SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2012; **22**:529–561
Published online 2 August 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.438

Testing-based process for component substitutability

Andres Flores[1,*,†] and Macario Polo[2]

[1] *GIISCo Research Group, Department of Computer Science, University of Comahue, Buenos Aires 1400, 8300 Neuquen, Argentina*
[2] *Alarcos Research Group, School of Informatics, University of Castilla-La Mancha, Paseo de la Universidad 4, E-13071 Ciudad Real, Spain*

SUMMARY

Software components have emerged to ease the assembly of software systems. However, updates of systems by substitution or upgrades of components demand careful management due to stability risks of deployed systems. Replacement components must be properly evaluated to identify if they provide the expected behaviour affected by substitution. To address this problem, this paper proposes a substitutability assessment process in which the regular compatibility analysis is complemented with the use of black-box testing criteria. The purpose is to observe the components' behaviour by analysing their internal functions of data transformation, which fulfils the *observability* testing metric. The approach is conceptually based on the technique Back-to-Back testing. When a component should be replaced, a specific Test Suite TS is built in order to represent its behavioural facets, viz. a Component Behaviour TS. This TS is later exercised on candidate upgrades or replacement components with the purpose of identifying the required compatibility. Automation of the process is supported through the *testooj* tool, which constrains the conditions and steps of the whole process in order to provide a rigorous and reliable approach. Copyright © 2010 John Wiley & Sons, Ltd.

Received 1 May 2008; Revised 5 May 2010; Accepted 17 May 2010

KEY WORDS: component-based software; testing automation; component assessment

1. INTRODUCTION

Component-based development has emerged as an engineering approach to enable faster deployment while reducing costs and effort. The promise of an extensive reusability, however, results usually associated with concerns on the reliability of component services [1, 2]. Similarly, maintenance of component-based systems, which involves replacing or upgrading components, concerns a serious risk upon the stability of functioning systems [3, 4].

Substitutability comprises a major issue on upgrading any software product, due to the evolutive nature of software and the unlikely controllable impact of changes. Particularly, on successive releases of a component, changes might spread across most of the codified functions and structures, affecting stable execution paths. Therefore, a massive difference could be evidenced with respect

*Correspondence to: Andres Flores, GIISCo Research Group, Department of Computer Science, University of Comahue, Buenos Aires 1400, 8300 Neuquen, Argentina.
†E-mail: aflores@uncoma.edu.ar

Contract/grant sponsor: MCIT/PEGASO; contract/grant number: TIN2009-13718-C02-01
Contract/grant sponsor: JCCM-PRALIN; contract/grant number: PAC-08-0121-1374
Contract/grant sponsor: UNComa-IEUCSoft; contract/grant number: 04-E072
Contract/grant sponsor: ANPCYT-MHSMD; contract/grant number: PAE_PICT 2007 2312

to the original component. For instance, releases that incorporate novel technologies or novel architectural solutions are likely to introduce many structural differences. This fact is even truer when components are acquired from different vendors, where system integrators cannot control deployment and are unable to know if the same environment (e.g. compiler and compiling options, among others) was used on components that are presumably alike [5].

Vendors frequently release new versions of popular components, to keep their products competitive in the market. For instance, Crystal Reports (http://www.businessobjects.com), a popular component for creating reports, is updated on a monthly frequency basis. Consequently, a tough responsibility falls upon system integrators to bear in mind keeping the overall system up-to-date and also competitive [5, 6].

Furthermore, system integrators become additionally overloaded with the typical tasks concerning unit and integration testing, which therefore affects the promise of effort reduction in component-based engineering. For this reason, the main concern to be addressed in this paper is to assist integrators to maintain the integrity of component-based systems, while attending its practical side to simplify everyday tasks.

Therefore, the intent of this paper is to provide an Assessment Process as a support for Substitutability, from where one can identify whether new releases or newly acquired components can safely replace the current components from a component-based system that is already deployed, released and in-use. This approach assumes the usual unavailability of internal aspects (such as source code or specifications) from candidate replacement components. The only information that is considered accessible concerns the set of interfaces provided by candidate components. Particularly, the approach is focused on Component Models where the underlying framework should support *introspection* in order to automatically extract interfaces information (e.g. the reflection mechanism on Java and .Net frameworks). Interfaces information is then used on a specific phase whose goal is to identify the level of syntactic interfaces equivalence from a component and its replacement, recognizing at the same time severe cases of mismatch [7].

The syntactic compatibility analysis is complemented by a subsequent phase that makes use of the black-box testing criteria. The purpose is to fulfil the *observability* testing metric [4, 8] to analyse the operational behaviour of a component, which is defined by its expected input and output data, and how data are transformed into another—i.e. its output as a function of its input. Therefore this phase is settled at a semantic level, for which specific testing coverage criteria have been selected to design an adequate Test Suite (TS) as a representation of behaviour for components, namely, a Component Behaviour TS. On a preceding phase, such TS is developed for the component that requires the substitution, and later the TS is exercised against candidate replacement components to observe the level of behaviour compatibility. The entire approach can also be understood through the technique called Back-to-Back testing [9], in which a reference component is used to judge the correctness of a certain unit under test (a candidate replacement component in this approach).

Automation of the approach is currently fully supported for the Java framework by means of the *testooj* tool [10]. While at the beginning of this research, the use of the .Net framework was also checked with successful results [11], Java was finally selected due to other opened working lines of our group. The *testooj* tool allows Test Case generation by the integration of well-known testing frameworks such as JUnit and MuJava [12, 13] both for building the Component Behaviour TS and executing this TS against replacement components to analyse compatibility. The tool additionally helps to achieve a rigorous and more reliable approach through automated steps and conditions.

The remainder of the paper is organized as follows. Section 2 introduces preliminary concepts and related work. Section 3 presents an overview of the Substitutability Assessment Process, and introduces a case study that will be developed along the remaining sections for illustration purposes. Section 4 describes aspects concerning the Component Behaviour TS. Section 5 presents the Syntactic Interface Compatibility phase. Section 6 describes the Test-based Behaviour Semantic Evaluation phase. Section 7 presents an additional experiment that helps to validate the approach. Finally, conclusions and future work are presented.

2. BACKGROUND

A widely accepted definition of the term software component is from Szyperski [14], who defines it as 'a unit of composition with contractually specified interfaces and explicit context dependencies, which can be deployed independently and is subject to composition by third parties'.

Complementary to the previous definition are the following concepts:

Interface: A component encapsulates specific knowledge that is accessible only by its interfaces [6]. Thus, an *interface* acts as an access point for a component, through which services declared in the interface can be requested by a client component.

Events: Interactions with components' interfaces can be identified as consequences of triggering events. An *event* is therefore an incident in which the resulting effect is the invocation of an interface. Particularly, components' interactions occur by external events in which the responding entity is external to the invoking entity. The incident may be triggered by a different interface, through an *exception* or simply through a user action such as pushing a button. In general, an event can be defined as an invocation of an interface through another interface [15].

Component model: The rules for creation, composition and communication of individual components are defined by *component models*. The specification of a software component may also contain the semantics of its interfaces, constraints, data formats and protocols as well as detailed information about timeouts or quality of services. Such a specification may be described by means of predicate calculus or graphical notation such as state charts or Petri nets. The higher the degree of formalization of such descriptions, the better the utilization by verification mechanisms [6].

In current component models, there is no obligation to enrich components with that kind of information. Actually, since component-based development is an inherently complex technology, it might look even more difficult for developers if they were forced to adopt formal specifications. Thus, components are usually provided as a binary code with simple and informal descriptions of integration and deployment aspects, by means of natural language narrative. However, the lack of both source code and complete specifications hinders the applicability and effectiveness of many classic testing and analysis techniques, and challenges system integrators and test designers [5, 6].

Component substitutability: Components are assembled with others in order to build more complex structures or 'composable' software systems. Both the composition mechanism and the set of specific techniques applied to glue together a collection of components impose a strong influence on the dependency between components. Such dependencies have an impact on the degree of *substitutability* of a component into a system [16]. The stronger the dependency of a component to other components within a system, the harder it is to upgrade with new component versions.

Java components: According to the analysis of the state-of-the-art about component evolution by Stuckenholz [6], a Java *package* can be considered as the unit of deployment, being seen therefore as a component. Java maps classes to single files, whereas Java packages may contain structures of directories, together with a physical organization of classes, resources and a manifest. In order for a Java package to suit the definition of a 'software component', there should be a particular class in the package that acts as a *facade* [17] representing the interface of the component. In that case the facade class could also be designed by resembling what in UML is the stereotype 'interface'. For instance, both javabeans and Enterprise Java Beans (EJB), which represent the Java answer for the component technology, are deployed as jar files with a Java 'interface' class acting as the facade or interface of the Java package. Assembly and communication with javabeans and EJB are achieved by means of method calls between classes in the jar files. The standard class loader resolves references along the CLASSPATH and returns the first occurrence of the component [6].

Not all Java packages, however, perfectly fit to the previous features by a clear differentiation of an interface. For example, Java packages based on the Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE) may not be initially designed to work as third-party components. Nevertheless, in some cases those Java packages may contain a class that distinguishes from the others by describing the main functionality and making use of the rest of the classes within the package. Besides, those Java packages are also deployed as jar files, and may communicate to others by method calls as well. This means, although not initially intended to work as components, Java packages based on J2SE and J2EE may also give an opportunity of reuse under the component-based paradigm.

As Java binaries contain meta information, reflection mechanisms can be used to get information from types and exported interfaces of components at runtime. In addition, Java packages could be enriched with coarse-grained version information to improve a versioning system by using the manifest, which usually includes the package name, version number, specification name and specification version [18]. However in Java, component users are responsible for evaluating version information by themselves, since developers are not forced to enrich components with such metadata and there is a lack of a rigorous version policy from which metadata may not be actually useful. The *Package* class contains a method named *isCompatibleWith*, which is supposed to receive a specification version to check the compatibility of the current component to the given version. Even when developers might provide a vague notion of compatibility on such a method, since there are no automatic tools on Java to do these kinds of calculations, developers could make wrong assumptions producing integration failures on a current system [6].

Component testing: As the approach proposed in this paper is based on testing techniques, some definitions regarding component coverage criteria are introduced as follows. Such definitions are particularly based on the concepts previously presented [4].

- *All-Methods*: [19]. Components may have several interfaces, each one composed of a set of methods or services. This criterion requires that every interface must be executed at least once. This means that every service from each interface is invoked at least once. This criterion is also called *all-interfaces* [20].
- *All-Events* [20]. An event is an incident in which the resulting effect is the invocation of an interface. Events could be synchronous (e.g. direct invocations to services) or asynchronous (e.g. triggering exceptions). The criterion requires that every event (synchronous or asynchronous) from a component must be covered by some test. Thus, this criterion covers the specific criterion *all-exceptions* [19, 21].
- *All-Context-dependence* [20]. Events can have sequential dependencies on each other causing distinct behaviours according to the order in which they (i.e. services or exceptions) are invoked. The criterion requires to traverse each operational sequence at least once.

The criteria above have been presented according to the subsumes relationship, from lower to higher coverage. In case of *all-context-dependence,* two cases apply: *intra-* or *inter-component* interface dependence, as follows [21]:

- *Intra-component dependence*. When events present interdependence among each other within a component interface. These events are called *internal events*, and could be invocations to service members of a component interface (or exceptions from those services).
- *Inter-component dependence*. When events from a component interface present a dependence with *external events*. External events are generated by one component and attended by a different component. This requires to design tests with a client and a server component.

All the concepts presented in this section concerning component and coverage criteria will be referenced in the remaining sections. Next, testing notions are particularly considered to give a comparative perspective of the current work related to the proposal of this paper.

2.1. Related work

The proposal of this paper is closely related to Regression Testing, which according to Orso *et al.* [22] is usually involved with applying reduction strategies on a TS in order to improve the 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). The proposal of this paper, 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 exposes the usefulness of this approach, which tries to distinguish the behaviour compatibility between an original component and an *a priori* unknown candidate replacement component.

The approach from Mariani *et al.* [5] has similarities with the proposal of this paper. The approach takes a previously generated TS that is executed against the system. During execution, a monitoring mechanism synthesizes specific models of interaction and data exchanged from participating components. Those models are then used to carry out test selection in order to extract a reduced TS focused on a given component under substitution for efficiency during the compatibility testing process. On the other hand, the process of this paper is based on designing a specific TS for compatibility purposes, by a thorough selection of testing coverage criteria. The reason for this is to be aware of the TS adequacy, which otherwise may give inappropriate testing coverage affecting the reliability of the approach. With this consideration in mind, the process may also utilize any previously developed TS, even those designed from specific models (by applying minimal adjustments). Moreover, the process of this paper includes an automatic procedure to work with non-syntactic equivalent components, by discovering interface matching that helps to execute a TS against candidate components.

Another important related work is summarized by Jaffar-Ur Rehman *et al.* [4] where approaches concerning Built-in Testing (BIT), testable architectures, metadata-based and user's specification-based testing are properly covered. In particular, approaches regarding a BIT strategy require from developers (vendor side) instrument components with an adequate TS that will later help to automatically check whether the component behaves in an expected way when inserted into a system (client side). This is to verify the contract-compliance of the server components (participating in the interactions), including the underlying platform. For instance, the approach by Edwards [9] is based on a RESOLVE formal specification, and the approach by Atkinson *et al.* [23] is based on a KobrA specification. These specifications are used to build assertions instrumented on components that will be verified upon the TS execution.

The main difference with those approaches concerns the underlying purpose of the proposal of this paper, which is not based on strategies to find faults to check the correctness of a component execution. The intent of this paper is to provide a process for component selection, that is to identify that a certain component may supply 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.

