with the 20 test templates (services sequences) and constraints files (pre/postcode). For this *testooj* provides four different algorithms: *each choice* (Ammann and Offutt, 1994), *antirandom* (Malaiya, 1995), *pairwise* (Czerwonka, 2006), and *all combinations* (Grindal et al., 2005). Each combination becomes a test case, in the form of a testing method inside a test driver file which is serialized and saved on a repository. In case of `JCalculator`, 468 test methods have been generated into a class called `TestJCalculator1`, which represents the Component Behaviour TS - i.e. the goal for this initial phase. In the following section is explained the second phase of the process, which applies when a candidate replacement component must be integrated into the system.

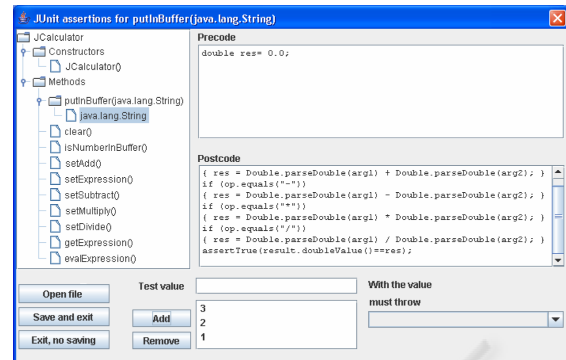


Figure 4: Constraints, Exceptions and Test Values.

4 INTERFACE COMPATIBILITY

This second phase is focused on components interfaces, which are compared at a syntactic level. Its procedure has been updated from a previous model (Flores and Polo, 2007). Concrete aspects related to Java components are now considered, which mainly concern facilities to access interfaces elements provided by the *reflection mechanism*.

Four levels are defined for services when comparing interfaces syntactically. (1) *Exact Match*: two services under comparison must have identical signature. This includes service name, return type, and for both parameters and exceptions: amount, type and order. (2) *Near-Exact Match*: similar to previous, though on parameters and exceptions it is relaxed the order into the list, and also on service names it is observed likely substrings equivalence. (3) *Soft-Match*: two mutually exclusive cases are considered. First one is similar to previous, though service name is ignored and for exceptions it is relaxed to only identify the existence of any. Second one implies subtyping equivalence (Zaremski and Wing, 1997; Gosling et al., 2005) for return and parameters, from where at this level it is required, for service names: equality or substring equivalence, and for exceptions: equality of amount, type and order. (4) *Near-Soft Match*: similar to the first case on the previous level, though considering subtyping equivalence for return and parameters at this level.

The outcome of this step is a matching list characterizing each correspondence according to the levels above. Figure 5 shows algorithms for this step. As can be seen, for each service s_C in C, it is saved a list of compatible services from K. 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 (HashMap) may be as follows:

$\{(s_{C1},\{(\text{n_exact},s_{K1}),(\text{soft},s_{K2}),(\text{n_soft},s_{K5})\}),$

$(s_{C2},\{(\text{exact},s_{K2}),(\text{soft},s_{K4}),(\text{soft},s_{K5})\}),(s_{C3},\{(\text{soft},s_{K3})\})\}$

Since the number of services offered by C and K may differ, every service of C must have a correspondence in the matching list. If a mismatch is found for any original service, the process requires a decision from an integrator. This could be either to provide a manual service matching to continue with the process or simply stop by concluding the incompatibility of the candidate component. Algorithms on Figure 5 try to find matches initiating with strong constraints and then following with the weaker ones (i.e. from *exact* to *near-soft*). It is very important to identify strong constrained matches because it reduces task in the next phase.

In an object-oriented framework like Java, there exists a set of methods that are inherited from the *Object* class (Gosling et al., 2005), which are always present unless inheritance is not considered on the evaluation. In some cases those methods may help finding matching, however they usually do not give interesting aspects for a comparison. Thus, the option is to initially them, and when no match is found for a given component service, such *Object* methods could then be included in the matching procedure.

4.1 `JCalculator-JCalc` Interface Matching

Results from the Interface Matching between `JCalculator` and `JCalc`, reveal for example that service `putInBuffer` has a *soft-match* with two services from `JCalc`, and a *near-exact–match* due to a substring equivalence, as shown in Figure 6. Four other `JCalculator` services obtained a *soft-match* with one service. Moreover, five other services obtained 18 *soft-matches*, two of them are actually a *near-exact–match*, and a third one obtained fifteen of them. Finally, the service `clear` obtained an *exact-match* and 17 *soft-matches*, as can be seen on Figure 7. *Soft-Matches* in this case study were concerned only with ignoring the service name, since no exceptions were found on neither both components. The matching list obtained on this phase gives the chance to discover a potential component compatibility by providing information to build wrappers for the test-based semantic compatibility.

