Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1335–1344 DOI: 10.15439/2017F139 ISSN 2300-5963 ACSIS, Vol. 11

Interface-based Semi-automated Testing of Software Components

Tomas Potuzak
Department of Computer Science, Faculty of Applied Sciences, University of West Bohemia, Univerzitni 8, 306 14 Plzen, Czech Republic
Email: tpotuzak@kiv.zcu.cz

Richard Lipka, Premek Brada
NTIS – New Technologies for the Information Society, European Center of Excellence, Faculty of Applied Sciences, University of West Bohemia, Univerzitni 8, 306 14 Plzen, Czech Republic
Email: {lipka,brada}@kiv.zcu.cz

***Abstract*—The component-based software development enables to construct applications from reusable components providing particular functionalities and simplifies application evolution. To ensure the correct functioning of a given component-based application and its preservation across evolution steps, it is necessary to test not only the functional properties of the individual components but also the correctness of their mutual interactions and cooperation. This is complicated by the fact that third-party components often come without source code and/or documentation of functional and interaction properties. In this paper, we describe an approach for performing rigorous semi-automated testing of software components with unavailable source code. Utilizing an automated analysis of the component interfaces, scenarios invoking methods with generated parameter values are created. When they are performed on a stable application version and their runtime effects (component interactions) are recorded, the resulting scenarios with recorded effects can be used for accurate regression testing of newly installed versions of selected components. Our experiences with a prototype implementation show that the approach has acceptable demands on manual work and computational resources.**

I. Introduction

THE component-based software development is an important part of contemporary software engineering. It is based on the utilization of isolated reusable parts of the software (called *software components*), which mutually provide and require services (i.e., functionalities) using public interfaces. A component can be utilized in multiple applications and, at the same time, an application can be constructed from components created by different developers [1]. This reinforces the necessity for testing.

The functionality of an individual component should be tested primarily by its developer. However, it is also necessary to test the functionality of the entire component-based application where the correct cooperation of the components is no less important. The situation is complicated by the fact that many components exist in several versions.

The versions of a single component can differ by the internal behavior (different computations), by external behavior (different interactions with other components), or by the interface (different provided and required services). Theoretically, the change of internal behavior of a component should not affect the behavior of the entire application. Nevertheless, in reality, the change can introduce an unwanted error into the new version, add or remove side effects of some method invocations, prolong computation, which can cause a time-out to expire, and so on. When installing a new version of a component to a functional component-based application, adequate regression testing is, therefore, desirable even when there are no apparent external changes of the component.

The usually performed manual testing is a lengthy and costly process and its automation is desirable wherever possible. In this paper, we describe an approach for semi-automated regression testing of software components whose source code is not available (e.g., third party components). The approach is suited for checking whether a newly installed version of a component exhibits the same behavior within a component-based application as its old version. The approach uses static analysis of the component implementations and employs methods of aspect-oriented programming and stochastic testing to record runtime behavior of the application with the old and new version of a component. The comparison of both recordings can then reveal possible different behaviors and thus support debugging on the architectural level.

The description of the approach along with its validation on two case studies is the main contribution of this paper. Its structure is as follows. The following section provides an overview of the basic notions and Section III discusses related work in component analysis and testing. Section IV covers the details of the proposed approach. Section V presents its validation including an analysis of performance implications, and Section VI summarizes the contribution and future work.

This work was supported by Ministry of Education, Youth and Sports of the Czech Republic, project PUNTIS (LO1506) under the program NPU I.

II. Software Components

In component-based software development, the applications are sets of individual software components.

A. Basic Notions

A software component is a black-box entity, which provides services to other components via its well-defined interfaces and may require services of other components in order to function. The inner state of a component is not observable from the outside. So, the components are expected to mutually interact solely using their interfaces. A component should be reusable (i.e., it can be used in multiple applications) and, at the same time, a component-based application can be constructed from components created by different providers. These features are common to the majority of software components regardless of the component model [1].

A component model prescribes the behavior, interactions, and features of its software components and is implemented by (usually several) component frameworks. The experimental implementation of the interface-based component testing approach described in this paper has been created for the OSGi component model. OSGi [2] is a dynamic component model for the Java programming language. It is currently widespread in both industrial and academic spheres making it a good choice for experimentation. There are several commonly used implementations of the OSGi component model (i.e., OSGi frameworks), for example Equinox [3] or Felix.

In OSGi, the components are referred to as *bundles*. Each bundle has the form of a single Java `.jar` file with additional information related to the OSGi component model (e.g., name of the bundle, version of the bundle, lists of provided/required packages, etc.) [2]. Each bundle can provide one or more services represented by standard Java interfaces. Together, the classes in exported packages and the provided services form the accessible interface of the OSGi components.

The dynamic nature of the OSGi means that the bundles can be installed, started, stopped, and uninstalled without the necessity to restart the OSGi framework [4]. For this purpose, the OSGi framework runtime provides standard methods [2] for the exploration of the bundle's context (i.e., environment) and the control of its life cycle.

B. Testing of Software Components

Testing of individual software components is similar to testing of ordinary monolithic software applications. However, the extra problems, which can be caused by the third party composition, need to be considered.

Generally, the testing methods can be divided according to the available knowledge of the tested software [5]. If its source code is known, it can be (and usually is) used for the preparation of the testing, leading to the *white-box testing*. If their source code is unknown or not considered in test preparation (the *black-box testing*), other resources can be used for test preparation such as descriptions of the expected software behavior, the definition of its user and application interfaces, and so on [6]. The source code is often unavailable when we want to utilize a third party component in our component-based application and we want to test it first (both individually and as a part of our application).