3. TESTING-BASED COMPATIBILITY ASSESSMENT

Users take for granted the availability of a daily used functionality, which challenges system integrators to appropriately maintain the expected stability affected by continuous upgrades required by the system. Hence any replacement component (i.e. a new release or a new acquired component) must be carefully managed, even when purchased from the same vendor. A basic need for a system integrator is an effective mechanism to select a certain replacement component from a set of candidate components, and find out whether the selected component can safely replace another currently integrated on a component-based system.

The proposal of this paper, that is, the substitutability assessment process, has been conceptualized under the basis of the technique called Back-to-Back testing, which makes use of a reference implementation for a component to generate a TS and then exercises both a unit under test and the reference implementation. Then, results from the reference component help to judge the correctness of the unit under test (i.e. the candidate component in this proposal) [9]. The proposal considers an original component C (i.e. a reference implementation) and a candidate replacement component K, and the whole process consists of the following three main phases, which are depicted in Figure 1:

First Phase. A TS is generated with the purpose of representing behavioural aspects from an original component C, which requires to be replaced. That is, to generate a description, based on test cases, concerning the likely interactions of component C with other components in a software system. However, the goal of this TS is not to find faults (commonly addressed through failed test cases). The design of the TS is based on a specific selection of the coverage criteria, such as those listed in Section 2, from which components' interactions can be conveniently described. In addition, the selected testing criteria are also used to check the adequacy of the TS. This is fully explained in Section 4.

Second Phase. Interfaces offered by C and a candidate replacement K are compared syntactically. If K offers an interface compatible with that of C, then K is passed to the next phase. The analysis to be performed considers whether the set of offered services from K contains the services offered by C. At this stage, there can be compatibility although services from C and K have different names, different orders in the parameters, etc. The outcome of this phase is a 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 5.

Third Phase. Component K that has passed the interface compatibility analysis must be evaluated at a semantic level. This implies to execute the TS generated for C in the first phase, against K. The purpose is to find the true service correspondences from the list generated in the second phase. For this, such a list is processed in order to generate a set of wrappers (adapters) W for K. The ultimate goal is to find a wrapper $w \in W$ that could replace C to allow current C's clients to interact with the K's interface. To achieve this, each wrapper 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, the results are analysed to conclude if a compatibility has been found. This also implies that at least one wrapper $w \in W$ can be selected as the most suitable to allow tailoring K to be integrated into the system as a replacement for C.

The subsequent sections provide detailed information about each step. The application of the process is illustrated by means of the following case study, which will be used to initially confirm (or definitively refute) the following research hypothesis:

'*The testing-based process for component substitution makes possible to select a replacement component for an original component with a measurable degree of compatibility*'.

Particularly, compatibility can be expressed in terms of the syntactic and semantic distance (being respectively interface and behaviour compatibility). A more formal experiment description is given in Section 7, where the following case study is included, and different experimental aspects are thoroughly analysed and discussed.

3.1. Case study

Let us consider a system developed in Java in which one of the components provides the functionality of a calculator. Such a component has been developed *in-house* and is called `JCalculator`. This component does not present a user front-end (or GUI) but only the main functionality related to mathematical operations, as can be seen in Figure 2(a). As the system needs to evolve, some components may require a redeployment. This is the case with `JCalculator`, for which external components have been considered as potential substitutes.

From the wide spectrum of Java calculators available in the market nowadays, 13 components have been selected to evaluate their compatibility with `JCalculator`. Table I presents those components: their name, the web site from where they have been downloaded and the version (or versions).

In order to clearly illustrate the Substitutability Assessment Process, the case study is initially explained with the first version of component `JCalc` (i.e. version 0.1a). This version includes five Java files: three the front-end (GUI) and two the back-end. Figure 2 shows the back-end

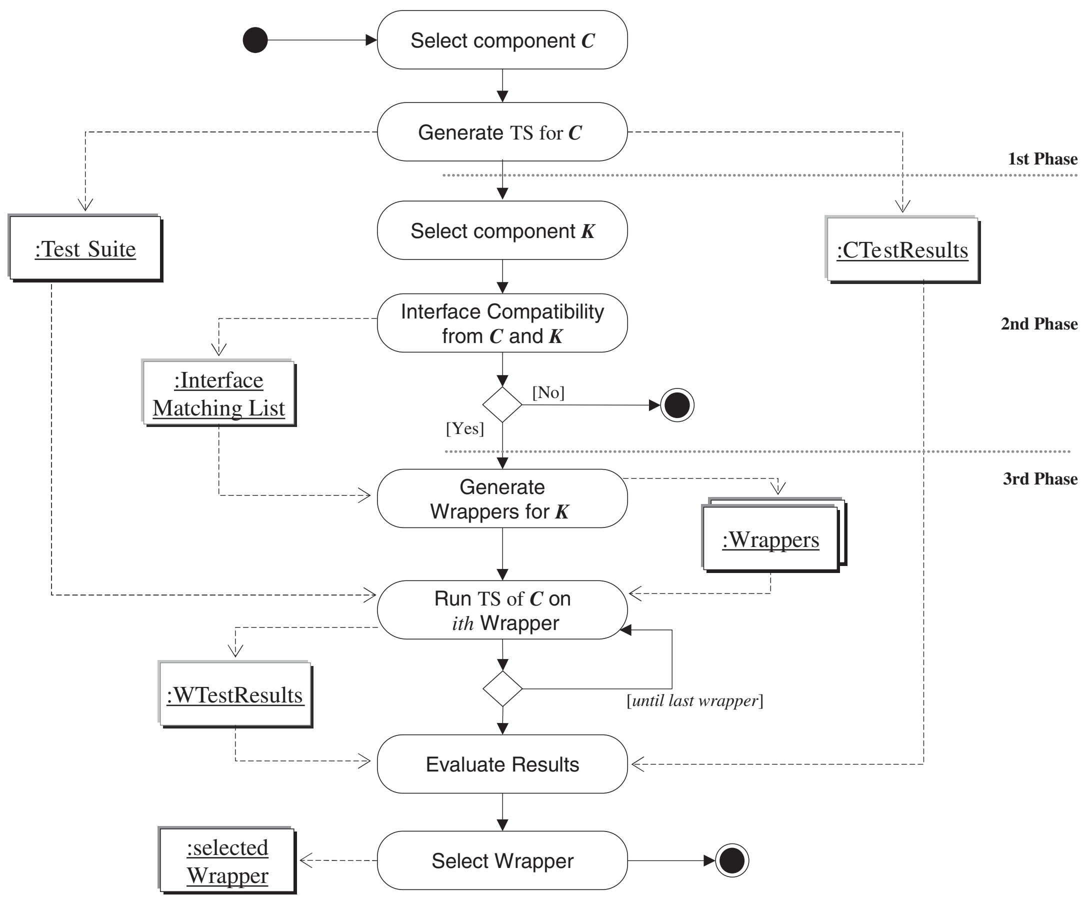


Figure 1. Testing-based substitutability assessment process for software components.

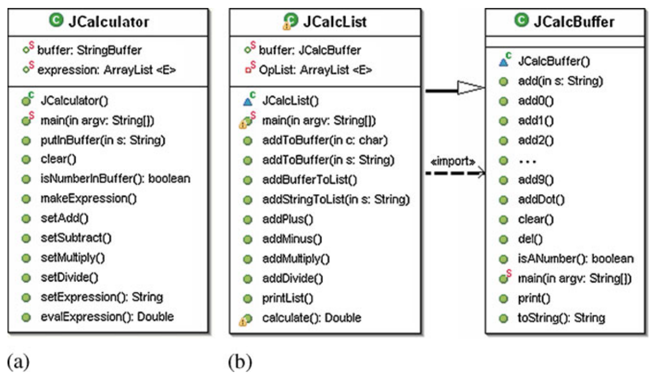


Figure 2. Case study *Java Calculator*—software components: original (JCalculator) (a) and replacement (JCalc0.1a) (b).

classes that represent the main functionality of the package, which therefore could be considered as the component-oriented '*facade*' to be evaluated on this case study. The evaluation of the remaining components is presented in Section 7, where the whole developed experiment is adequately uncovered.

To give a conclusive decision on compatibility between `JCalculator` and `JCalc0.1a`, the first phase of the process must be initiated, which is explained in the following section.

Table I. Java calculators to evaluate compatibility w.r.t. JCalculator.

Component	Web site	Versions
JCalc	*http://sourceforge.net*	0.1a; 0.1b; 0.0.3a; 0.0.4a; 0.1.5
NumberMonkey	*http://sourceforge.net/projects/NumberMonkey/*	3.0b
TerpCalc	*http://www.cs.umd.edu/~atif/TerpOfficeWeb/TerpOffice/TerpCalc/*	4.0
GCalc	*http://gcalc.net, http://sourceforge.net*	3.0
JAC	*http://www.samnhum.net/, http://sourceforge.net*	1.1.232
JSciCalc	*http://jscicalc.sourceforge.net*	2-0.3
OpenCalculator	*http://sourceforge.net/projects/openCalculator/*	0.19
PocketCalc	*http://www.df.lth.se/~mikaelb/, http://pocketCalc.softonic.com/pocketpc/*	1.1
SolCalc	*http://sourceforge.net*	1.1

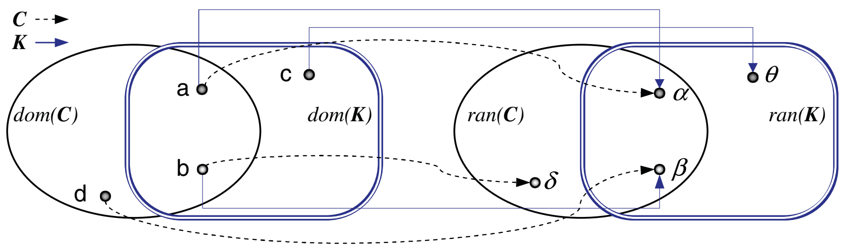


Figure 3. Functional mapping of components C and K.

4. COMPONENT BEHAVIOUR TS

Given a component C and a candidate replacement K, each one accepts certain input data (domain) on their services, from where an internal transformation function returns specific output data (range). It is expected that domain and range from both components should match. However, there is still another concern, which implies the right generation of a particular output from a specific input. This is referred to as the *functional mapping* between domain and range data, and is particularly reflected by the *observability* testing metric [4, 8]. Figure 3 shows a case where domain and range intersect but do not coincide, and the functional mapping does not match as well. For example, for the individual data b from the domains' intersection, the corresponding outputs are not the same—β for K and δ for C, which can be explained as C is '*semantically distant*' from K. A case like this should be properly identified to avoid an undesirable situation in order to maintain the expected system stability.

Therefore, it is very important to analyse the data functional mapping performed by a component to understand its behaviour. When such a behavioural perspective is conveniently exposed from two different components, and then the adequate knowledge is extracted for a comparison procedure, a potential compatibility could be identified—as discussed in [24, 25]. While exploring a complete functional mapping could be extensive, focusing on specific aspects and representative data is more efficient and is also highly effective. To do so, a specific selection of testing coverage criteria provides a useful manner to address functional mapping analysis, which is the option followed in this approach.

After identifying from a component-based system that a given original component C requires a substitution, a TS is built as a representation of the behaviour for that component, for which specific coverage criteria for component testing have been selected. The TS named Component Behaviour TS is composed of test cases that consist of a set of calls to services of component C. The goal of this TS is to be finally exercised against any candidate replacement component K in order to check if K coincides on behaviour with the original component C. Therefore, the results from testing the component C are saved on a repository for determining acceptance or refusal when the TS is applied against component K.

Based on the concepts about components and testing coverage criteria introduced in Section 2, the selected strategy implemented on this approach is explained as follows. The Component Behaviour TS, particularly concerns *intra-component dependence*, from where it is required to describe the interdependence of services (among each other) inside the interface of the original component C. As no external interfaces are involved, no other component is required during testing, which implies that no additional platform or environmental requisites will be involved.

In addition to the *intra-component dependence*, the Component Behaviour TS has been also designed to cover the *all-context-dependence* criterion, which helps to describe the interdependences within the interface of component C by means of sequences of events—either synchronous (like invocations to services) or asynchronous (like triggering exceptions). Such sequences of events (or operational sequences) are formalized in this approach by means of *regular expressions*, where the required alphabet comprises signatures from services of component C. Regular expressions help to describe a general pattern, which can be considered as the '*protocol of use*' for a component interface.

This approach concerns maintenance of a component-based system, which makes components within the system well known. Thus, for a component C to be replaced, the system can be explored to analyse interactions from client components within the system. After this, a system integrator can identify the sequences of invocations to services of the component C, to then compose the regular expression to formalize the *protocol of use*.

Specific coverage criteria for regular expressions have been proposed in [26] concerning their basic elements: the alphabet, the operators and the possibility to derive subexpressions. Table II presents coverage criteria for regular expressions, where the *all-operators* criterion has been extended on this approach according to [27]. The only quantifier or iteration operator considered in [26] was the operator '$*$' (*kleen*). Extending to other quatifiers increases the formalization expressiveness of a *protocol of use*, improving behavioural descriptions of components as well. Criteria for regular expressions expose a relation with the criteria for components presented in Section 2, which is depicted in Figure 4.

Table II. Coverage criteria for regular expressions.