5 BEHAVIOUR COMPATIBILITY

This phase does not only may give a differentiation from syntactic similar services, but mainly assures that syntactic correspondences also match at the semantic level. This means the purpose is finding ser-

```
HashMap buildInterfaceCompatibility(Class C, Class K)
   HashTable result= empty
   foreach method_C in C.getMethods()
      Array compatibles= empty;
      result.put(method_C, compatibles);
      loadCompatibleMethods(method_C, result, C, K)
   endForeach
   return result
end

void loadCompatibleMethods (Method method_C, HashMap result,
                            Class C, Class K)
   foreach method_K in K.getMethods()
      Array compatibles = result.get(method_C)
      if exact_match(method_C, method_K)
          compatibles.add("exact", method_K)
      elseif near_exact_match(method_C, method_K, C, K)
          compatibles.add("n_exact", method_K)
      elseif soft_match(method_C, method_K, C, K)
          compatibles.add("soft", method_K)
      elseif near_soft_match(method_C, method_K, C, K)
          compatibles.add("n_soft", method_K)
      endIf
   endForeach
end
```

Figure 5: Interface Matching Algorithms.

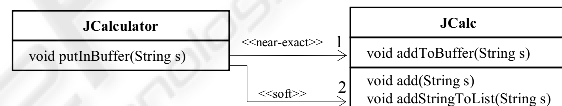


Figure 6: Near-Exact and Soft-Match for `putInBuffer` service.

vices from a candidate K that expose a similar behaviour with respect to an original component C. In our approach this implies to exercise the Component Behaviour TS on the component K.

The automation of this phase is based on the syntactic matching list, which is used to build a set of wrappers W for component K. Each wrapper will be a class which can replace component C, by including the same interface. Wrappers thus act as adapters (i.e. an *adapter pattern* (Gamma et al., 1995)) simply forwarding requests to component K. The amount of wrappers is set according to combinations from the matching of services. Instead of simply making a blind combination, we may get a reduced amount through the previous syntactic evaluation.

The wrapping approach thus makes use of concerns from *interface mutation* (Gosh and Mathur, 2001; Delamaro et al., 2001) by applying operators to change service invocations and also to change parameter values. The former is done through the list of matching services from C to a K component. The later, by varying arguments on parameters while calling to a C service after setting one particular corre-

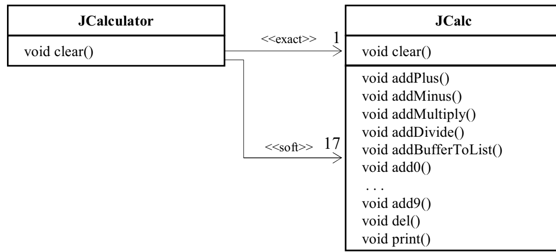


Figure 7: Exact and Soft-Match for `clear` service.

spondence from the matching list. For this, stronger matching cases beginning from *exact–match* are considered to reduce the amount of wrappers. For example the `clear` service from the case study, which resulted with an *exact-match* and the rest being initially omitted. Whether no success is obtained, weaker matching cases could then be considered to expand possibilities.

After building wrappers, the testing step may proceed by taking each $w_i \in W$ as the target testing component and executing the Component Behaviour TS. Test cases evaluation is done with the included *Assert* operation, which thus acts as the test oracle. Hence, test cases produce a binary result: either success or failure. The percentage of successful tests from each wrapper determines its acceptance or refusal, i.e. either killing the wrapper (as a mutation case) or allowing it to survive. The greater the killed wrappers the better, because it might facilitate making decisions on compatibility for the component under evaluation.

5.1 Running `JCalculator`'s TS on `JCalc`

In order to proceed with the semantic compatibility between `JCalculator` and `JCalc` it is required to build the set of wrappers according to the syntactic matching list, which in this case the would be 286 in size. For this case study, however, we have decided to initially follow a more controlled experiment. Thus we have carefully analyzed specific cases of *interface mutation* to select 17 wrappers as follows: 1 wrapper with the true services matching; 4 wrappers varying the `setAdd` service among `addMinus`, `addBufferToList`, `add0` and `add9` – i.e. all with *near-exact–match*; 6 wrappers varying the `setSubtract` service- similar to previous, but changing to`addPlus`, and adding services `del` and `print`; and 6 wrappers varying the `setExpression` service - again similar to previous, replacing `addBufferToList`.

After that the Component Behaviour TS saved on file `TestJCalculator1` was executed against each wrapper to check the semantic compatibility. For this the tool testooj launches the JUnit tool with the testing file and iterating through the wrappers list. Figure 8 shows the results where only one wrapper passed successfully the tests, therefore is the only that may survive (as a mutation case) which facilitates to make decisions whether to accept of discard the candidate replacement component.