Regardless the type of the testing, its principle lies in subjecting the tested component(s) to a set of stimuli and observing the congruence of their reactions with the expected ones [7]. In most real situations, it is not feasible to test the responses to all possible stimuli. Instead, a subset of all possible stimuli is used. In that case, it is important to ensure that the stimuli of the subset represent well the complete set of stimuli and various methods for the subset creation are used in practice [5].

An important criterion of the testing is its *coverage*, i.e. the amount of implementation code exercised by the tests. In the case of black-box component testing, coverage can be measured by the different invocations of individual operations on both provided and required sides of the component interface, considering also the actual parameter values. An important constraint is that it must be possible to achieve good coverage using the chosen subset of stimuli in a reasonable time [6].

The test design is usually described in so-called *scenarios* containing the stimuli and (optionally) the expected effects. Considering the testing of software components with unknown source code (i.e., black-box testing), each stimulus corresponds to an invocation of a service method provided by the tested component. The effects can be for example the return of a value or an invocation of an (outgoing) operation through the required side of component's interface.

When the scenario is executed, manually or in an automated way, the actual effects are compared to the expected ones to establish whether the component complies with the behavior specified by the scenario. Automated testing allows the scenarios to be executed repeatedly, which is important for the regression testing verifying whether new versions of components exhibit the same – or equivalent – behavior as the previous version(s). This aspect is important with respect to the highly flexible composition of components by third parties where the component provider cannot foresee the ultimate configuration of the component-based applications.

III. Related Work

The approach for the semi-automated testing of software components with unknown source code is related to several existing approaches, which are described below.

A. Behavioral-Diagram-based Scenarios Generation

Many approaches to testing scenarios generation are based on behavioral diagrams of UML (e.g., activity diagrams, sequence diagrams, state machine diagrams, etc.) [8]. The approaches described below are not intended for utilization with software components, because such examples are rare.

The activity diagrams are used for example in [6] for object-oriented applications. There, these diagrams representing concurrent activities (corresponding to method invocations) in an application are exhaustively explored. The scenarios are generated during the exploration. Because the exploration of all possible flows in the diagrams is infeasible for large applications, there are some constraints based on the application domain. These constraints are used to discard illegal or irrelevant scenarios [6].

The activity diagrams are also used in [8] where they are generated from multiple UML use case diagrams. Their purpose is to express the concurrency of the use cases. The exploration of the created activity diagrams is again used for the generation of the testing scenarios. The approach is intended for object-oriented applications [8].

B. Natural-Language-based Scenarios Generation

The generation of the testing scenarios from a specification in natural language is an appealing approach. Nevertheless, this approach is still difficult to implement because of the poor understandability, ambiguity, incompleteness, and inconsistency of natural language [9]. To overcome these difficulties, a set of restrictions is commonly used.

A restricted form of natural language is used for descriptions of the use cases in [10]. From them, a control-flow-based state machine is created for each use case. These state machines are then combined into a single global system level state machine. The testing scenarios are then created by this state machine exploration [10].

A similar approach was considered in our previous research focused on the simulation-based testing of software components based on the descriptions of use cases written in natural language (see Section III.E). These descriptions are transformed into an overall behavioral automaton (OBA) using the FOAM tool [11]. Using the OBA, the testing scenarios can be generated. The restrictions of the approach lie in the descriptions of uses cases, which must conform to the rules described in [12], and in the necessity to manually enrich the descriptions of use cases with the annotations describing the flow of the program and its temporal dependencies, as well as connections between the actions in use cases and the corresponding method invocations [13].

C. Interface-Probing-based Scenarios Generation

Interface probing is an approach, which utilizes the public interface of a software component (or another piece of software with a defined interface) for the examination of its behavior. This approach does not require the source code or any knowledge of the internal working of the software component. So, it is convenient for the black-box testing. Its basic idea is used in our approach as well (see Section IV).

Using the interface probing, the interface of the component – the services of the component and their methods – is identified first. Then, the input values for the methods are generated and the methods are invoked using them. The outputs of the methods are then observed [14], [15]. For this purpose, the tested component can be wrapped in an encasing object controlling the input and output data flows [16].

A disadvantage of this approach is the necessity to generate the input values. This can be done randomly, systematically, or manually. In any case, it is possible that input values will be omitted, which are in fact important for examination of the behavior of the tested component [14]. The programmer therefore needs to instruct the generator on suitable value ranges, using a set of the test design methods [5] and based on other descriptions (e.g., Javadoc) of the component if available.

D. Static Byte Code Analysis

The static byte code analysis is an example of checking of the applications constructed from software components of various developers for type inconsistencies. These inconsistencies can arise even in statically-typed languages (e.g., Java), considering the component-based application. Because each component is compiled separately, the mutual dependencies of the components are not considered by the compiler [17].

A solution of this issue proposed in [17] is the byte code analysis. It consists of three steps – the discovery of component dependencies, the matching of component dependencies, and the consistency verification. All the information necessary for all steps is extracted from the byte code. During the analysis, a graph representing the dependencies of the components is created. The graph is then traversed and the particular dependencies are checked for the type compatibility.

Although this approach represents a reliable method for the static determination of the compatibility of software components of a single component-based application, it cannot detect problems, which are not type-related. For example, if a method returns `null` instead of an expected instance, the problem will not be detected [17].