Criteria for regular expressions	Sample: $a^*b(b\|c)$
All-Alphabets: For each symbol in the alphabet, there must be at least a test case that contains the symbol	TS = $\{abc\}$ satisfies *all-alphabet* for the regular expression sample
All-Operators: For each operator: • '|' (*union*) there must be at least a test case that contains the first operand and some other containing the second operand • '+' there must be at least a test case corresponding to exactly one iteration and at least another test case for more than one iteration of the operand • '$*$' (*kleen*) similar to previous and also there must be at least a test case corresponding to zero iteration of the operand • '?' there must be at least a test case corresponding to exactly one iteration and also at least a test case corresponding to zero iteration of the operand	TS = $\{bb, abc, aabb\}$ satisfies *all-operator* for the regular expression sample
All-Expressions: For each choice of operators that results in different sentences of likely different lengths, there must be at least a test case to cover them	TS = $\{bb, bc, abb, abc, aabb, aabc\}$ satisfies *all-expressions* for the regular expression sample

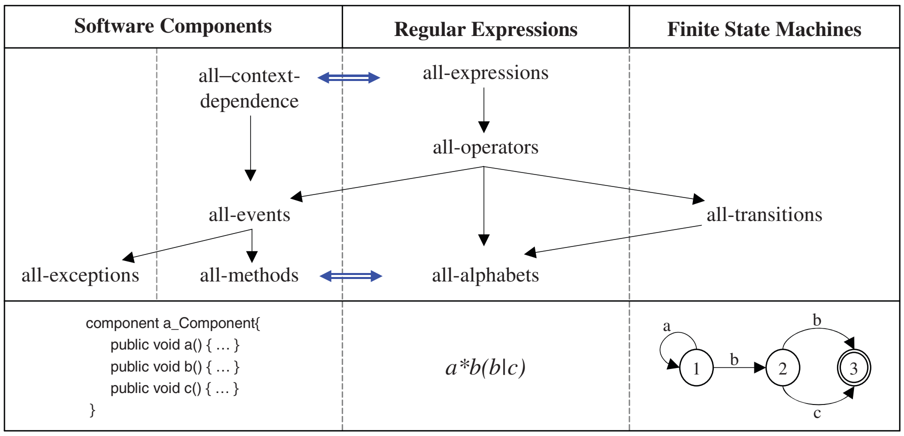


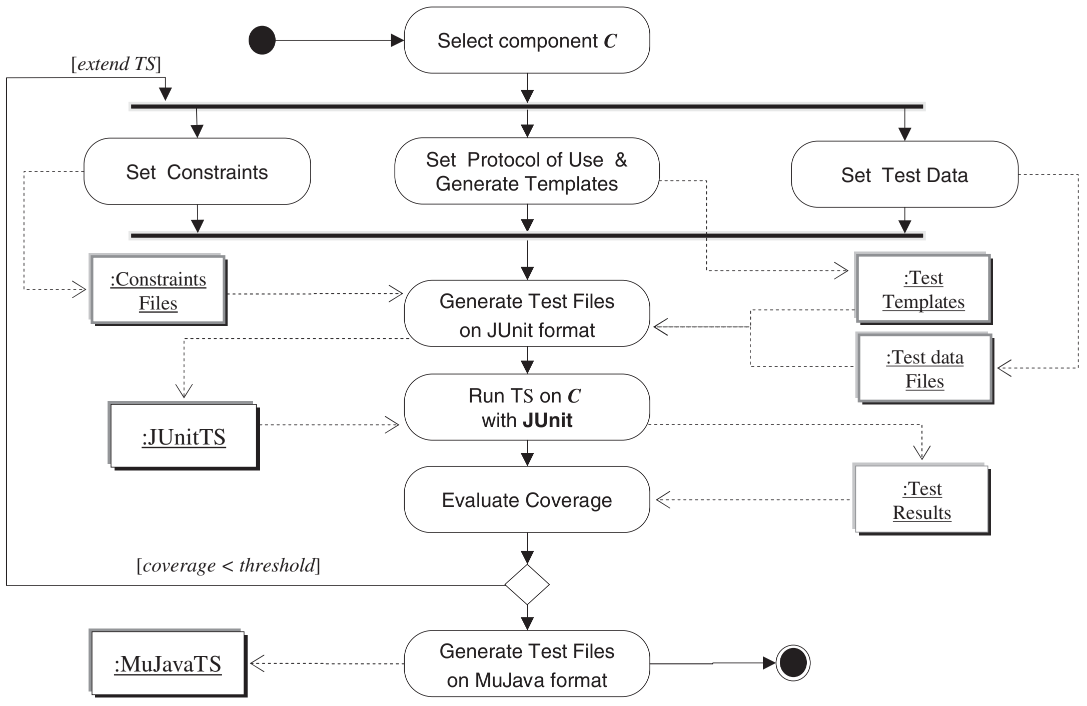
Figure 4. Subsumption and equivalence relation among different criteria.

As component services supply the symbols for the alphabet of regular expressions, the *all-alphabets* criterion allows to invoke all services from a component, therefore covering the *all-methods* criterion. Additionally, the set of operators for regular expressions (e.g. '|', '∗', etc.) helps to describe patterns that include more cases of operational sequences. Thus, the *all-operators* criterion is almost equivalent to *all-context-dependence*. Finally, the *all-expressions* criterion may give a more complete set of operational sequences by describing other meaningful cases that involve additional combinations of operators. For instance, the TS satisfying the *all-expressions* criterion for the regular expression sample in Table II, includes extra test cases not considered for *all-operators*. However, only component services (synchronous events) have been considered on the *protocol of use*, which does not fully cover the *all-events* criterion. That is, exceptions (asynchronous events) must also be included.

In this approach, exceptions are described in a subsequent step by their link to a service invocation and setting what particular data will make the exception to occur. Thus, each declared exception is finally inserted into those operational sequences that include the corresponding service invocation and data. In this way, the *all-exceptions* criterion and therefore the *all-events* criterion are covered. This allows the *all-exceptions* criterion to provide an equivalent coverage to the *all-context-dependence* criterion for components (as can be seen in Figure 4).

The reflection mechanism of the Java framework allows to access elements from a Java component interface to be able to automate Test Case generation. Thus, service signatures can be extracted from an original component C to be used as the alphabet symbols for regular expressions. Additionally, those declared exceptions collected from services (also available via reflection) help in improving the behavioural representation of components. In fact, some exceptions require the component to be on a specific state which may only occur after previous executions of other events (e.g. invocations to certain services); therefore, operational sequences (context-dependence) usually imply the only strategy for a proper coverage.

In a complementary approach, operational sequences could also be derived from Finite State Machines (FSM), which in fact have been widely used in the Testing field for representing behaviour and deriving test cases [28–30]. In this context, the edges of an FSM would represent service signatures, which could be processed to describe different operational sequences—e.g. the example in Figure 4 that is based on the regular expression sample of Table II. Certainly, FSMs can also be used for complex representations of component behaviour with its abstract states and the way they are reached and traversed. As FSMs can be actually represented by regular expressions, equivalence or subsumption relationships can be found on criteria from both notations. For example the FSM's *all-transitions* criterion (called *all-edges* in [26]) subsumes *all-alphabets*. However, *all-operators* has been defined as a wider criterion thus subsuming *all-transitions*.



Select component *C*
[*extend TS*]
Set Constraints
Set Protocol of Use & Generate Templates
Set Test Data
:Constraints Files
Generate Test Files on JUnit format
:Test Templates
:Test data Files
:JUnitTS
Run TS on *C* with **JUnit**
Evaluate Coverage
:Test Results
[*coverage* < *threshold*]
:MuJavaTS
Generate Test Files on MuJava format

Figure 5. Generation of component behaviour test suite.

This section has given the necessary basis to properly understand the main intent of this approach: to provide a process for Component Selection—i.e. to identify that a certain component *K* (from a set of candidate replacements) may supply the required behaviour. This is achieved by observing a compatibility on behaviour through valid configurations of test cases—i.e. those that either do not fail during testing or help in recognizing the presence of exceptions at specific and controlled circumstances.

By means of the case study introduced in the previous section, the procedure to build the Component Behaviour TS is illustrated subsequently, by also explaining how it deals with the analysis concerning coverage criteria.

4.1. TS for JCalculator

In order to build a Component Behaviour TS for `JCalculator`, a set of steps depicted in Figure 5 must be carried out, which are supported by the *testooj* tool [10]. Test cases can be generated on two formats allowed by the tool, which are JUnit [12] and MuJava [13]. Both JUnit and MuJava are testing tools for Java programs—the former being focused on unit testing and the latter on applying mutation strategies.

The TS is initially generated on JUnit format with the intent to run their test cases against the original component—i.e. `JCalculator` in this case study. The purpose is to conveniently validate this TS, which is achieved when 100% of successful results are obtained. The reason is that the TS is actually designed to include configurations of test cases that either do not fail or raise controlled exceptions. In this way, the TS achieves the goal of representing the behavioural facets of the original component. After the TS has been properly validated, then a TS on MuJava format will be derived for being used on the third phase of the compatibility process, as this eases the analysis tasks on that phase—this is fully explained in Section 6.1.

According to Figure 5, one of the initial steps for building the TS implies to specify the *protocol of use* (in the form of a regular expression). By analysing the current use of `JCalculator` in the enclosing system, it was found that an integrator could achieve the following regular expression:

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

From the regular expression above, the *testooj* tool makes use of the *Java.util.regex.Pattern* class to derive sentences in order to create *test templates* describing operational sequences. Such templates are generated according to the expected length for each sentence (amount of alphabet symbols included into a sentence). The minimum length for sentences in this case would be six, which derives four sentences with only one math service, which is initially enough to cover the *all-alphabets* criterion, although, this implies only one iteration for the '+' operator. Some sentences derived from the previous regular expression can be seen as follows, where the first two (columns) are samples with a length of six:

```
JCalculator     JCalculator     JCalculator     JCalculator     JCalculator
putInBuffer     putInBuffer     putInBuffer     putInBuffer     putInBuffer
setAdd          setSubtract     setMultiply     setAdd          setDivide
putInBuffer     putInBuffer     putInBuffer     putInBuffer     putInBuffer
setExpression   setExpression   setAdd          setAdd          setSubtract
evalExpression  evalExpression  putInBuffer     putInBuffer     putInBuffer
                                setExpression   setExpression   setExpression
                                evalExpression  evalExpression  evalExpression
```

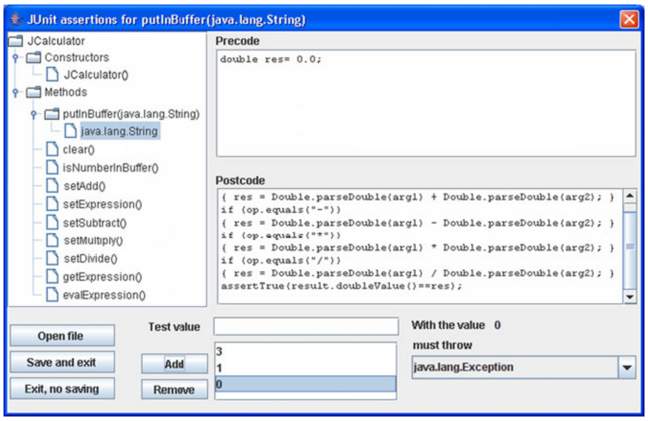
Concerning the *all-operators* criterion, in the previous samples the union '|' operator has been fully covered, and the last three columns show sentences with an additional iteration for the '+' operator. Nevertheless, only one of such sentences with an extra iteration is actually required for the *all-operators* criterion, which added to the four sentences previously mentioned gives a total of five test templates. This means, just adding one of the last three columns (from the sample sentences above) would be more than enough to cover the *all-operators* criterion. Although, the remaining combinations of math services would be excluded, leading to achieve a TS does not adequately describe all meaningful operational sequences.

Therefore, the TS should be generated based on the *all-expressions* criterion, which in this case requires a minimum length of eight. From this, 16 sentences are derived that expose all combinations of math services. Therefore, added to the four sentences previously mentioned gives a total of 20 *test templates* finally generated.

The following steps involve some other settings for constraints, exceptions and 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 such as Equivalence Partitioning and Boundary Value Analysis [28, 31], among others, could be very helpful. Figure 6 shows the *test values* (`0,1,3`) which have been assigned to the only parameter of `putInBuffer` service, 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.

When specific *constraints* or assertions need to be added before and/or after an invocation to a certain service, they can be loaded in the precode and postcode areas—see Figure 6. These constraints are later inserted on test cases before and after the call to the corresponding service. Some reserved words are provided to manipulate elements and to invoke services, such as *argX* (e.g. *arg1* and *arg2* in Figure 6) that is used to reference arguments for parameters, and *result* that is used to hold the value that must be returned by a test case.

The complete description of the correct behaviour for a component may require the triggering of *exceptions*. The list of exceptions is extracted from services signatures to be able to specify when they should be raised. Figure 6 (right-hand side, bottom) illustrates how an exception (from the list of exceptions throwable by a given method) can be selected and is associated with a specific test value previously loaded. Similar to settings for the protocol of use, the test data related to exceptions are selected by a system integrator by identifying in the original component and its



Precode

```
double res= 0.0;
```

Postcode

```
{ res = Double.parseDouble(arg1) + Double.parseDouble(arg2); }
if (op.equals("-"))
{ res = Double.parseDouble(arg1) - Double.parseDouble(arg2); }
if (op.equals("*"))
{ res = Double.parseDouble(arg1) * Double.parseDouble(arg2); }
if (op.equals("/"))
{ res = Double.parseDouble(arg1) / Double.parseDouble(arg2); }
assertTrue(result.doubleValue()==res);
```

Test value

3
1
0

With the value 0
must throw
java.lang.Exception

Figure 6. Constraints, exceptions and test values.

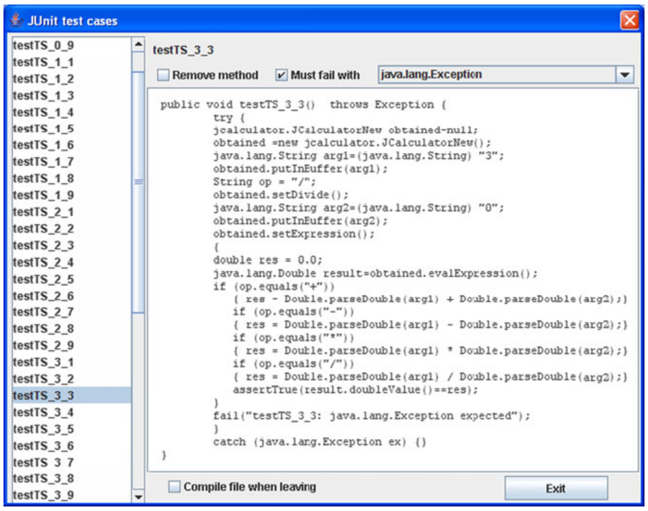
client components, how these kinds of interactions actually occur within the system. In the case of `JCalculator`, there is an exception for the `evalExpression` service upon a division (`setDivide`) by 0 (zero).

