# Semi-automated Generation of Simulated Software Components for Simulation Testing

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

Richard Lipka
Department of Computer Science and Engineering/
NTIS – European Center of Excellence
Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 306 14, Plzen, Czech Republic
e-mail: lipka@kiv.zcu.cz

***Abstract*—Using a component-based software development, applications can be constructed from individual reusable software components providing particular functionalities. The testing of a particular component is very important not only individually, but in the context of its neighboring components as well. In the SimCo simulation tool, which we developed, it is possible to perform tests of the software components in a simulation environment. For this purpose, the neighboring components of the tested components can be replaced by their simulation models. In this paper, we describe an approach for the semi-automated generation of the simulation model of a software component based on the analysis of its interface and on the observation of its behavior among its neighboring components.**

***Keywords-software component; simulation testing; simulation model generation.***

## I. INTRODUCTION

Component-based software development is a spreading trend in contemporary software engineering. The main idea of this approach lies in the utilization of isolated reusable parts of software, which provide functionality via services. These parts – called *software components* – can then cooperate and form an entire application. Each component has a defined interface, which is a set of services the component provides. The component may also require services of other components in order to work. A single component can be used in multiple applications and particular components in a single application can be provided by different developers [1]. This reinforces the need for the testing of software components and their interactions.

The testing of software components should be ensured by their developers. However, it is also useful to test the functionality, quality of services, and extra-functional properties of the component while it is interacting with other components forming together an application. Such kind of testing has to be performed by programmers who did not create the components themselves, but use them to make up a new application. So, the source code, descriptions, and other resources may be unavailable to them. In an extreme case, only the public interfaces of the components may be known.

During our research, we developed the SimCo simulation tool, which enables simulation testing of particular components, sets of components, or entire applications directly – without creating additional models of the tested components [2][3][4]. When only a part of a component-based application (i.e. a single component or a set of components) is tested, it is necessary to satisfy all dependencies of the tested components. More specifically, the tested components can require services of other components, which are not considered necessary for the testing and thus are not present in the simulation environment. In the SimCo simulation tool, these required components are replaced by their simulation models in order to satisfy the dependencies of the tested components. They have the same interfaces as the components they are replacing, but the original functionality can be substituted for example by lists of pre-calculated values or random numbers generators [5].

So far, the behavior of the simulation models of components for the SimCo simulation tool is created manually, which is a lengthy and error-prone process. In this paper, we describe an approach for semi-automated generation of the simulation models of components, whose implementation is at our disposal, but the source code is not. The approach is based on the analysis of the public interface of the component and on the observation of the behavior of the component among its neighboring components. The result of the approach is a generated skeleton of the simulation model of the component with partial functionality and clues, which can be used by a programmer for finishing of the functionality of the simulation model of the component.

The remainder of this paper is structured as follows. Software components are described in Section II. In Section III, the simulation testing is discussed. Section IV describes the SimCo simulation tool. In Section V, the related work is discussed. The generation of simulated components demonstrated on a case study is described in Section VI. The future work is described in Section VII and the paper is concluded in Section VIII.

## II. SOFTWARE COMPONENTS

Before we proceed with the description of our approach, we will briefly discuss the basics of the component-based software development.

 ISBN: 978-1-61208-371-1 

### A. Basic Notions

Using the component-based software development, the applications are constructed from software components. A software component is a black box entity with a well defined public interface, but no observable inner state. So, the components mutually interact solely using their interfaces. They should be reusable and the author of an application can be different from the author(s) of the particular components. This general definition is common for most component models [1].

A component model describes how the particular software components look, behave, and interact. A component framework is then a specific implementation of a component model and there can be several different implementations of each component model [1]. The existing component models are generally incompatible, since they use different approaches to solving particular issues of the software components.

### B. OSGi

An example of a widespread component model is the OSGi (Open Service Gateway initiative), which is contemporary used in both industrial and academic spheres. There are several OSGi frameworks (i.e. implementations of the OSGi model), which are commonly used [6]. The OSGi component model is designed for Java programming language. It is dynamic in nature, which means that it enables to install, start, stop, and uninstall particular software components without the need to restart the OSGi framework itself [7].