E. Simulation Testing

The basic idea of the approach described further in this paper (see Section IV) was already utilized for the simulation testing of software components during our previous research [18]. This approach was based on the testing of real software components in a simulated environment. The testing was based on a discrete-event simulation when the individual events corresponded to the invocations of the particular methods of the tested component. Simulated and intermediate components were used to observe and record the behavior of the tested component [18].

The issues of this approach were the discrepancies of the tested software components running in a real time and the simulation running in a discrete time and the necessity to create the simulation and intermediate components [18]. Hence, the discrete simulation was abandoned and the entire process was significantly simplified. The result is the approach described further in this paper (see Section IV).

IV. Interface-based Component Testing

As was mentioned above, the main theme of this paper is an approach for the black-box regression testing of software components in a component-based application – the interface-based component testing. During the regression testing, we are determining whether the application with the old and the new version of a component exhibits the same behavior.

In this section, we describe its idea, the model of application it uses, and the particularities of a prototype implementation. The overall process works as follows. It starts with the analysis of the interfaces, services, and methods of software components of a component-based application. Based on this analysis, the sets of invocations of the particular methods are generated and then performed. The consequences of each invocation are observed and recorded. The result is a testing scenario with actions and their consequences. The scenario can be then used to check whether a newly installed version of a component in a component-based application exhibits the same behavior as its old version.

The details of the key parts of this process are discussed below, including experimental implementation aspects.

A. The Invocations-Consequences Data Structure

Usually, our method assumes that the entire component-based application is subject to the testing, because the components inside a single application interact with each other and these interactions are important to uncover the behavior of the components. This is the main difference from the interface probing method (see Section III.C) where each component is tested alone.

The service methods need to be determined for all the components of the tested application. This can be in general done by any method, which is able to recover complete method signature information. The components, their services, and their methods are explored and their identifiers are inserted into a tree data structure (see Fig. 1a).

An initial set of invocations for each of the methods thus determined is generated and added to the tree data structure. A set of test data values for each parameter of the method has to be provided in conjunction; each invocation is created as a unique permutation of the values of all parameters of the method.

In the current implementation of the approach (see Section IV.C), we use fully automatic generation of parameter values with a rather straightforward approach to cover the main test cases. The generated values depend on the parameter types. For primitive types, several representative and border values are generated. For general objects, only `null` value is used.

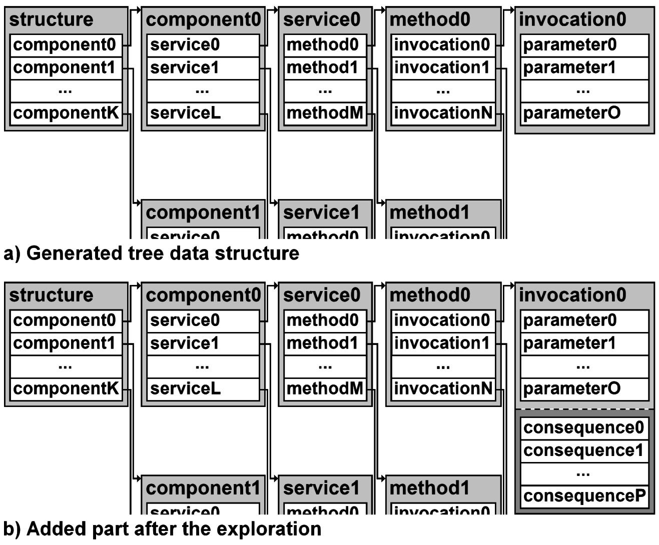


Fig. 1 The tree data structure

Additionally, when the user knows, which parameter values are critical for the tested component application, he or she can select any method, set the required parameter values and thus create and add a new invocation. The user can also restrict the set of generated values, where appropriate.

In principle, it would not be necessary to use all methods for the initial invocation set to achieve good coverage. Since the source code is unknown however, the (side) effects of the methods execution are unknown as well. Hence, with the utilization of all methods, the probability of better exploration of the behavior of the components is higher than if some methods were excluded.

Once the initial invocations are added to the tree data structure, they are performed (i.e., the application is executed in a testing mode) and their consequences (effects) are observed and recorded. The consequences of a method invocation are: the return of a value, a raised exception, a value change in the "out" parameters of the method, a subsequent invocation of a service method of another (depended on) component, and a change of the inner state of the component. The change of the inner state of the component is different from the others consequences, because it is not easily observable. Hence, it is not considered by our method.

There can be more than one consequence per an invocation of a method. All consequences are recorded and inserted into the tree data structure (under the invocation, which caused them), but only if they are not already present. Each invocation consequence record incorporates its type and further data depending on this type (e.g., the return value, the instance of an exception, etc.).

The subsequent (outgoing) invocations are the most important consequences. Each subsequent invocation is described by the method it is invoking and the unique permutation of its parameter values. When a subsequent invocation of a method is performed such that has not yet been observed, it is recorded along with its parameter values.

This invocation is then added to the generated and already recorded invocations in the tree data structure. These invocations are valuable, because their parameter values are genuine, created by the internal logic of the component, which invoked the method. They can for example contain instances of objects, which would be difficult to generate automatically. Again, this is a substantial difference from the interface probing method (see Section III.C).