As test cases on JUnit format require to include an oracle, the *Assert* class from the JUnit framework provides some operations that help to check the state of a CUT instance. For example, within the postcode area (Figure 6) was used the *assertTrue* operation for the `evalExpression` service. After this, *test values* were used in combinations with the 20 *test templates* (operational sequences) and *constraints* files (pre/post-code). Such combinations are carried out according to one of the four algorithms that are provided by the *testooj* tool, which are particularly useful when more than one *test value* has been loaded. Such algorithms are: *each choice* [32], *antirandom* [33], *pairwise* [34] and *all combinations* [35], for which extended comparative explanations are given in [10, 35]. Particularly, the *all combinations* algorithm was applied for the `JCalculator` TS, which produces the biggest amount of combinations and helps to describe a wider range of interactions from client components (within the system) to `JCalculator`.

From the 20 *test templates*, four of them are combined with test data pairs from a set of three, generating $4*3^2=36$ test cases. The remaining 16 *test templates* were combined with three test data, generating $16*3^3=432$ test cases. Each test case becomes a method inside a test driver file, which is saved on a repository. In the case of `JCalculator`, 468 test cases were generated as methods into a class called `JUnitJCalculator`. Figure 7 lists the generated test cases on JUnit format, and particularly shows one of the test cases, the method `testTS_3_3`, which exercises the `setDivide` math service with test values 3 and 0 on their arguments. This case implies a division by zero, which should raise an exception. Hence, an additional *Assert* operation (*fail*) was inserted into the test case to capture this kind of exception and properly identify the originating cause.

After this, the generated TS can be validated by its execution against the `JCalculator` component. To do so, the *testooj* tool launches the JUnit tool with the `JUnitJCalculator` class and iterates through their test cases. Evaluation of test cases makes use of the included *Assert* operations that act as the test oracle. Hence, test cases produce a binary result: either success or failure. The results produced 72 test cases which raised the expected exceptions (upon a division by zero) and the rest gave successful results, thus validating the TS.

Thus, `JUnitJCalculator` represents the Component Behaviour TS for `JCalculator`, which was the goal to be accomplished in this initial phase of the whole process. Then, the last step of this phase implies to derive a version of the TS on MuJava format, which will be used in the third phase of the process. Thus, the `MuJavaJCalculator` class was generated with minimal variations: neither pre/post-code was necessary nor oracle (since these methods return a String). Figure 8 shows the same method `testTS_3_3` in MuJava format, where the main



testTS_3_3

Remove method [✓ Must fail with] java.lang.Exception

```
public void testTS_3_3()  throws Exception {
        try {
        jcalculator.JCalculatorNew obtained-null;
        obtained =new jcalculator.JCalculatorNew();
        java.lang.String arg1=(java.lang.String) "3";
        obtained.putInBuffer(arg1);
        String op = "/";
        obtained.setDivide();
        java.lang.String arg2=(java.lang.String) "0";
        obtained.putInBuffer(arg2);
        obtained.setExpression();
        {
        double res = 0.0;
        java.lang.Double result=obtained.evalExpression();
        if (op.equals("+"))
           { res - Double.parseDouble(arg1) + Double.parseDouble(arg2);}
           if (op.equals("-"))
           { res = Double.parseDouble(arg1) - Double.parseDouble(arg2);}
           if (op.equals("*"))
           { res = Double.parseDouble(arg1) * Double.parseDouble(arg2);}
           if (op.equals("/"))
           { res = Double.parseDouble(arg1) / Double.parseDouble(arg2);}
           assertTrue(result.doubleValue()==res);
        }
        fail("testTS_3_3: java.lang.Exception expected");
        }
        catch (java.lang.Exception ex) {}
}
```

Compile file when leaving

Figure 7. JUnit test cases from JCalculator's TS.

```
public String testTS_3_3() {
  try {
      JCalculator obtained=null;
      obtained =new JCalculator();
      java.lang.String arg1=(java.lang.String) "3";
      obtained.putInBuffer(arg1);
      String op = "/";
      obtained.setDivide();
      java.lang.String arg2=(java.lang.String) "0";
      obtained.putInBuffer(arg2);
      obtained.setExpression();
      java.lang.Double result=obtained.evalExpression();
      return result.toString();
    }
      catch (Exception e) {
         return e.toString();
    }
}
```

Figure 8. MuJava version of a specific test case for JCalculator.

difference concerns the return type and the fact that all MuJava test cases are prepared to catch exceptions.

4.2. *Additional experiment*: *Validation of black-box testing criteria with white-box criteria*

In order to measure the effectiveness of the criteria selected for the Component Behaviour TS as a strategy to evaluate behaviour compatibility, an additional experiment has been performed, in which the original component (`JCalculator`) was processed by a white-box mutation technique. The availability of a source code for `JCalculator` offered the chance to apply this type of technique. The set of mutant operators applied are classified as class-level and method-level according to [13, 36]. This step was particularly performed by the support of the new version of the MuJava tool which was developed as a plug-in for Eclipse, named MuClipse [37, 38]. This tool is also based on the JUnit framework by accepting a TS on JUnit format to exercise the corresponding mutants.

After running the mutation tool, 144 mutants were generated for `JCalculator`: 3 class-mutants and 141 method-mutants. Then the `JUnitJCalculator` class corresponding to the JUnit format of the Component Behaviour TS for `JCalculator` was run against the mutants. The time required to exercise 468 test cases against the 144 mutants is about 5 h in a regular Pentium 1.83 GHz 2 GB RAM. The results indicate that 33 mutants remained alive and 111 mutants were killed, which implies a mutation score of 77%. However, the alive mutants involved a certain change that actually made them functionally equivalent to the original `JCalculator`. This means that the TS developed for `JCalculator` exhibited the required effectiveness, exposing the adequacy of the criteria selected to design the Component Behaviour TS.

The following section concerns the second phase of the process, which is carried out when a candidate replacement component must be integrated into a system.

5. INTERFACE COMPATIBILITY

When a component is considered as a potential replacement for a given component into a system, the Interface Compatibility phase should be initiated. This particular evaluation is focused on components' interfaces, which are compared from a syntactic point of view.

A services matching schema has been developed for this phase, which comprises four levels. Each matching level implies a different case describing a specific set of syntactic conditions. Table III summarizes the set of conditions for service matching, which are classified according to the element (*return*, *name*, *parameter*, *exception*) of a service signature. Particularly, some subconditions were identified for parameters and exceptions, which are distinguished in Table III. Every condition is explained in detail in Section 5.1. The schema of four levels for syntactic services matching is introduced as follows:

- *Exact-Match*. Two services under comparison must have identical signature. This includes service name (N1), return type (R1), and for both parameters and exceptions: amount, type, and order (P1,E1). Thus, there is an *exact-match* between two services when the following conjunction of conditions is satisfied: $\mathrm{R1} \wedge \mathrm{N1} \wedge \mathrm{P1} \wedge \mathrm{E1}$, which is also condensed by giving a value of 4 (the addition of value 1 from each condition).
- *Near-Exact-Match*. Similar to previous, though the order in the list might be relaxed for parameters and exceptions (P2,E2), for service names there could be a substring equivalence (N2). Then the conditions are roughly grouped as follows: $\mathrm{R1} \wedge (\mathrm{N1} \vee \mathrm{N2}) \wedge (\mathrm{P1} \vee \mathrm{P2}) \wedge (\mathrm{E1} \vee E2)$, but at least P2, N2 or E2 exist, which gives a condensed value between 5 and 7.
- *Soft Match*. Two mutually exclusive cases are considered. The first one is similar to previous; although service name could be ignored (N3), for parameters there can be subtyping equivalence (P3) and exceptions can be relaxed to only identify the existence of any (E3). The second one implies subtyping equivalence for the return (R2) and parameters (P3), where the service name cannot be ignored, and for exceptions it is not relaxed to the existence of any. This gives a value between 6 and 9.
- *Near-Soft Match*. Only the subtyping equivalence for parameters (P3) and the existence of at least one exception (E3) is considered at this level. In case of the return type, there can be either equality (R1) or subtyping equivalence (R2), where for the former the service name cannot be ignored. This gives a value between 8 and 11.

The outcome of this step is a matching list where each service correspondence is characterized according to the four levels previously presented. As the number of services offered by a candidate component may be equal or greater than the original, it has been enforced that every service of an original component must have a correspondence in the matching list. Whether a mismatch is found for any original service, the process requires a decision from an integrator. Thus, a system integrator might either manually define a service matching in order to follow with the process or simply end up the process by concluding the incompatibility of the candidate component.

In an object-oriented framework like Java, there exists a set of methods that are always inherited from the *Object* class [39]. Those methods may help to find matchings when they are conveniently overridden; however, they do not usually provide meaningful aspects to be included into a comparison. Thus, a first option could be omitting those methods, and observe if no mismatch is found

Table III. Summary of syntactic service matching conditions for interface compatibility.

Signature element	Condition	Description
Return type	R0	Not compatible
	R1	Equal return type
	R2	Equivalent return type (subtyping)
Service name	N1	Equal service name
	N2	Equivalent service name (substring)
	N3	Service name ignored
Parameters	P0	Not compatible
	P1	Equal amount ($\texttt{Pc}_1$), type ($\texttt{Pc}_1$) and order ($\texttt{Pc}_3$) for parameters into the list
	P2	Equal amount ($\texttt{Pc}_1$) and type ($\texttt{Pc}_1$) for parameters into the list
	P3	Equal amount ($\texttt{Pc}_1$) and type at least equivalent (subtyping) ($\texttt{Pc}_{2.1}$) for parameters into the list
Exceptions	E0	Not compatible
	E1	Equal amount and type ($\texttt{Ec}_1$), and also order ($\texttt{Ec}_2$) for exceptions into the list
	E2	Equal amount and type ($\texttt{Ec}_1$) for exceptions into the list
	E3	If non-empty original's exception list, then non-empty candidate's list (no matter the type)

for any service of the original component C. Otherwise, such *Object* methods could be included the next time to improve chances of the matching procedure.

The procedure for Interface Compatibility builds the matching list, linking each service s_C from an original component C to a list containing services from a replacement component K, which presents some degree of syntactic compatibility with s_C. For example, let C be a component with three services s_{Ci}, $1 \leq i \leq 3$, and K another component with five services s_{Kj}, $1 \leq j \leq 5$. After the procedure, the matching list may look as follows:

$$\{(s_{C1}, \{(\text{n\_exact}, s_{K1}), (\text{soft}, s_{K2}), (\text{n\_soft}, s_{K5})\}), (s_{C2}, \{(\text{exact}, s_{K2}), (\text{soft}, s_{K4}), (\text{soft}, s_{K5})\}), (s_{C3}, \{(\text{soft}, s_{K3})\})\}$$

The procedure performs the discovery of matches initiating by a strong level to then continue with the weaker ones (i.e. from *exact* to *near-soft*). Identifying strong constrained matches is very important since the outcome of this phase denotes a pre-analysed knowledge from components under evaluation, which is used as a basis for the third phase to finally get a conclusive result of compatibility. In addition, the higher the level of matching found on services, the better it will be for reducing computation on the third phase. This is because on a higher matching level, the amount of correspondences is usually lower—e.g. in case of *exact-match*, only one correspondence can be found. This is explained in-depth in the third phase, by means of the case study.

5.1. *Conditions for syntactic service matching*

Let C be an original component and K its candidate replacement component. For all pair of services $(s_C \in C, s_K \in K)$, a set of individual conditions are described as follows. For brevity reasons, a short explanation is given for conditions of parameters and exceptions.

Return Type (Service Type)

- *Condition R1.* The return types of both services are the same: $\texttt{typeOf}(\texttt{s}_C)\texttt{=typeOf}(\texttt{s}_K)$. `typeOf(x)` is a function returning a type, where `x` can be a service, a parameter or an exception.

Table IV. Built-in direct subtyping.

First Type		Second Type
Byte	$<_1$	Short
Short	$<_1$	Int
Int	$<_1$	Long
Long	$<_1$	Float
Float	$<_1$	Double

Table V. Subtyping equivalence for services.

Type on s_C	Type on s_K
Char	String
Byte	Short, int, long, float, or double
Short	Int, long, float, or double
Int	Long, float, or double
Long	Float or double
Float	Double

- *Condition R2.* The return types of both services are equivalent: `typeOf(`s_C`)`$\approx$`typeOf(`s_K`)`. Data-type equivalence concerns the subsumption relationship of data types, also referred to as subtyping (written <:) [39, 40], which is particularly established for built-in types in this approach, according to the *direct* subtyping (written $<_1$) from the Java language [39], that is shown in Table IV. Thus, for services signatures are considered as subtyping relations as shown in Table V, where types on s_K must have at least as much precision as types on s_C. For instance if service s_C includes an `int` type, then a corresponding service s_K cannot have a lower precision such as `short` or `byte` (among numerical types).

Service Name

- *Condition N1.* The names of both services are the same: `nameOf(`s_C`)=nameOf(`s_K`)`.
- *Condition N2.* The names of both services are equivalent: `nameOf(`s_C`)`$\approx$`nameOf(`s_K`)`. Service name equivalence implies trying to find substring similarity. For instance, on services `putInBuffer` and `addToBuffer` (from the case study that is being developed along the sections), there is similarity on the substring 'Buffer'.