A software component in the OSGi component model is referred as a *bundle* and has a form of a standard Java `.jar` file with additional component-model-related information (e.g., lists of services provided by the bundle and list of services required by the bundle). Each bundle can contain any number of classes. So, it can provide arbitrary complex functionality [6]. As the interface of the bundle, standard Java interfaces are used. So, the particular services provided by a bundle have the form of standard Java methods [6].

Since the OSGi component model is widespread and the transformation of one component model into another is difficult (see Section II.A), the SimCo simulation tool was developed solely for the OSGi [2]. However, the main ideas and approaches utilized in it can be used in other component models as well.

## III. Simulation Testing

Now, as we described the basics of the component-based software development, we will briefly discuss the simulation testing.

### A. Discrete-Event Simulation

A discrete-event simulation is a widely used simulation technique. The simulation run is subdivided into a sequence of time-stamped events representing incremental changes of the simulation state. The simulation time between two succeeding events can be arbitrary long. Nevertheless, the simulation does not perform a real waiting for the specified time. Instead, the simulation time is set to the time stamp of the event, which is processed. So, the simulation time "jumps" from the time stamp of an event to the time stamp of the next one [8].

The events are handled by a calendar, which incorporates the list of events ordered by their time stamps. At the simulation run start, the calendar removes the first event from the event list, sets the current simulation time on the time stamp of the event, and performs the action (i.e. a change to the simulation state) associated with the event. This action can (but does not have to) create one or more new events, which are added to the event list of the calendar on the positions corresponding to their time stamps. Then, the next event is removed from the event list and the entire process repeats until a stop condition is reached or the event list is empty [8].

The discrete-event simulation, which was briefly described in previous paragraphs, is used in the SimCo simulation tool [2].

### B. Simulation Testing of Software Components

Although the simulation testing can be used for various systems, for the needs of this paper, we will focus on the simulation testing of software components. There are two approaches that can be used.

The first approach is to create a simulation model of the software component, which shall be tested. This model is then used in the simulation [9]. The advantage of this approach is that the simulation model of the software component is suited for the simulation. The disadvantage is the necessary creation of the model, which can be often a lengthy and error-prone manual process. Moreover, the model usually does not incorporate all aspects of the original software component, but only aspects, which are considered important for the simulation testing by the creator of the model. This can lead to an unintentional omission of features, which in fact can be important for testing. These disadvantages can be avoided or at least diminished by a full or a partial automation of the process of the creation of the component simulation model.

The second approach is to use the software component in the simulation directly. The advantage of this approach is that there is no need for creation of the simulation model of the tested component and the component is tested directly. So, its responses to the stimuli induced during the testing are genuine. The main disadvantage is that the component is not suited to run in a simulation environment. Hence, the simulation environment must be able to allow running of the software component as it would run in a real application. It should also be noted that the problems connected to the creation of simulation models (see previous paragraph) are not fully avoided by the direct testing of the software component in a simulation environment. The simulation model of the tested component is not created, but it is often necessary to create simulation models of other components required by the tested component. Similarly to the first approach, the disadvantages related to the simulation models creation can be diminished by automation of the creation process.

The simulation testing of the software components can be further divided according to the knowledge of the tested components. If their source code is known, it can be used for

 ISBN: 978-1-61208-371-1 

preparation of the tests. This is so-called *white box* testing. If the source code in not known, the tests must be prepared using interfaces or other available specifications of the tested components. This is so-called *black box* testing [10].

## IV. SIMCO SIMULATION TOOL

The SimCo simulation tool, which we developed, enables to test the software components directly in the simulation environment. Its main features are described in the following sections.

### A. SimCo Features

The SimCo is written using the Java programming language. It is based on the OSGi model [6] and, currently, it is running in the Equinox OSGi framework [11]. Hence, the software components, which can be tested within the SimCo, are limited to OSGi bundles. This restriction is not so hard, since the OSGi is quite widespread in various industry areas as well as in the academic sphere (see Section II.B).