The disadvantage of recording the subsequent invocations in this way is that we cannot be entirely sure what their actual cause was. As the components under tests are black box, we cannot create their full behavioral model and determine the causal relation between the method invocations. For example, if a component uses active threads, it can perform invocations on other components independently on the incoming invocations performed on its service methods. The invocations performed by such (internal) threads can still be intercepted and added to the tree data structure. Even though there is no causal relation between them and the incoming invocation, which preceded them chronologically, a false cause-effect relation is still recorded in the tree data structure. This may cause false alarms during the comparison of the tree data structures (see Section IV.B), because the corresponding invocation-consequence pairs may not occur in further application executions. The mitigation of this problem can be repeating the invocation and further analysis; it is a part of our future work.

In order to maximally exploit the subsequent invocations during the testing, the invocation-driven exploration of the tree data structure repeats several times. The subsequent invocations generated in nth exploration can be performed in the $(n + 1)$th run and its consequences observed. When no new consequences are generated, the exploration ends.

In the final tree structure, all invocations and consequences contain the number representing the iteration, in which they were inserted (starting with 1). The initial generated invocations or provided by the user prior to the exploration of the structure are numbered 0. The filled tree data structure is therefore enriched by the invocation consequences (see Fig. 1b) and by the invocations extracted from the subsequent invocations. This structure can be then saved to a file as a testing scenario.

B. Testing Application Evolution: Comparison of Tree Data Structures

The stored tree data structure is useful when a new version of a component is installed to the component-based application. In this case, it can be tested whether the application with the new version of the component exhibits the same behavior as the old version (regression testing). For this purpose, the entire process described in Section IV.A is performed for the application with the new version of the component. The result is again the filled tree data structure.

The original tree data structure (corresponding to the behavior of the application with the old version of the component) is then loaded from the file and the structures are compared. The comparison is performed on each level of the two tree data structures, which use the same set of initial invocations, starting from the component level.

If a component is only in one tree data structure, this difference is reported and the services of this component are not considered further. For each pair of corresponding components, their services are compared by their names. If a service is only in one tree data structure, this difference is reported and the methods of this service are not considered further. For each pair of corresponding services, their methods are compared by their signatures. If a method is only in one data structure, this difference is reported and the invocations of this method are not considered further. Analogically, this continues down to the invocation consequences level (see Fig. 2 for examples).

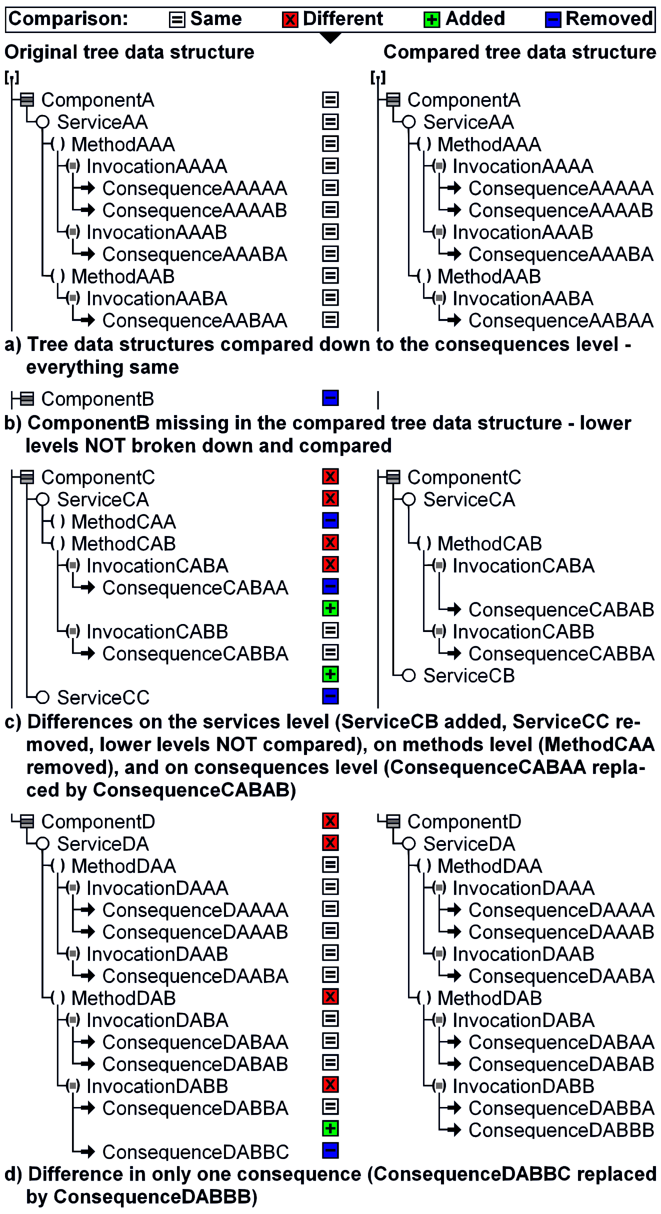

Comparison: Same Different Added Removed
Original tree data structure
Compared tree data structure
a) Tree data structures compared down to the consequences level - everything same
b) ComponentB missing in the compared tree data structure - lower levels NOT broken down and compared
c) Differences on the services level (ServiceCB added, ServiceCC removed, lower levels NOT compared), on methods level (MethodCAA removed), and on consequences level (ConsequenceCABAA replaced by ConsequenceCABAB)
d) Difference in only one consequence (ConsequenceDABBC replaced by ConsequenceDABBB)

Fig. 2 Comparison of two tree data structures

The differences are expected to occur mainly in the invocations and invocation consequences levels of the tree data structure. Such a difference means that the tested component-based application with the new version of the component exhibits a different behavior. A difference on the methods level or on the services level implies the change in the public interface of the component. Our method is of course capable to detect these changes, but they can be detected also by the other means such as advanced methods of static analysis (e.g., see [17]). However, discovering differences in the invocations and consequences level by the static verification is difficult. Since these differences can mean significant problems in the tested application, the described interface-based component testing add significant value to the functional verification process.