Parameters

- *Condition Pc1.* The number of parameters on both services is the same.
- *Condition Pc2.* The number of parameters on both services is the same and the parameter types are the same in both services, although their order may vary. For example, services s_C`(int,float)` and s_K`(float,int)` fulfill this condition.
- *Condition Pc2.1.* The number of parameters on both services is the same and the parameter types are at least equivalent, in the sense of condition R2. The order of the parameters may vary.
  Equivalence is similar to Condition R2. For example, according to Table V, services s_C`(int,float)` and s_K`(long,double)` fulfill this condition (since `int`$<_1$`long` and `float`$<_1$`double`).
- *Condition Pc3.* Both services have the same order in the parameters list and the type of the ith parameter in s_C coincides with the type of the ith parameter in s_K.

Exceptions

- *Condition Ec1.* The number of exceptions on both services is the same, and every exception type thrown by s_C also appears in the list of exceptions thrown by s_K, though the order of declaration of the exceptions may vary. For example, services s_C`() throws`

`AException,BException` and s_K`() throws BException,AException` fulfill condition Ec1.
- *Condition Ec2.* The number of exceptions on both services is the same and each exception type thrown by s_C also appears, in the same order and with the same type, in the list of exceptions thrown by s_K.
- *Condition E3.* If the exceptions list for service s_C is not empty, then the exceptions list for service s_K cannot be empty.

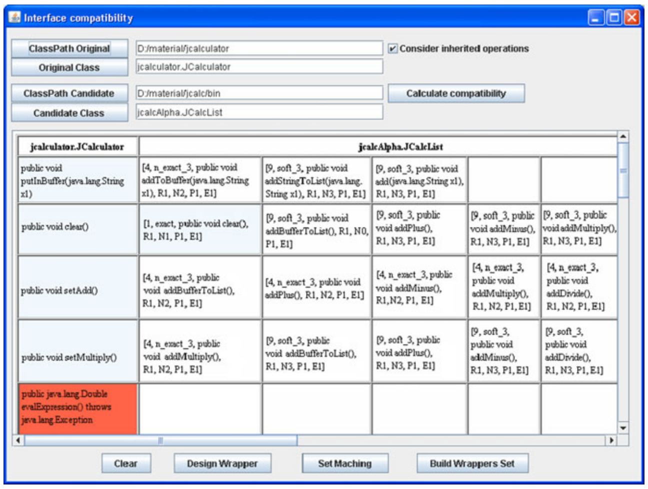
After a detailed view of the conditions for a syntactic interface matching, the Interface Compatibility phase will be illustrated in the following section by means of the case study introduced in Section 3.1.

5.2. *Interface matching between JCalculator and JCalc01a*

When running the Interface Compatibility procedure between `JCalculator` and `JCalc01a`, the *testooj* tool presents the results on a table like that shown in Figure 9. As some of the required services are inherited from a superclass on `JCalc01a` component, the option '*consider inherited operations*' has been set. In case a service from the original component does not match any service from the candidate component, it is displayed with a dark grey cell in the table; otherwise, the cell is light grey as in Figure 9. A summary of such results can be seen in Table VI, which shows that the total value for Interface Compatibility is 92. As a reference, the best value would be 76, when only *exact-matches* are obtained—i.e. 4 * 19 (amount of services on `JCalculator`). Some interesting aspects from this syntactic interface matching are pointed out as follows:

- A match has been found for all services of `JCalculator`, except for the service `evalExpression` which is shown with a dark grey cell in Figure 9. There should be a correspondence with the `calculate` service from `JCalc01a`, condition E3 being the mismatching aspect. That is, an empty exception list on `calculate` service, whereas on `evalExpression` there is an exception to control a division by zero. Therefore, for the process not coming to an end, it was required to manually set the corresponding matching for the third phase of the process. This correspondence has been classified as a special case of *soft-match* in Table VI.
- Those methods inherited from the *Object* class were included in the comparison, and they resulted with only an *exact-match*. The exception was in services `notify` and `notifyAll,` which have found a crossed-relation to their counterparts in `JCalc01a` component (a *near-exact-match*), and also obtained *soft-matches* with 20 other services.
- Among the *Object* class methods, which might be meaningless for a comparison (as discussed in Section 5), the `toString` service from `JCalc01a` found it important to produce a matching with the `getExpression` service, which otherwise would make a service not finding a correspondence, thus leading to the situation explained in the first item.
- High compatibility level is important for the third phase as will be fully explained. This was the case with the first set of services shown in Table VI, among which the `clear` service also obtained an *exact-match*—highlighted in Figure 9 with a light grey cell. This is a propitious case since it also obtained a `soft-match` with 21 services from `JCalc01a`, which otherwise had made more complex the tasks on the third phase.
- Three services obtained a *near-exact-match*, where the substring equivalence allowed to find a higher level than a *soft-match*. Among that set of services, `setDivide` and `setMultiply` had a similar helpful case like `clear` service since they also obtained a *soft-match* with 21 services from `JCalc01a`.

The matching list obtained on 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.



ClassPath Original: D:/material/jcalculator
Original Class: jcalculator.JCalculator
ClassPath Candidate: D:/material/jcalc/bin
Candidate Class: jcalcAlpha.JCalcList
☑ Consider inherited operations

jcalculator.JCalculator	jcalcAlpha.JCalcList				
public void putInBuffer(java.lang.String x1)	[4, n_exact_3, public void addToBuffer(java.lang.String x1), R1, N2, P1, E1]	[9, soft_3, public void addStringToList(java.lang.String x1), R1, N3, P1, E1]	[9, soft_3, public void add(java.lang.String x1), R1, N3, P1, E1]		
public void clear()	[1, exact, public void clear(), R1, N1, P1, E1]	[9, soft_3, public void addBufferToList(), R1, N0, P1, E1]	[9, soft_3, public void addPlus(), R1, N3, P1, E1]	[9, soft_3, public void addMinus(), R1, N3, P1, E1]	[9, soft_3, public void addMultiply(), R1, N3, P1, E1]
public void setAdd()	[4, n_exact_3, public void addBufferToList(), R1, N2, P1, E1]	[4, n_exact_3, public void addPlus(), R1, N2, P1, E1]	[4, n_exact_3, public void addMinus(), R1,N2, P1, E1]	[4, n_exact_3, public void addMultiply(), R1, N2, P1, E1]	[4, n_exact_3, public void addDivide(), R1, N2, P1, E1]
public void setMultiply()	[4, n_exact_3, public void addMultiply(), R1, N2, P1, E1]	[9, soft_3, public void addBufferToList(), R1, N3, P1, E1]	[9, soft_3, public void addPlus(), R1, N3, P1, E1]	[9, soft_3, public void addMinus(), R1, N3, P1, E1]	[9, soft_3, public void addDivide(), R1, N3, P1, E1]
🟥 public java.lang.Double evalExpression() throws java.lang.Exception					

Figure 9. Interface compatibility between JCalculator and JCalc components.

Table VI. Summary of interface compatibility for JCalculator–JCalc01a.

Exact	Near-exact	Soft	Near-soft	(Amount) Services	Best value
1				(7) getClass, toString, wait, etc.	4
1	1	20		(2) notify, notifyAll	4
1		21		(1) clear	4
	1			(1) isNumberInBuffer	5
	1	2		(1) putInBuffer	5
	1	21		(2) setMultiply, setDivide	5
	6	16		(1) setAdd	5
		1		(1) getExpression	6
		22		(2) setExpression, setSubtract	6
		1		(1) evalExpression (*manually set*)	9
	Total	JCalculator services		19 *Total compatibility*	92*

*Best compatibility value = 76.

6. BEHAVIOUR COMPATIBILITY

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

The automation of this phase is based on the syntactic matching information from the Interface Compatibility analysis, which is used to build wrappers for the candidate replacement component. Each wrapper will be a class that can replace the original component, since it includes the same interface and even the same class name. A wrapper thus behaves as an adapter (i.e. an *adapter pattern* [17]), which simply forwards requests to the candidate component. The amount of wrappers is set according to combinations from the matching of services. Instead of simply making a blind combination, it is possible to obtain a reduced amount through the previous syntactic evaluation.

The wrapping approach thus makes use of concerns from *interface mutation* [19, 41] by applying operators to change service invocations and also to change parameter values. The former is done through the list of matching services from the original to a candidate component; the latter, by varying arguments on parameters with the same type, while calling to a candidate component

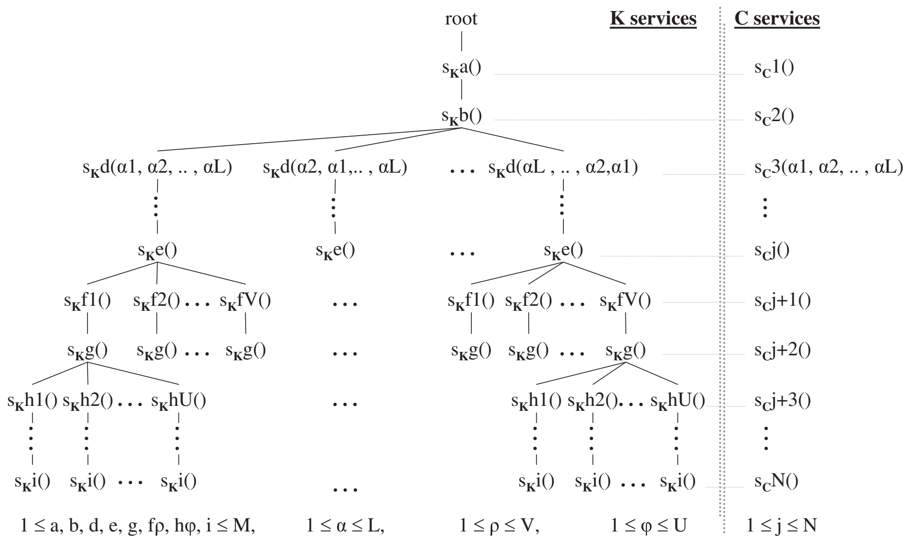


Figure 10. Generation of wrappers by building a tree of services.

service after setting one particular correspondence from the matching list. Figure 10 shows how for each service from the original component C its correspondences from the candidate component K are arranged by means of a tree structure. Each level in the tree implies correspondences for a different service from the C's interface. If a service includes a parameter list of size greater than one, then each combination from types matching produces a new wrapper, which implies another child node in the tree—i.e. a new branch. This can be seen in Figure 10, where the s_{C3} service includes a parameters list of size L.

Each path in the tree (from the root to one leaf) represents a different wrapper to be generated, where the leaves' level gives the total amount of wrappers. This amount proceeds from the correspondences obtained by each service from component C. For example in Figure 10, service s_{Cj+1} (among others) has a correspondence with V services from component K, which produces V branches under each current leaf from the tree.

Nevertheless, the amount of correspondences can be reduced by taking the highest compatibility level obtained in the Interface Compatibility analysis from the second phase of the process. Note, for example, that the `clear` service from the case study, which had an *exact-match*, and other lower compatibility levels (see Table VI): taking only such correspondence and omitting the rest, this service does not produce an additional branch on the tree—e.g. services s_{C1} and s_{C2} (among others) in Figure 10.

In summary, the total amount of wrappers can be calculated by multiplying the number of matchings for those services from C with more than one correspondence to services of K, and also those which have parameter correspondences. The corresponding formula for the set of wrappers W is introduced as follows:

$$size(W) = \prod_{i=1}^{N} \sum_{j=1}^{V_i} \prod_{t=1}^{T_i} p_t! \tag{1}$$

where N is the size of interface for C, M the size of interface for K, V_i the amount of matching from s_{Ci} to K interface, $1 \leq V_i \leq M$, L_i the size of parameter list for s_{Ci}, T_i the amount of different types on parameter list for s_{Ci}, $0 \leq T_i \leq L_i$, p_t the amount of occurrences for a type in the parameter list for s_{Ci}, $0 \leq p_t \leq L_i$ and $p_t!$ the permutations of the occurrences for a parameter type on service s_{Ci}.

Since the size of W could be quite big (considerably affecting the timing for the current phase), a different option could also be considered. The previous phase, which analyses Interface Compatibility, provides an important knowledge from services correspondences, by highlighting

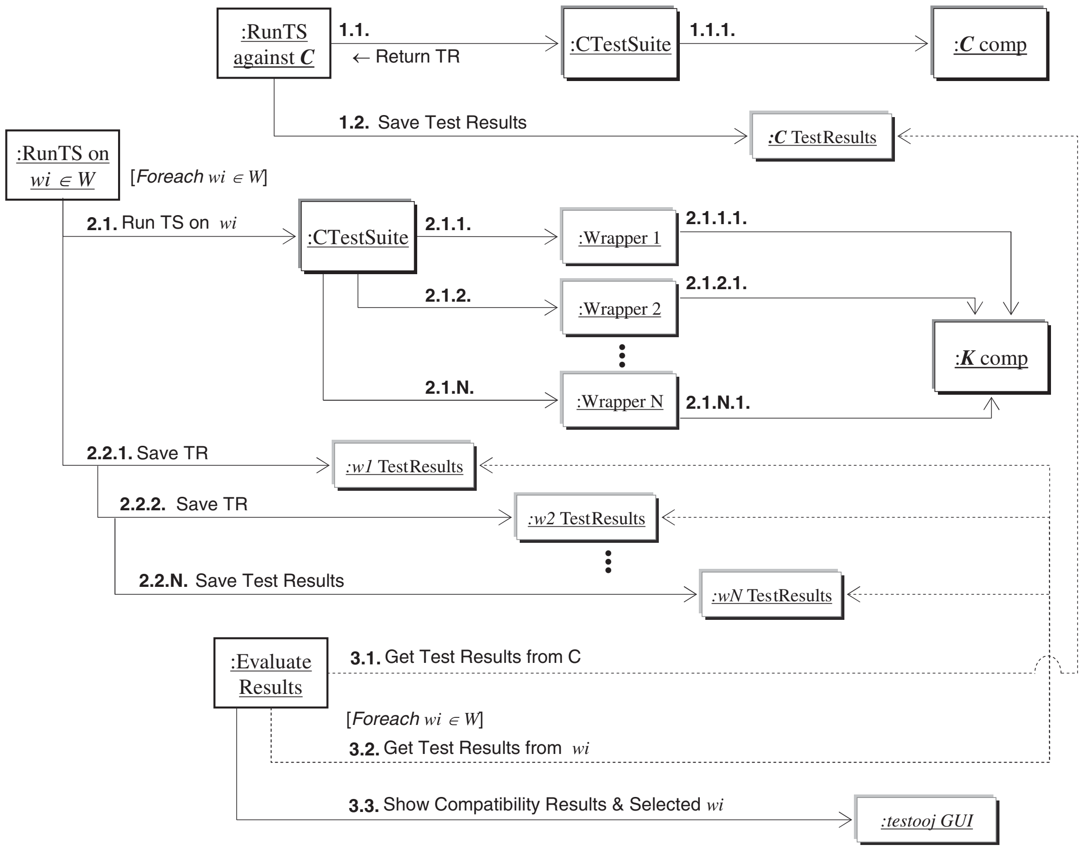


Figure 11. Running TS against both C and K's wrappers, and evaluation of their results.

the likely purpose (or behaviour) of different services. This basically concerns the relation between a service name and its intended purpose as a part of the whole functionality provided by the components under comparison. Thus, for some services it might become clear what the appropriate correspondence would be.