Thus the survivor wrapper not only help discovering compatibility but it also represents the artefact an integrator requires when tailoring the candidate component to be effectively assembled into the system.

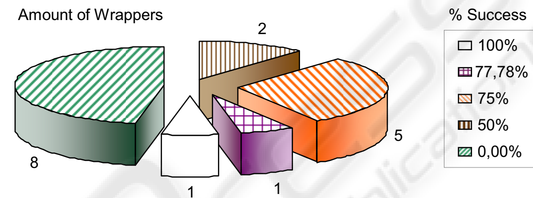


Figure 8: Results of running JCalculator's TS on JCalc.

5.2 Wrappers Set Reduction

The set of wrappers could grow quite high on size according to matching cases identified on the Interface compatibility phase. Though, most of them certainly correspond to faulty versions. This means, many wrappers into the set, in fact do not qualify as interesting artefacts to be considered on an evaluation.

We have selected a second set of 29 wrappers that represent *interface mutation* cases not considered on the initial set. This means, weaker matching cases were applied this time, though at least *soft-match* is considered since data types matching, which otherwise do not pass the static type checking. The second set of wrappers is based on the following: 2 wrappers varying `putInBuffer` service with *soft-matches* shown on Figure 6; 3 wrappers varying the math services and based on the two previous wrappers plus the wrapper with the true matching of the first set; and 4 groups of 6 wrappers varying `setExpression` service among `add0`, `add9`, `del` and `print`; and also taking the 3 previous groups.

Results can be seen on Figure 9, where one wrapper passed successfully the tests and the rest obtained either zero or a very low percentage of success. This means we have another wrapper which could actually survive (as a mutation case), together with the first wrapper of the first set. This second survivor wrapper is included into the first group of this second set, where the `add(String s)` service was taken (Fig. 6) – it is in fact a true matching as well, though actually inherited from `JCalc` superclass.

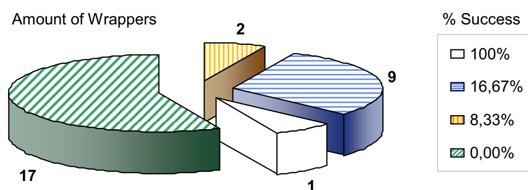


Figure 9: Results of 2nd Group of Wrappers.

This second survivor wrapper could pass unrecognized in a normal process when only high matching cases are considered. Nevertheless, the goal is being able to properly recognize a semantic compatibility, which was perfectly achieved with the first set of wrappers, that was even closer to find survivors on most of their members. On the contrary, the second set of wrappers besides the survivor was too far from finding survivors. This means that the first set was based on a stronger basis, which thus expose the importance of the Interface Compatibility procedure.

6 RELATED WORK

Goals of the work in (Mariani et al., 2007) are very similar to ours. It can dynamically build component data and interaction models from where test suites can be generated. A reduced prioritized test suite for compatibility purposes can be achieved thus becoming an effective approach. Although our approach initially requires to design a specific TS for compatibility purposes, actually any previously developed TS could perfectly be used. Even TS designed from specific models could be applied by providing a Test driver file enclosing the test cases in the form of methods, which does not represent a complex task at all.

Other important related work is summarised in (Jaffar-Ur Rehman et al., 2007) where approaches concerning BIT (Built-in Testing), testable architectures, metadata-based, and user's specification-based testing are properly covered. A main initial difference with those approaches concerns the underlying purpose, which implies to assure a proper component execution, which most them are based on strategies to find faults. However, our approach has a completely different purpose, far from trying to find faults, the process intends to observe a compatibility on behaviour. 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.

Regression testing is closely related to our goals, which as explained in (Orso et al., 2007) 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 recognizing "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, assumes no existence of other information but the one accessible through the reflection mechanism. Additionally, candidate components 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.

7 CONCLUSIONS

Our work is focused on maintenance stage where component-based systems must be updated by integrating upgraded components. We propose an approach based on testing criteria to describe components behaviour with the purpose of analyzing compatibility on candidate upgrades. The approach thus integrates two aspects: compatibility evaluation and testing tasks, which therefore reduces effort for component integrators without missing concerns on reliability. A tool support gives automation to several parts of our approach, *testooj*, which is properly adjusted on every improvement we are performing to our proposal. This helps reducing time and effort and also enforcing control over conditions of each phase in order to achieve a rigorous approach. A next step will be focused on changing the way test results are described. This concerns exceptions related information and the component state after executing testing files, which will be saved on separated XML files. This may improve decision making by quality information from every execution on wrappers for candidate components. Another aspect is related to test selection for the Component Behaviour TS, where prioritisation strategies could help to structure a manageable set of test cases, in order to easy understanding on component behaviour, therefore facilitating explaining levels of compatibility. In order to validate the approach, more experimentation is required which will be achieved through components provided at the SIR repository [1], in order to get visibility on results for

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

external analysis.