C. Experimental Implementation and Its Discussion

The approach has been implemented for the Java programming language and the OSGi component model; however, the main ideas behind it can be used for other component models and programming languages as well. The implementation is a part of our Interface Analysis Tool (InAnT), which has the form of a single component (OSGi bundle) and is expected to be installed in the same OSGi framework as the tested component-based application.

The interface-based testing begins when the InAnT component and the components of the tested application are started. The tool searches for all available components and their registered services using standard methods of the OSGi framework for this purpose. A proxy is then automatically created and assigned to each registered service of the components. This is achieved using standard OSGi hooks [2]. Each future invocation of a method of a service is subsequently mediated by the corresponding proxy, which records the invocation and executes it on the corresponding component using Java reflection. This way, all method invocations performed within the tested component-based application can be detected and traced.

The proxies are similar to but much simpler than the intermediate components used in simulation testing (see Section III.E and [18]). There is a single generic proxy implementation whose sole purpose is to record method invocations and, for each registered service, there is one proxy instance. The intermediate components, on the other hand, incorporate functionalities related to the running of the components in a simulated environment [18]. Moreover, each service has an intermediate component designed specifically for it.

There can be more than one independent component-based application deployed in one OSGi framework. So, the user is encouraged to select the components of the application intended for the testing. It is possible to select only some components of the application, but the testing is then likely to be incomplete. Once the components for the testing are selected, the methods of their services are determined using Java reflection (since the services correspond to standard Java interfaces – see Section II.A)

Since the framework services do not guarantee that the components are discovered in the same order across different framework runs, the components are sorted by bundle symbolic name in the tree data structure. Similarly, particular services of each component and particular methods of each service are sorted as well. This way, it is ensured that the initial set of invocations is always executed in exactly the same order. This is the necessary condition for the correct comparison of two tree data structures.

The automatic generation of the invocations and their parameter values is deterministic and repeatable. Therefore, it does not negatively influence the comparison. However, when the user adds invocations manually or restricts the range of the parameter values of the automatically generated invocations, it is vital that he or she uses the same settings (including the order of manually added invocations).

As it was described in Section IV.A, the parameter values generated by our implementation depend on the parameter types. For the number types, a set of representative values is generated. These values include the maximal and minimal possible values, 0, -1, 1, and several negative and positive values with a constant step. The size of the step can be selected by the user and can significantly influence the number of values and consequently the number of generated invocations. For the `boolean` type, both possible values are generated. For the `char` type, several single-byte values (corresponding to letters, digits, punctuation, and non-printable characters) and several double-byte values are generated. For `enum` types, all possible values including the `null` value are generated. For the `String` class, the `null` value and the empty string are generated. For the wrapping classes of the primitive types (e.g., `Integer` for `int` or `Character` for `char`), the same values as for the primitive data types and `null` value are generated. For the remaining classes, only the `null` value is generated.

The invocations of the tree data structure are then performed as described in Section IV.A. Their consequences are observed directly (the return of a value, a raised exception, a value change in parameters of the method) or indirectly by the proxies (a subsequent invocation).

The filled tree data structure can be then stored to a specific XML file. The XML format was chosen because of its hierarchical nature and legibility for humans, which is useful during the development. The stored filled tree data structure can be compared to another stored filled tree data structure or to filled tree data structure created in the memory. The comparison is performed as described in Section IV.B.

V. Validation and Results

The described interface-based component testing approach was validated by two sets of tests. The first one was focused on the dependency of the number of generated invocations on the number of methods and the number of their parameters; this is described in Section V.B. The second set of tests was focused on the testing the ability of the approach to discover the different behavior of the tested application when a new version of a component was installed (see Section V.C).

A. Environment and Application Used for Testing

All tests were performed on a single desktop computer with quad-core Intel i7-4770 CPU at 3.40 GHz, 16 GB of RAM, and 1TB HDD. The software environment consisted of the operating system Windows 7 SP1 (64 bit), Java 1.6 (32 bit), and the Equinox OSGi framework.

A component-based application of our own design was used for the first set of tests. We chose not to use a 3rd party application to facilitate the analysis and manipulation of the test application source code (e.g., add and remove methods and their parameters, change the behavior of the methods, etc.) in order to test various features of the approach. The test application is a simple tool for mathematical calculations and processing of strings. It consists of five components (see Fig. 3). The `Utilities` component represents the interface of the entire application. The methods of its service perform high-level operations. The `Text` component provides a service for the processing of strings and utilizes the `Calculator` component for mathematical operations. The `Calculator` component provides a service for mathematical operations including geometrical transformations. For this purpose, it utilizes the `Geometry` component. The `Logger` component logs the running of the `Utilities` component.

For the second set of tests, the test application was used as well, but, additionally, we used the well known CoCoME (Common Component Modeling Example – see [19]) application to demonstrate that our interface-based component testing approach is able to work with a real world application. The implementation, which was used for the testing, was developed internally in our group.