Therefore, an integrator could decide to manually set all correspondences, one by one, to build only one wrapper. And for this the *testooj* tool also provides *ad hoc* facilities to do it. In case no success is obtained with the generated wrapper, another correspondence could be applied, or even a decision to change to an automatic generation of a bigger set of wrappers could be made.

After building the wrappers set (W), the Behaviour Compatibility may proceed by taking each wrapper as the target testing component and executing the Component Behaviour TS. Figure 11 shows three steps of the process: (1) how test results are obtained from the original component C; (2) how wrappers of candidate component K are tested against the TS generated for C and (3) how results from tested wrappers are compared with those from C. Test cases evaluation is done by means of the returned value, which is a serialized String. Thus, the evaluation on each test case gives a binary result: either success or failure. The percentage of successful tests from each wrapper determines its acceptance or refusal, that is either killing the wrapper (as a mutation case) or allowing it to survive. The greater the killed wrappers, the better it is, because it might facilitate making decisions on compatibility for the component under evaluation.

The Behaviour Compatibility phase is illustrated as follows by using the case study that has been developed in the previous sections.

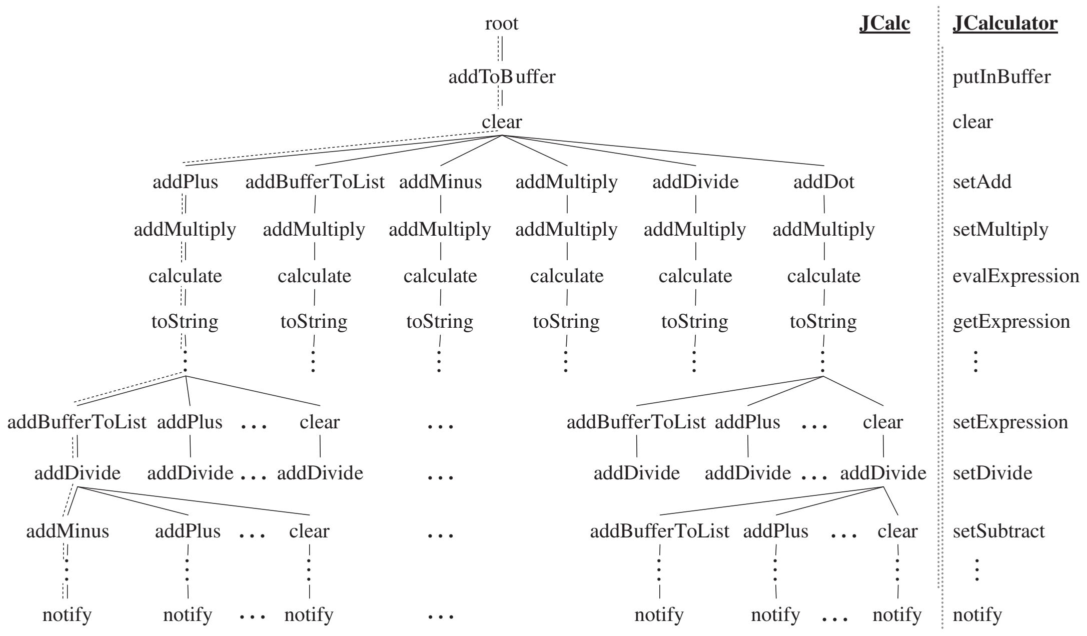
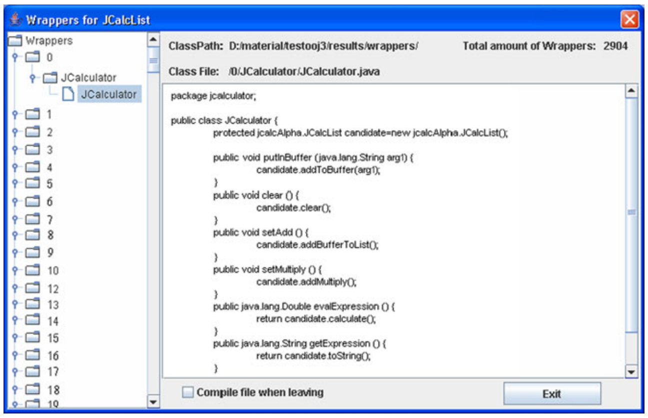


Figure 12. Tree from where wrappers to JCalc01a are generated.



ClassPath: D:/material/testooj3/results/wrappers/ Total amount of Wrappers: 2904

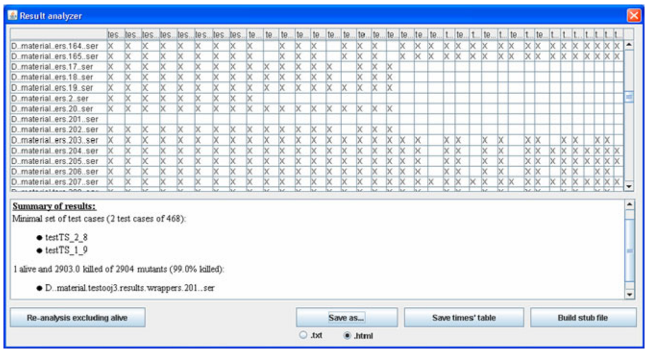
Class File: /0/JCalculator/JCalculator.java

```
package jcalculator;

public class JCalculator {
        protected jcalcAlpha.JCalcList candidate=new jcalcAlpha.JCalcList();

        public void putInBuffer (java.lang.String arg1) {
                candidate.addToBuffer(arg1);
        }
        public void clear () {
                candidate.clear();
        }
        public void setAdd () {
                candidate.addBufferToList();
        }
        public void setMultiply () {
                candidate.addMultiply();
        }
        public java.lang.Double evalExpression () {
                return candidate.calculate();
        }
        public java.lang.String getExpression () {
                return candidate.toString();
        }
```

Figure 13. JCalculator's wrappers to the JCalc component.

6.1. *Testing-based compatibility between JCalculator and JCalc01a*

In order to initiate the Behaviour Compatibility between `JCalculator` and `JCalc01a`, 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 in this case study. Figure 12 shows the tree that is generated before building wrappers, where the size of the leaves level rises to 2904—i.e. the size of W. Only three services from `JCalculator` involved a matching with more than one service from `JCalc01a`: the `setAdd` service with six matches and both `setExpression` and `setSubtract` with 22 matches each—as can be seen in Table VI from Section 5.2. Formula (1) gives the $size(W)$: $6*22*22=2904$. Figure 13 shows how the *testooj* tool presents the list of generated wrappers, where in particular the first wrapper is displayed, which corresponds to the first path (to the left) in the wrappers tree in Figure 12—marked with dashed lines. Generation of the wrapper set W was done in about 1.5 h.



	tes...	tes...	tes...	tes...	tes...	tes...	tes...	tes...	te...	te...	te...	te...	te...	te...	te...	te...	te...	te...	te...	te...	te...	te...	te...	te...	te...	t...	te...	t...	te...	t...	te...	t...	te...	t...	t...	t...	t...	t...	t...	t...
D..material..ers.164..ser	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X		X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
D..material..ers.165..ser	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X		X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
D..material..ers.17..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X																		
D..material..ers.18..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X																		
D..material..ers.19..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X																		
D..material..ers.2..ser	X	X	X	X	X	X	X	X	X																															
D..material..ers.20..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X																		
D..material..ers.201..ser																																								
D..material..ers.202..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X	X																		
D..material..ers.203..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X	X		X	X		X	X		X	X	
D..material..ers.204..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X	X		X	X	X	X	X	X	X	X	X
D..material..ers.205..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X	X		X	X	X	X	X	X	X	X	X
D..material..ers.206..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X	X		X	X		X	X		X	X	
D..material..ers.207..ser	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Summary of results:

Minimal set of test cases (2 test cases of 468):

- testTS_2_8
- testTS_1_9

1 alive and 2903.0 killed of 2904 mutants (99.0% killed):

- D..material.testooj3.results.wrappers.201...ser

Figure 14. Execution of JCalculator'TS to analyse semantic compatibility of JCalc01a.

Table VII. Summary of results from running JCalculator's TS on JCalc01a.

Case	Success (%)	Wrappers
1	100	1
2	77.77	1
3	75	20
4	72.22	20
5	69.44	6
6	52.77	80
7	50	163
9	8.33	1290
10	0	1323
Total		2904

In order to evaluate the semantic compatibility, the next step is to run the Component Behaviour TS saved on the `MuJavaJCalculator` file on each wrapper from W . For this the *testooj* tool provides an *executor* facility, which executes the MuJava test cases, and takes the testing file iterating through the wrappers list. For this case study, the execution was done in about 7 h.

Then by pressing the '*Result Analysis*' button, *testooj* shows a table with the list of wrappers, whose cells are marked with '`X`' if the corresponding test case shows a different result when comparing with the original component `JCalculator` (see Figure 14). Those failed wrappers correspond to killed mutants (in mutation language), since they were generated by the application of the *interface mutation* technique. Table VII shows a summary of results where only one wrapper passed successfully the tests, which then corresponds to an alive mutant. This means that a specific wrapper has been identified whose behaviour is exactly equal to the original component, and has been checked with an adequate TS according to the selected black-box coverage criteria. Therefore, the example reveals that `JCalc01a` is a fair option as a replacement component for `JCalculator`.

Nevertheless, there is another wrapper with a quite high percentage of successful results, i.e. 77.77%, which actually involves only one wrong service matching: from the `setSubtract` service to `del` service of `JCalc01a` (instead of `addMinus` which implies the true matching). Even when this wrapper is clearly a faulty version of the 'alive' wrapper (with 100%), it could represent an alternative good choice by describing a percentage above 70%.

The survivor wrapper not only helps to discover compatibility between `JCalculator` and `JCalc01a`, but it also represents a potential artifact to be used by an integrator to adapt the candidate component (`JCalc01a` in this case study) to be effectively assembled into the system.

Table VIII. Results from running the second group of wrappers for JCalc01a.

Case	Success (%)	Wrappers
1	100	1
2	16.66	90
9	8.33	20
10	0	170
Total		281

6.2. *Size reduction of wrappers set*

The set of wrappers could grow on size pretty high according to matching cases identified on the Interface compatibility phase. Nevertheless, most of them certainly will correspond to faulty versions of a target wrapper—the one that can describe the true service matching. Thus, many wrappers in the set do not qualify as interesting artefacts to be considered on an evaluation.

For example, the amount of wrappers for the developed case study could grow over a thousand million when only *soft-matches* are considered. Instead, the wrapper set has a size of only 2904, which is far lower from that hypothetical initial size. This reduces the testing effort and increases the performance of the process.

Reduction strategies are applied on the Interface compatibility phase, with the intention to increase the amount of higher levels of compatibility—i.e. *exact-matches* cases or at least *near-exact-matches*. Hence, the practical approach that was implemented involves to analyse service names in order to find substring correspondences—as mentioned in Section 5. In the case study for example, there is a *near-exact-match* for services `setMultiply` and `setDivide` with `addMultiply` and `addDivide`, respectively. Even `setAdd` service obtained a less amount of correspondences (6 instead of 22), by using such strategy.

A second group of 281 wrappers has been built concerning *interface mutation* cases not considered on the set of 2904 wrappers. This means, lower compatibility levels were applied this time. For instance, the two *soft-match* correspondences for the `putInBuffer` service were considered this time (instead of *near-exact-match*), which can be seen in Figure 9 (Section 5.2). The results after running the Component Behaviour TS for `JCalculator` can be seen in Table VIII, where one wrapper passed successfully the tests and the rest obtained either zero or a very low percentage of success. This means that there is another wrapper that could actually survive. This second survivor wrapper has the only difference of a matching from `putInBuffer` service to `add(String s)` service (see Figure 9)—which in fact represents a true matching as well, although actually inherited from a superclass in the `JCalc01a` component.

This second survivor wrapper could pass unrecognized in a normal process when only high compatibility levels are considered. Nevertheless, the important 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. This second set of 281 wrappers on the contrary, 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.

7. DESCRIPTION OF THE EXPERIMENTS

In Section 3.1 it was mentioned that the case study concerning the Java calculator `JCalculator` was actually developed with 13 components. Only one component has been shown so far for running a comparison, the `JCalc01a` component. This section exposes the evaluation of the remaining components to observe different situations when dealing with externally developed components. After that, another experiment is introduced, concerning a completely different set of components.

Table IX. Java calculators evaluated on compatibility w.r.t. JCalculator.