ACKNOWLEDGEMENTS

This work is financially supported by UCLM–Indra Software Labs. (Mixed Center of Research and Development) and projects: CyTED–CompetiSoft (506AC0287), UNCo–ISUCSoft (04-E0XX), and UCLM–ESFINGE (TIN2006-15175-C05-05).

REFERENCES

Alexander, R. and Blackburn, M. (1999). Component Assessment Using Specification-Based Analysis and Testing. Technical Report SPC-98095-CMC, Software Productivity Consortium, Herndon, Virginia, USA.

Ammann, P. and Offutt, A. (1994). Using Formal Methods to derive Test Frames in Category-Partition Testing. In *9th IEEE COMPASS*, pages 69–80, Gaithersburg, MD, USA.

Binder, R. (2000). *Testing Object Oriented Systems - Models, Patterns and Tools*. Addison-Wesley.

Cechich, A. and Piattini, M. (2007). Early detection of COTS component functional suitability. *Information and Software Technology*, 49(2):108–121.

Cechich, A., Piattini, M., and Vallecillo, A. (2003). *Component-based Software Quality: Methods and Techniques*, volume 2693 of *LNCS*. Springer-Verlag.

Czerwonka, J. (2006). Pairwise Testing in Real World. In *24th PNSQC*, pages 419–430, Portland, OR, US.

Delamaro, M., Maldonado, J., and Mathur, A. (2001). Interface Mutation: An Approach for Integration Testing. *IEEE Transactions on Software Engineering*, 27(3):228–247.

Flores, A. and Polo, M. (2007). Software Component Substitutability through Black-Box Testing. In *5th Intl Workshop STV'07, during ICSSEA'07*, Paris, France.

Freedman, R. S. (1991). Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6):553–564.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gosh, S. and Mathur, A. P. (2001). Interface Mutation. *Software Testing, Verification and Reliability*, 11:227–247. http://www.interscience.wiley.com.

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

Grindal, M., Offutt, A., and Andler, S. (2005). Combination Testing Strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199. http://www.interscience.wiley.com.

Heineman, G. and Council, W. (2001). *Component-Based Software Engineering - Putting the Pieces Together*. Addison-Wesley.

Jaffar-Ur Rehman, M., Jabeen, F., Bertolino, A., and Polini, A. (2007). Testing Software Components for Integration: a Survey of Issues and Techniques. *Software Testing, Verification and Reliability*, 17(2):95–133. http://www.interscience.wiley.com.

Malaiya, Y. (1995). Antirandom Testing: Getting the most out of Black-box Testing. In *IEEE ISSRE*, pages 86–95, Toulouse, France.

Mariani, L., Papagiannakis, S., and Pezzè (2007). Compatibility and Regression Testing of COTS-component-based software. In *IEEE ICSE*, pages 85–95, Minneapolis, USA.

Mariani, L., Pezze, M., and Willmor, D. (2004). Generation of Integration Tests for Self-Testing Components. In *Workshop ITM-FORTE*, LNCS 3236, pages 337–350, Toledo, Spain. Springer-Verlag.

OMG (2005). UML Testing Profile - Version 1.0. Technical Report formal/05-07-07, Object Management Group, Inc. http://www.omg.org.

Orso, A., Do, H., Rothermel, G., Harrold, M. J., and Rosenblum, D. (2007). Using Component Metadata to Regression Test Component-based Software. *Software Testing, Verification and Reliability*, 17:61–94. http://www.interscience.wiley.com.

Polo, M., Tendero, S., and Piattini, M. (2007). Integrating Techniques and Tools for Testing Automation. *Software Testing, Verification and Reliability*, 16(1):1–37. http://www.interscience.wiley.com.

Warboys, B., Snowdon, B., Greenwood, R., Seet, W., Robertson, I., Morrison, R., Balasubramaniam, D., Kirby, G., and Mickan, K. (2005). An Active-Architecture Approach to COTS Integration. *IEEE Software*, pages 20–27.

Wu, Y., Pan, D., and Chen, M.-H. (2000). Techniques of Maintaining Evolving Component-based Software. In *16th IEEE ICSM*, page 236, San Jose, CA, USA.

Wu, Y., Pan, D., and Chen, M.-H. (2001). Techniques for Testing Component-based Software. In *7th IEEE ICECCS*, pages 222–232, Skovde, Sweden.

Zaremski, A. M. and Wing, J. (1997). Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4).