B. Dependency on Number of Methods and Parameters

From the description of the generation of the invocations for the methods in the tree data structure (see Section IV.C), it is obvious that the number of generated invocations grows very rapidly. We expected that it grows linearly with the total number of methods and exponentially with the number of parameters of each single method. In order to verify the assumption, a set of tests was performed.

First, the dependency of the number of generated invocations on the number of parameters of a single method

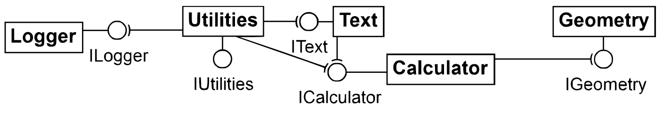


Fig. 3 The component-based application used for the testing

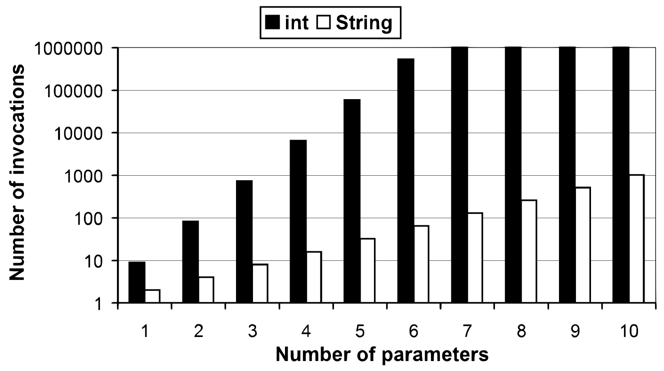
was investigated. The `Utilities` component was used for this purpose. All its methods were removed and a testing method was added. The method had between one and ten parameters, either all `int` parameters or all `String` parameters. When the number of parameters was changed, the component was recompiled and the invocations were generated for it.

The results are depicted in Fig. 4a. As expected, the dependency on the number of parameters is exponential (note the logarithmic scale of the y-axis). It can be also observed that the increase in the number of generated invocations is far steeper for the `int` parameters than for the `String` parameters. In fact, an out of memory exception occurred for more than 6 `int` parameters. This difference is caused by the number of generated values used for each parameter (9 for `int`, and only 2 for `String` – see Section IV.C). So, the user should limit the range of the generated values wherever he or she is able (e.g., based on the documentation of the component). Although these results seem to negatively affect the usability of our approach, it should be noted that methods with more than 6 parameters are quite rare. Moreover, it is possible not to use all existing combinations of parameters, but use the common t-way approach (discussed for example in [20]) instead.

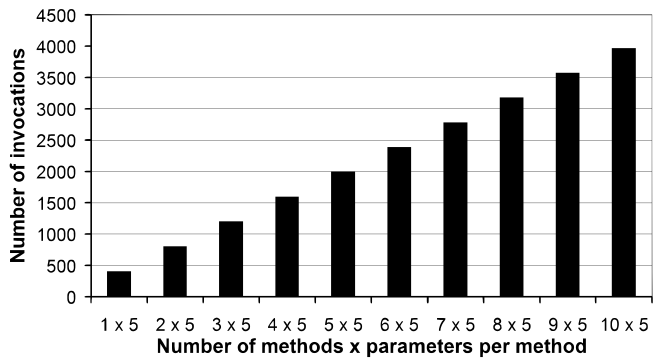
Second, the dependency of the number of generated invocations on the number of methods with 5 parameters within one component was investigated. The parameters were `String`, `Object`, `Object`, `int`, and `double`. The number of parameters was chosen as a higher than average number in common applications. Similarly, the parameter types were chosen to represent common methods. Again, the `Utilities` component was used for testing and the tests were performed in the same way. The results are depicted in Fig. 4b. As expected, the dependency on the number of methods is linear (note the linear scale on the y-axis). This means that the number of methods per component does not significantly affect the usability of our approach.

Third, the dependency of the number of generated invocations on the number of components each with 10 methods (with total of 26 parameters of various types) was investigated. All the components of the test application (see Section V.A) were used for this purpose. The results are depicted in Fig. 4c. Again, the dependency on the number of components is linear and thus the number of components does not significantly affect the usability of our approach.

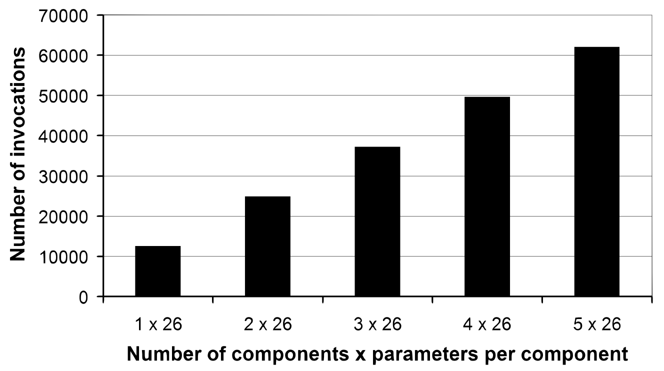
The absolute number of generated invocations is highly dependent on the tested application. The purpose of the described set of tests was merely to investigate the dependency of the generated invocations count on the number of method parameters, the number of methods, and the number of components. It was shown that the major problem is the high number of method parameters. This can be mitigated by a more advanced generation of parameter values and the usage of the t-way approach, which is a part of our future work.



a) Number of parameters of a single method



b) Number of methods with 5 parameters



c) Number of components with 10 methods

Fig. 4 Dependencies of the number of generated invocations

C. Functioning of the Entire Approach

In order to demonstrate the functioning of the entire interface-based component testing on an example, the second set of tests was performed using the same test application and the CoCoME application. The purpose of the tests was to demonstrate the ability of the approach to uncover changes in the component behavior.