Component	Versions	Content (classes)	Source code available	Compatible
JCalc	0.1a	5	Yes	✓
	0.1b	4	Yes	✓
	0.0.3a	6	Yes	X
	0.0.4a	6	Yes	X
	0.1.5	17	Yes	X
NumberMonkey	3.0b	5	Yes	✓
TerpCalc	4.0	32	Yes	✓
GCalc	3.0	>100	Yes	X
JAC	1.1.232	20	Yes	X
JSciCalc	2–0.3	>100	Yes	X
OpenCalculator	0.19	38	No	X
PocketCalc	1.1	3	Yes	X
SolCalc	1.1	3	Yes	X

However, before giving details of the two case studies, the whole experiment is presented in a formal way by describing its associated features. The first aspect implies the research hypothesis that was presented before introducing the case study of a Java calculator in Section 3.1, and was stated as follows:

‘*The testing-based process for component substitution makes possible to select a replacement component for an original component with a measurable degree of compatibility*’.

Particularly, compatibility is expressed in terms of the syntactic and semantic distances (being, respectively, interface and behaviour compatibility), which then represent the dependent variables. Independent variables involve the elements used to obtain a syntactic and semantic distance, as follows:

- Syntactic distance. Independent variables concern the elements from each service signatures, which are analysed to obtain a service matching level. They are defined in Section 5.
- Semantic distance. Independent variables concern each successful or failed test case for each wrapper, from where the percentage of successful results is calculated to obtain at least a wrapper that may be considered adequate to exhibit a semantic compatibility. This was defined in Section 6.1

7.1. *Thirteen java calculators*

Thirteen Java calculators have been downloaded from different web sites and used for evaluation against the `JCalculator` component. The `JCalc` component particularly includes five versions, from where version 0.1a was already observed along the previous sections, since it was used to illustrate the whole assessment process presented on this paper. Table IX shows the amount of classes comprising each component (Java packages) in order to have a perception of their complexity, together with the final result on compatibility (in a reduced form: acceptance or refusal). Each case of comparison is explained in a more detail as follows.

7.1.1. Compatible components. Table X presents a summary of second and third phases of the assessment process applied on the four compatible components, including version `01a` of `JCalc`, in order to contrast the results of each case. Additionally, Table XI shows the results of running the TS for `JCalculator` against the four compatible components.

JCalc01b: This component is very similar to the previous version, although it provides an extra mathematical service called `addNegative`. Although this produces additional correspondences on the Interface Compatibility phase, the resulting value remains the same as in the first version of `JCalc` (see Table X), even when the same situation of a manual matching set for `evalExpression` service was required this time as well. However, the additional correspondences have affected the

Table X. Compatible Java calculators w.r.t. JCalculator.

			Behaviour compatibility	
Component	Facade class	Interface* compatibility	Wrappers	Best wrapper
JCalc01a	JCalcList.class	92†	2904	100% success
JCalc01b	CalculatorList.class	92†	4608	100% success
NumberMonkey	CalcData.class	124‡	2500	100% success
TerpCalc	TCBackendInterface.class	164§	1458	100% success

*Best interface compatibility value = 76.
†One match manually set.
‡Seven matches manually set.
§Eight matches manually set.

third phase, by making the wrapper set to grow up to 4608 in size. As can be seen in Table XI, the set of successful cases for `JCalc01b` is the same as in `JCalc01a`.

NumberMonkey: For this component, the second phase revealed an important situation of mismatching. The reason seem to be an existing difference in the distribution of functionality among the component services. Thus the initial result included three mismatches, which required a manual set of correspondences, hence not to end up the process at this point. The mismatches involve services `putInBuffer`, `evalExpression` and `isNumberInBuffer`. In addition, after doing a thorough inspection of the remaining correspondences, four other required matches were set, involving services with the mathematical operations. After this, the resulting value for Interface Compatibility could be calculated, as can be seen in Table X, which is bigger than the result for the previous two components. This means that there is a bigger *compatibility distance* between `JCalculator` and `NumberMonkey`.

However, for the third phase a smaller set of wrappers (2500) was generated (see Tables X and XI), where more cases of successful results where discovered and one of the wrappers passed successfully 100% of the test cases. The last fact allows to have a certainty of compatibility between the comparing components.

TerpCalc: Similar to `NumberMonkey` component, the Interface Compatibility procedure discovered a high condition of mismatching for this component. The initial result included two mismatches concerning services: `isNumberInBuffer` and `evalExpression`—the latter with a similar case as on the first two versions of `JCalc` component. However, after a deep analysis of the remaining correspondences, six other services required a manual set of matches. Such services are those related to mathematical operations together with services `putInBuffer` and `clear`. The Interface Compatibility result has the highest value, as shown in Table X, which may also indicate about the complexity to integrate `TerpCalc` component as a replacement for `JCalculator`.

The third phase in this case produced the smallest set of wrappers (1458), with the smallest set of successful cases as well, as shown in Tables X and XI. Moreover, from the wrappers set, three wrappers passed successfully 100% of the test cases. Such wrappers involve true correspondences for all services, except for service `clear`, which seems to not affect the execution of the remaining services. Thus, the incorporated manual matching was not really required, since the other two initial matches were already enough.

7.1.2. Incompatible components. Table XII presents a summary of incompatibility reasons discovered for 8 out of the 13 components used to apply the substitutability assessment process. Basically for components `JCalc003a`, `JCalc004a`, `PocketCalc` and `SolCalc`, the problem that has been found is that no back-end interface is provided, just a front-end GUI. This means that those components were not developed to interoperate with other components, but only to work in isolation. For the remaining components, some particular adaptation in order to achieve the required integration could be applied. However, since there is a massive difference with respect to

Table XI. Results of running JCalculator's TS on compatibleComponents.

Case	Success (%)	Wrappers
JCalc01a		
1	100	1
2	77.77	1
3	75	20
4	72.22	20
5	69.44	6
6	52.77	80
7	50	163
9	8.33	1290
10	0	1323
Total		2904
JCalc01b		
1	100	1
2	77.77	6
3	75	25
4	72.22	30
5	69.44	9
6	52.77	165
7	50	271
9	8.33	1779
10	0	2322
Total		4608
NumberMonkey		
1	100	1
2	83.33	4
3	77.77	8
4	75	4
5	61.11	32
6	55.55	16
7	58.33	15
8	52.77	32
9	30.55	64
10	36.11	128
11	8.33	849
12	0	1347
Total		2500
TerpCalc		
1	100	3
2	75	16
3	50	32
4	25	48
5	61.11	32
6	55.55	16
7	11.11	124
8	0	1235
Total		1458

`JCalculator` component, the effort for adaptation may be highly unreasonable. Those components represent calculators with an unexpected complexity, which will not be initially exploited in the targeted system.

7.1.3. Component selection. After a whole view of the results obtained on the subsequent executions of the assessment procedure on the different components, a final decision must be made for selecting one of the components as a replacement for `JCalculator`. Four out of 13 components

Table XII. Incompatible Java calculators w.r.t. JCalculator.

Component	Incompatibility
JCalc003a	Fully user oriented. Input only through GUI. No loose couple back-end
JCalc004a	Almost no difference with JCalc003a
JCalc015	Very different from the other versions. The expected input implies a whole mathematical expression, instead of accepting operands and operators to build an expression and then doing the evaluation and getting a result
GCalc	Not easy to solve a matching. Change of internal representation of a mathematical expression by using a tree structure, due to its purpose which implies the rendering of graphics from mathematical functions
JAC	Similar reason to GCalc, although it does not provide graphical drawing of mathematical functions
JSciCalc	Similar reason to JAC
OpenCalculator	Similar reason to JAC and also JCalc015
PocketCalc	Similar reason to JCalc003a
SolCalc	Similar reason to JCalc003a

resulted with an evident compatibility. From that set of four compatible components, however, the two first versions of `JCalc` component involved the minimum effort on additional setting of manual matching. From these two `JCalc` versions, `JCalc01a` seems to be the initial consideration as a replacement for `JCalculator`. However, the system where the selected component will be integrated has been presented in a situation of evolution (see Section 3.1). Therefore, an attractive option could be the selection of `TerpCalc` component, from its additional functionality related to scientific mathematical functions and the graphical drawing capability.

7.2. *JTopas project*

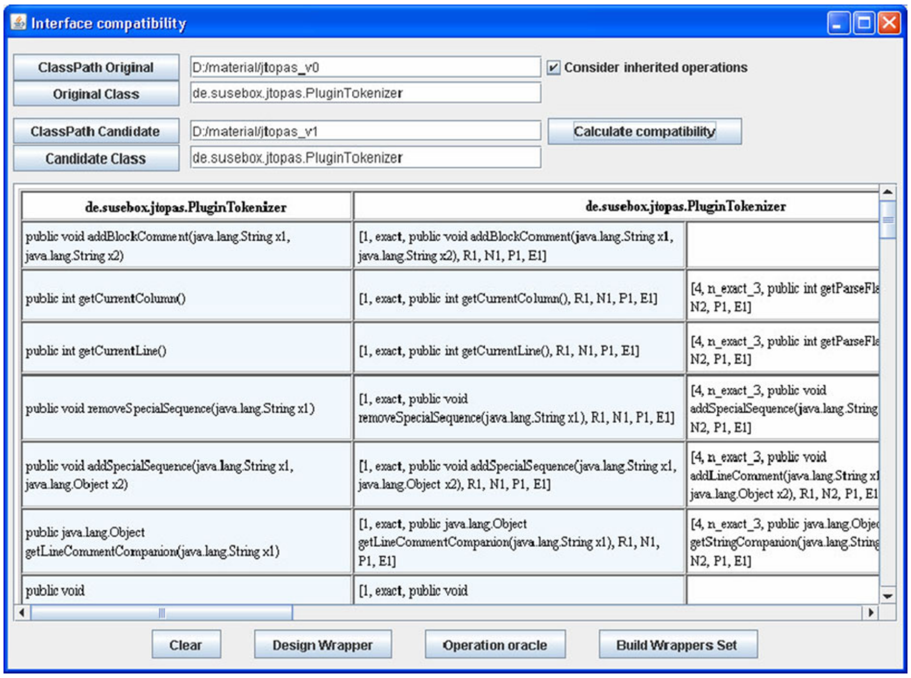
Another experiment was carried out in order to observe the effectiveness of the process. In this occasion, the set of components was downloaded from the Software-artefact Infrastructure Repository (SIR) [42] *http://esquared.unl.edu/sir*, which is a public repository intended to be used as a benchmark for testing experiments.

The `JTopas` Java package 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 component. `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 four versions, from where $version0$ (zero) has been considered as the original component, and the other three versions as the candidate components.

The first phase of the Substitutability Assessment Process was then initiated in order to develop the Component Behaviour TS for *version 0* of `PluginTokenizer`. Together with the project available at the SIR, a TS on JUnit format is also provided, which was used as a base for learning about the component with the intend to develop the corresponding TS. Thus, the initial step for describing the *protocol of use* (to represent operational sequences) has been done to achieve an adequate TS for uncovering the required testing coverage criteria—as discussed in Section 4. As a result of this step, three *test templates* were generated.

The downloaded TS from the project also provided a set of *test data*, which consist of 14 HTML files to be ‘tokenized’. These test data were then combined into the three *test templates* to generate a TS comprising 42 test cases (on JUnit format) that were saved on a file called `JUnitPluginTokenizer`.

After 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.



ClassPath Original: D:/material/jtopas_v0

Original Class: de.susebox.jtopas.PluginTokenizer

[✓] Consider inherited operations

ClassPath Candidate: D:/material/jtopas_v1

Candidate Class: de.susebox.jtopas.PluginTokenizer

de.susebox.jtopas.PluginTokenizer	de.susebox.jtopas.PluginTokenizer	
public void addBlockComment(java.lang.String x1, java.lang.String x2)	[1, exact, public void addBlockComment(java.lang.String x1, java.lang.String x2), R1, N1, P1, E1]	
public int getCurrentColumn()	[1, exact, public int getCurrentColumn(), R1, N1, P1, E1]	[4, n_exact_3, public int getParseFl N2, P1, E1]
public int getCurrentLine()	[1, exact, public int getCurrentLine(), R1, N1, P1, E1]	[4, n_exact_3, public int getParseFl N2, P1, E1]
public void removeSpecialSequence(java.lang.String x1)	[1, exact, public void removeSpecialSequence(java.lang.String x1), R1, N1, P1, E1]	[4, n_exact_3, public void addSpecialSequence(java.lang.String N2, P1, E1]
public void addSpecialSequence(java.lang.String x1, java.lang.Object x2)	[1, exact, public void addSpecialSequence(java.lang.String x1, java.lang.Object x2), R1, N1, P1, E1]	[4, n_exact_3, public void addLineComment(java.lang.String x1 java.lang.Object x2), R1, N2, P1, E1
public java.lang.Object getLineCommentCompanion(java.lang.String x1)	[1, exact, public java.lang.Object getLineCommentCompanion(java.lang.String x1), R1, N1, P1, E1]	[4, n_exact_3, public java.lang.Objec getStringCompanion(java.lang.Strin N2, P1, E1]
public void	[1, exact, public void	

Figure 15. Interface compatibility between *version 0* and 1 of `PluginTokenizer`.

The second phase was then initiated, that is the Interface Compatibility, which is partially shown in Figure 15, where it is particularly displayed with *version 1* of `PluginTokenizer` as the candidate component. As can be seen, the results involved only *exact-matches*. In fact, *version 1* has a bigger interface, which can be perfectly discovered if *version 1* is assumed as the original component. In this case, the table displays dark cells for the additional services.

After *version 0*, the remaining versions actually maintain the same interface for the `PluginTokenizer` class. However, in *version 3* an important change has been done to part of the package, which basically involves some optimizations to their internal functions.