First, the testing method was performed on the test application. The numbers of methods and the total numbers of parameters per component are summarized in Table I. The user did not provide any initial invocations nor placed any restrictions on the invocation generation. The result of the method – the filled tree data structure – was stored to a file. Then, three changes were separately performed to the `Calculator` component. For each change, the testing was performed again. The resulting filled tree data structure was then compared to the tree data structure created earlier for the original component. The differences are summarized in Table II. The first change (#1) was one added method (with 4 parameters) to the `Calculator` component. Second change (#2) was that one method of this component ceased to throw an exception when invoked with the `null` value. The third change (#3) was that one method of this component started to return `null` instead of an instance.

The first change caused only one difference on the methods level. Of course, the filled tree data structure with the added method incorporates its invocations (and their consequences) as well (see column #1 in Table II). However, this is not counted as a difference, since the comparison does not explore lower levels of branches of the filled tree data structures when a difference is discovered on the higher levels. The second change caused numerous differences on the invocations and consequences levels (see column #2 in Table II), because the changed behavior of the method (i.e., not throwing an exception when invoked with the `null` value) influenced other methods as well. The third change caused numerous differences on the consequences level only (see column #3 in Table II).

Second, the testing method was performed on the CoCoME application. The numbers of methods and parameters of the utilized components of the CoCoME application are summarized in Table III. Again, the user did not provide any initial invocations nor placed any restrictions on the invocation generation. The testing was performed the same way as with the test application (see above). Three changes were separately performed to the `Data` component. The differences are summarized in Table IV. The first change (#1) was one added method (with 1 parameter) to the `Data` component. Second change (#2) was that one method of this component ceased to throw an exception when invoked with the `null` value. The third change (#3) was that one method of this component started to return `null` instead of an instance.

The first change caused only one difference on the methods level. The second change caused several differences on the consequences levels (see column #2 in Table IV), but not on the invocations level like the similar change in the test application (see Table II). The reason is that in the CoCoME application, the second change did not affect other methods. The third change caused only one difference on the consequences level (see column #3 in Table IV).

It should be also noted that there are no subsequent invocations in Table IV. This does not mean that the components do not utilize services of the other components. The reason is that, in the CoCoME application (unlike the test application), the majority of the inter-component interactions are performed using OSGi Events, which are currently not intercepted by the interface-based component testing implementation. Despite this setback, the approach did uncover all introduced differences.

TABLE I. NUMBER OF METHODS AND PARAMETERS OF THE COMPONENTS OF THE TEST APLICATION

Component	Number of methods	Total number of parameters
`Geometry`	3	4
`Calculator`	16	21
`Logger`	5	3
`Text`	7	10
`Utilities`	4	8

TABLE II. DIFFERENCES OF THE FILLED TREE DATA STRUCTURES OF THE TEST APPLICATION

Structure	Original	#1	#2	#3
Explorations	3	3	3	3
Generated invocations	895	917	895	895
Subsequent invocations	747	769	569	725
Exceptions	10	10	9	10
Return values	910	933	890	910
Parameters changes	0	0	0	0
Differences (methods)	N/A	1	0	0
Differences (invocations)	N/A	0	21	0
Differences (consequences)	N/A	0	202	22

TABLE III. NUMBERS OF METHODS AND PARAMETERS OF THE COMPONENTS OF THE COCOME APPLICATION

Component	Number of methods	Total number of parameters
`Coordinator`	1	1
`Data`	16	25
`Dispatcher`	2	6
`Reporting`	3	3
`Store`	12	8
`Bank`	2	3
`CardReaderController`	3	3
`CashBoxController`	7	7
`CashDeskApplication`	10	10
`CashDeskGUIController`	10	10
`LightDisplayController`	2	2
`ScannerController`	1	1

TABLE IV. DIFFERENCES OF THE FILLED TREE DATA STRUCTURES OF THE COCOME APPLICATION

Structure	Original	#1	#2	#3
Explorations	2	2	2	2
Generated invocations	297	299	297	297
Subsequent invocations	0	0	0	0
Exceptions	224	224	213	224
Return values	1	3	12	1
Parameters changes	0	0	0	0
Differences (methods)	N/A	1	0	0
Differences (invocations)	N/A	0	0	0
Differences (consequences)	N/A	0	11	1

The second set of tests successfully demonstrated that the interface-based testing was able to uncover all three introduced differences in both component-based applications. The thorough testing of the method including a significantly higher number of components and third party applications is a part of our future work.

VI. CONCLUSION

In this paper, we described an approach to component testing automation, with scenario generation and augmentation based on a static interface analysis and runtime logging. The approach can be used for detecting the differences in the behavior of various versions of a software component inside a given component-based application.

The feasibility and effectiveness of the approach, as well as its limitations, were demonstrated using two sets of tests, which were performed using a prototype test generation implementation. Although the extent of generated data (parameter value combinations) grows prohibitively fast in the fairly rare case of methods with many parameters, the number of tested components does not significantly affect the usability of our approach.

For future work, enhancing the formal models, on which the approach is based, could improve coverage while reducing test set size. Further, considering the effects of threading together with exploring the possibility of recording all occurrences of the invocation consequences, not only the first one, should improve test quality through better information about the behavior of the components. We will also explore the behavior of our method when there are two or more new versions of components in the tested application and focus on the situations when a subset of highly dependent components is changed for new versions in the component application.

The priority of our future research is however the improved generation of parameter values for method invocations. We are working on creating an automatic generator that will provide the complex testing data, such us objects or object collections with all attributes set to values fulfilling expected criteria. This requires a method for describing the limits for the object attributes and also a tool that will be able to analyze object structure, including references to other objects and create automatically the necessary testing data[1]. Furthermore, along with attribute description, analysis of program control flow can be used in order to create tests that will provide sufficient coverage of tested application.

REFERENCES

[1] C. Szyperski, D. Gruntz, and S. Murer, Component Software – Beyond Object-Oriented Programming, ACM Press, New York, 2000.

[2] The OSGi Alliance, OSGi Service Platform Core Specification, release 4, version 4.2, 2009.

[3] J. McAffer, P. VanderLei, and S. Archer, OSGi and Equinox: Creating Highly Modular JavaTM Systems, Pearson Education Inc., 2010.

[4] D. Rubio, Pro Spring Dynamic Modules for OSGiTM Service Platform, Apress, USA, 2009.

[5] G. J. Myers, T. Badgett, and C. Sandler, The Art o Software Testing, Third Edition, John Wiley and Sons, Inc., Hoboken, 2012.

[6] P. G. Sapna and H. Mohanty, "Automated Scenario Generation based on UML Activity Diagrams," International Conference on

[1] Current implementation is at https://github.com/mrfranta/jop

Information Technology, 2008, December 2008, pp. 209–214, http://dx.doi.org/10.1109/ICIT.2008.52

[7] S. J. Cunning and J. W. Rozenbiit, "Test Scenario Generation from a Structured Requirements Specification," IEEE Conference and Workshop on Engineering of Computer-Based Systems, 1999, Proceedings, March 1999, pp. 166–172, http://dx.doi.org/10.1109/ECBS.1999.755876

[8] X. Hou, Y. Wang, H. Zheng, and G. Tang, "Integration Testing System Scenarios Generation Based on UML," 2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering, August 2010, pp. 271–273, http://dx.doi.org/10.1109/CMCE.2010.5610488

[9] V. A. De Santiago Jr. and N. L. Vijaykumar, "Generating model-based test cases from natural language requirements for space application software," Software Quality Journal, vol. 20(1), 2012, pp. 77–143, http://dx.doi.org/10.1007/s11219-011-9155-6

[10] S. S. Somé and X. Cheng, "An Approach for Supporting System-level Test Scenarios Generation from Textual Use Cases," Proceedings of the 2008 ACM symposium on Applied computing, Fortaleza, 2008, pp. 724–729, http://dx.doi.org/10.1145/1363686.1363857

[11] V. Simko, D. Hauzar, T. Bures, P. Hnetynka, and F. Plasil, "Verifying Temporal Properties of Use-Cases in Natural Language," LNCS, Vol. 7253, 2011, pp. 350–367, http://dx.doi.org/10.1007/978-3-642-35743-5_21

[12] A. Cockburn, Writing Effective Use Cases. Addison-Wesley, 2000.

[13] T. Potuzak and R. Lipka, "Possibilities of Semi-automated Generation of Scenarios for Simulation Testing of Software Components," International Journal of Information and Computer Science, vol. 2(6), September 2013, pp. 95–105.

[14] B. Korel, "Black-Box Understanding of COTS Components," Seventh International Workshop on Program Comprehension, Pittsburgh, 1999, pp. 92–99, http://dx.doi.org/10.1109/WPC.1999.777748

[15] S. Liu and W. Shen, "A Formal Approach to Testing Programs in Practice," 2012 International Conference on Systems and Informatics, Yantai, 2012, pp. 2509–2515, http://dx.doi.org/10.1109/ICSAI.2012.6223564

[16] J. M. Haddox, G. M. Kapfhammer, and C. C. Michael, "An Approach for Understanding and Testing Third Party Software Components," Proceedings of Annual Reliability and Maintainability Symposium, Seattle, 2002, pp. 293–299, http://dx.doi.org/10.1109/RAMS.2002.981657

[17] K. Jezek, L. Holy, A. Slezacek, and P. Brada, "Software Components Compatibility Verification Based on Static Byte-Code Analysis," 39th Euromicro Conference Series on Software Engineering and Advanced Applications, Santander, September 2013, pp. 145-152, http://dx.doi.org/10.1109/SEAA.2013.58

[18] T. Potuzak and R. Lipka, "Interface-based Semi-automated Generation of Scenarios for Simulation Testing of Software Components," SIMUL 2014 - The Sixth International Conference on Advances in System Simulation, Nice, October 2014, pp. 35-42.

[19] S. Herold, H. Klus, Y. Welsch, C. Deiters, R. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, C. Pfaller, "CoCoME - The Common Component Modeling Example," The Common Component Modeling Example, LNCS, Vol. 5153, 2008, pp. 16–53.

[20] B. S. Ahmed, K. Z. Zamli, "A variable strength interaction test suites generation strategy using Particle Swarm Optimization," The Journal of Systems and Software, Vol. 84, 2011, pp. 2171–2185, http://dx.doi.org/10.1016/j.jss.2011.06.004