As only *exact-matches* has been found, only one wrapper needs to be generated. The only additional concern may involve those services whose parameter list has a size bigger than one. Some of them include parameters with the same (or equivalent) types—like `addBlockComment` with two *String* parameters (see Figure 15). Those 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 are 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, the need for generating other wrappers is not 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. As four services involve two alike parameters and three others involve six alike parameters, the amount of wrappers by considering that option can actually grow to 3456—according to the wrappers size formula (1) described in Section 6.1.

7.3. *Discussion of results*

Concerning the research hypothesis, the two instances of the experiment give a clear evidence of the capability of the approach to select a replacement component from a set of candidate components.

The testing-based process for component substitution has provided the compatibility degree of all candidate components based on the syntactic and semantic distance, which are, respectively, calculated in terms of the interface and behaviour compatibility.

The experiment consisted of two different substitution cases. The first one involved the substitution of a given component by considering a set of candidates including both upgrades from the same provider and completely different components from other providers. As a result, four fully compatible components were identified (shown in Tables X and XI). The second one is referred to the substitution of a component by a new version (upgrade) proceeding from the same vendor, where a perfect match was also discovered.

Therefore, the applicability of this approach is not constrained only to evaluate subsequent releases of a certain component, but also to include components from different providers. However such an application requires a previous searching step for potential components, in order to establish a set of candidates to be considered for evaluation. Therefore, when such set of candidates does not contain any component offering neither a compatible interface nor a similar behaviour, the process will certainly provide a negative result, thus preventing a lower-grade on the current system stability.

With these considerations, the research hypothesis has been visibly confirmed, since the testing-based process for component substitution makes possible to select a replacement component for an original component with a measurable degree of compatibility.

7.4. *Threats to the validity of the experiment*

This section discusses several issues that could threaten the validity of the experiment and how they have been alleviated.

7.4.1. Construct validity. The construct validity is the 'degree to which the independent and the dependent variables are accurately measured by the measurement instruments used in the experiment' [43].

In this experiment, the dependent variables are defined by the syntactic and semantic distance, which are, respectively, calculated by means of the interface and behaviour compatibility. For interface compatibility, the independent variables involve the basic elements from services signatures, which are syntactically compared to measure the matching level for interface compatibility. For the behaviour compatibility, the independent variable is deterministically counted as the number of faults in each wrapper.

7.4.2. Internal validity. The internal validity is the degree of confidence in a cause–effect relationship between factors of interest and the observed results [43].

According to the nature of both experiments—automatic and deterministic generation of a TS, comparison of components interfaces, generation of a set of wrappers, execution of test cases and deterministic result analysis—all variables have been controlled, and therefore threats to internal validity are minimized.

7.4.3. External validity. The external validity is the 'degree to which the research results can be generalized to the population under study and other research settings' [43]. The greater the external validity, the more the results of an empirical study can be generalized to actual software engineering practice.

In both case studies, components were downloaded from public repositories. In the first case study, 13 components were used, which involved a wide range of different features to deal with. In the second case study, four versions of a component from a different domain were evaluated, which presented a more complex functionality and interface. Therefore, a wide experience has been gained with component samples from different domains and also different sizes, each providing a representative case which is likely to occur in actual scenarios of substitution.

Even when the approach seems to be good enough for managing components' substitution, more experimentation could certainly be very rewarding. Nevertheless, since dependent and independent variables are associated with a clear and objective set of methods and equations, the whole approach presents an appropriate validity. Therefore, the main effort could be focused on improving the user-friendly automation of the approach, in order to release a version which could be finally adopted by the software industry.

8. CONCLUSIONS

The approach presented in this paper is focused on the maintenance stage where component-based systems require updates by replacing certain components with new releases (upgrades) or completely different software units (i.e. components from a different vendor). The proposal involves a specific process that uses testing coverage criteria to describe components' behaviour with the purpose to analyse compatibility on candidate replacement components. In this way, the Substitutability Assessment Process presented in this paper integrates two aspects: component selection and also testing tasks. Therefore, the approach provides integrators in a concrete manner to reduce their usual effort under the support of a reliable process.

Automation of the whole process is currently supported by the *testooj* tool, which helps to reduce time and effort and also reinforces control over conditions of each phase in order to achieve a rigorous approach. However, an additional experience was achieved in a previous work [11] with a similar prototype of this tool implemented in the .Net framework, where a reflection mechanism is also available which could therefore easily allow the application of the process.

A subject that requires additional advance concerns scalability upon the generation and management of wrappers (as mutation cases). Although some heuristics were already defined, an enhanced set of heuristics could highly improve the efficiency of the approach. Another helpful strategy may imply to minimize branching in the wrappers generation tree, which could be achieved by searching repeated nodes on each path from the tree, that is, a service from a candidate component which is used more than once in correspondence to different services from the original component—e.g. the service `addPlus` in the second path (to the left) from the tree in Figure 12 (Section 6.1).

Additionally, the particular features concerning testing strategies, which are the basis of this approach, give high opportunities to automate other aspects from the generation of the TS. For instance, the protocol of use could be derived from a monitoring mechanism of the actual interactions of components within a system. In this case the generation of a log file at run-time could be an effective manner, where either components could be instrumented or an intermediate wrapper focused on the logging task could be used as a non-invasive strategy.

Finally, another issue also related with efficiency concerns the size of the TS, for which appropriate reduction strategies might give the chance to work with a smaller set of test cases without affecting the required efficacy and reliability of the process. This may involve to select a different combination algorithm from those mentioned in Section 4.1, a reduced set of test data, and even to design the TS with a less restrictive testing criteria—like covering the *all-operators* criterion instead of *all-expressions*, according to what has been highlighted in Section 4.1.

REFERENCES

1. Heineman G, Council W. *Component-based Software Engineering—Putting the Pieces Together*. Addison-Wesley: Reading, MA, 2001.
2. Warboys B, Snowdon B, Greenwood R, Seet W, Robertson I, Morrison R, Balasubramaniam D, Kirby G, Mickan K. An active-architecture approach to COTS integration. *IEEE Software* 2005; **22**(4):20–27.
3. Cechich A, Piattini M, Vallecillo A. *Component-based Software Quality*: *Methods and Techniques* (*Lecture Notes in Computer Science*, vol. 2693). Springer: Berlin, 2003.
4. Jaffar-Ur Rehman M, Jabeen F, Bertolino A, Polini A. Testing software components for integration: A survey of issues and techniques. *Software Testing*, *Verification and Reliability* 2007; **17**(2):95–133. Available at: http://www.interscience.wiley.com.
5. Mariani L, Papagiannakis S, Pezzè M. Compatibility and regression testing of COTS-component-based software. *Twenty-ninth International Conference on Software Engineering* (*ICSE'07*), Minneapolis, U.S.A. IEEE Computer Society Press: Silver Spring, MD, 2007; 85–95.

6. Stuckenholz A. Component evolution and versioning state of the art. *ACM SIGSOFT Software Engineering Notes* 2005; **30**(1):7–20.
7. Flores A, Polo M. Towards software component substitutability through black-box testing. *Fifth Workshop on System Testing and Validation* (*STV'07*), *During 20th International Conference on Software and Systems Engineering and their Applications* (*ICSSEA'07*). Fraunhofer IRB Verlag: Paris, France, 2007; 111–120.
8. Freedman RS. Testability of software components. *IEEE Transactions on Software Engineering* 1991; **17**(6): 553–564.
9. Edwards SH. A framework for practical automated black-box testing of component-based software. *Software Testing*, *Verification and Reliability* 2001; **11**:97–111. Available at: http://www.interscience.wiley.com.
10. Polo M, Tendero S, Piattini M. Integrating techniques and tools for testing automation. *Software Testing*, *Verification and Reliability* 2007; **16**(1):1–37. Available at: http://www.interscience.wiley.com.
11. Flores A, Garcia I, Polo M. Net approach to run-time component integration. *Third Latin American Web Congress* (*LA-WEB'05*), Buenos Aires, Argentina. IEEE Computer Society Press: Silver Spring, MD, 2005; 45–48.
12. JUnit Home Page. JUnit.org resources for test driven development, 2008. Available at: http://www.junit.org/home.
13. μJava Home Page. Mutation system for Java programs, 2008. Available at: http://cs.gmu.edu/~offutt/mujava/.
14. Szyperski C. *Component Software*: *Beyond Object-oriented Programming* (2nd edn). Addison-Wesley: Reading, MA, 2002.
15. Wu Y, Chen MH, Offutt J. UML-based integration testing for component-based software. *Second International Conference on Cots-based Software Systems* (*ICCBSS'03*) (*Lecture Notes in Computer Science*, vol. 2580), Ottawa, Canada. Springer: Berlin, 2003; 251–260.
16. Mann S, Borusan A, Ehrig H, Große-Rohde M, Mackenthun R, Sunbul A, Weber H. Towards a component concept for continuous software engineering. *Technical Report*, Fraunhofer ISST, Berlin, Germany, October 2000.
17. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns*: *Elements of Reusable Object-oriented Software*. Addison-Wesley: Reading, MA, 1995.
18. Riggs R. Java$^{\text{TM}}$ product versioning specification. *Technical Report—Online*, SUN Microsystems Inc., November 1998. Available at: http://java.sun.com/j2se/1.5.0/docs/guide/versioning/spec/versioningTOC.html [November 2007].
19. Ghosh S, Mathur AP. Interface mutation. *Software Testing*, *Verification and Reliability* 2001; **11**:227–247. Available at: http://www.interscience.wiley.com.
20. Wu Y, Pan D, Chen MH. Techniques for testing component-based software. *Seventh International Conference on Engineering of Complex Computer Systems* (*ICECCS'01*), Skovde, Sweden. IEEE Computer Society Press: Silver Spring, MD, 2001; 222–232.
21. Wu Y, Pan D, Chen MH. Techniques of maintaining evolving component-based software. *Sixteenth International Conference on Software Maintenance* (*ICSM'00*), San Jose, CA, U.S.A. IEEE Computer Society Press: Silver Spring, MD, 2000; 236.
22. Orso A, Do H, Rothermel G, Harrold MJ, Rosenblum D. Using component metadata to regression test component-based software. *Software Testing*, *Verification and Reliability* 2007; **17**:61–94. Available at: http://www.interscience.wiley.com.
23. Atkinson C, Grob HG, Barbier F. Component integration through built-in contract testing. *Component-based Software Quality*: *Methods and Techniques* (*Lecture Notes in Computer Science*, vol. 2693). Springer: Berlin, 2003.
24. Alexander R, Blackburn M. Component assessment using specification-based analysis and testing. *Technical Report SPC-98095-CMC*, Software Productivity Consortium, Herndon, VA, U.S.A., May 1999.
25. Cechich A, Piattini M. Early detection of COTS component functional suitability. *Information and Software Technology* 2007; **49**(2):108–121.
26. Mariani L, Pezze M, Willmor D. Generation of integration tests for self-testing components. *International Workshop of Testing Methodologies* (*ITM'04*) *held at FORTE'04* (*Lecture Notes in Computer Science*, vol. 3236), Toledo, Spain. Springer: Berlin, 2004; 337–350.
27. The Java$^{\text{TM}}$ utorials. Lesson: Regular expressions. Sun Microsystem Inc., 2008. Available at: http://java.sun.com/docs/books/tutorial/essential/regex/index.html.
28. Binder R. *Testing Object Oriented Systems—Models*, *Patterns and Tools*. Addison-Wesley: Reading, MA, 2000.
29. OMG. Unified modeling language: Superstructure version 2.0. *Technical Report*, Object Management Group, Inc., 2005. Available at: http://www.omg.org.
30. Kirani SH, Tsai WT. Method sequence specification and verification of classes. *Journal of Object-oriented Programming* 1994; **7**(6):28–38.
31. Myers GJ. *The Art of Software Testing* (2nd edn). Wiley: New York, 2004.
32. Ammann P, Offutt A. Using formal methods to derive test frames in category-partition testing. *Ninth Annual Conference on Computer Assurance* (*COMPASS'94*), Gaithersburg, MD, U.S.A. IEEE Computer Society Press: Silver Spring, MD, 1994; 69–80.
33. Malaiya Y. Antirandom Testing: Getting the most out of Black-box Testing. *International Symposium on Software Reliability Engineering* (*ISSRE'95*), Toulouse, France. IEEE Computer Society Press: Silver Spring, MD, 1995; 86–95.
34. Czerwonka J. Pairwise testing in real world. *Twenty-fourth PNSQC*, Portland, OR, U.S.A., 2006; 419–430.
35. Grindal M, Offutt A, Andler S. Combination testing strategies: A survey. *Software Testing*, *Verification and Reliability* 2005; **15**(3):167–199. Available at: http://www.interscience.wiley.com.

36. Offut A. A practical system for mutation testing: Help for the common programmer. *Twelfth International Conference on Testing Computer Software*, Washinton, DC, U.S.A., 1995; 99–109.
37. MuClipse Home Page. Eclipse Plugin for the MuJava mutation engine, 2008. Available at: http://muclipse.sourceforge.net/.
38. Smith B, Williams L. An empirical evaluation of the MuJava mutation operators. *Testing*: *Academic and Industrial Conference Practice and Research Techniques* (*TAICPART-MUTATION*), Windsor, U.K., 2007; 193–202.
39. Gosling J, Joy B, Steele G, Bracha G. *Java^{TM} Language Specification* (3rd edn). Sun Microsystems Inc., Addison-Wesley: U.S.A., 2005. Available at: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.
40. Zaremski AM, Wing J. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology* 1997; **6**(4):141–172.
41. Delamaro M, Maldonado J, Mathur A. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering* 2001; **27**(3):228–247.
42. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*: *An International Journal* 2005; **10**(4):405–435.
43. Wohlin C, Runeson P, Höst M, Ohlsson M, Regnell B, Wesslén A. *Experimentation in Software Engineering*: *An Introduction*. Kluwer Academic: Norwell, MA, 2000.