Within the SimCo simulation tool, it is possible to test a single component (i.e. an OSGi bundle), set of components, or an entire component-based application. The SimCo itself is constructed from the components (OSGi bundles) as well in order to enable its simple modifications and extensions [5].

For the simulation testing, the SimCo utilizes the discrete-event simulation (see Section III.A) where the events correspond to the invocations of particular services of the tested components [4]. The initial events, which shall be performed during the simulation run, are imported to the event list from a testing scenario. This testing scenario is loaded from an XML file prior to the execution of the simulation testing [5].

### B. SimCo Software Component Types

There are four types of the software components in the SimCo simulation tool – the *core* components, the *real tested* components, the *simulated* components, and the *intermediate* components [5].

The core components provide the functionality of the SimCo itself. These software components ensure the running of the simulation. Furthermore, they provide all necessary supplementary functions such as logging, measuring, and storing of the observed parameters, visualization of the simulation, and so on. The most important core component is the `Calendar`, which interprets the events from its event list. More specifically, it invokes corresponding services of the software components and so ensures the advancement of the simulation in time [5].

The real tested components are the components which are tested in the SimCo simulation tool. These components can be created by various manufacturers. So, the source code of the components may not be available to us, which implies the black box testing (see Section III.B). Since the components are running in the simulation environment, there are several issues, such as discrepancy of the real and the simulation time, undesirable hidden network communication of the real tested components, and so on. These issues can be solved using various means including aspect oriented programming or changing of the bytecode of the tested components. For more information, see [4] and [5].

The simulated components are simulation models of components, which provide services required by the real tested components. They are not necessary when an entire component-based application is tested within the SimCo simulation tool. However, when only a single component or a limited set of components are tested, components providing some services required by the real tested components do not have to be present in the simulation environment. The reason is that these components are not considered important for the testing, for example because they were already tested before and/or the execution of their services is time-consuming. Nevertheless, it is necessary to satisfy the requirements of the real tested components. So, the required components, which are not present in the simulation environment, are replaced by their simulation models – the simulated components. The simulated components can be useful for the speedup of the simulation, because they do not have to perform the calculations of the original components they are mimicking. Instead, prerecorded values, table of return values, or random numbers generators can be used, depending on the situation [5]. At the same time, the simulated components must exhibit the same external behavior as the original components they are mimicking. For example, if the consequence of a service invocation on the original component is the invocation of a service on another component, this must be true for the simulated component as well. The simulated components also ensure the control of the simulation by the SimCo simulation tool. All invocations of the services of the simulated components are performed using the events and the calendar [5].

The intermediate components are used to ensure the control of the simulation by the SimCo simulation tool even between two real tested components. More specifically, an intermediate component is used as a proxy for a real tested component. It provides the same services and all service invocations on the real tested component are in fact handled by this proxy (see Figure 1). The intermediate component ensures that the service invocation on a real tested component is performed using the events and the calendar, similar to the simulated components (see previous paragraph) [5]. Moreover, it performs the invocation of the same service on the real tested component, for which it serves as a proxy. This is necessary in order to obtain the return value and/or trigger further actions (e.g., an invocation of a service on another component).

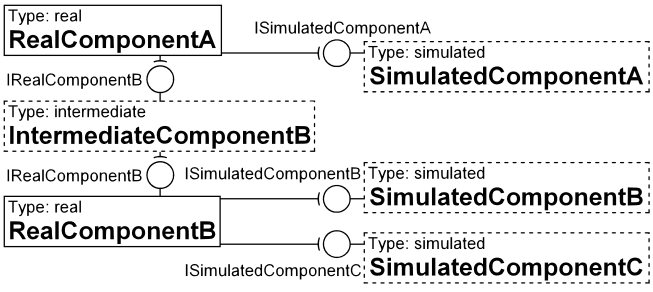


Figure 1. An example of the SimCo component types

The simulated components and the intermediate components can also be used for the placing of probes observing selected properties of the real tested components. For example, it is possible to use the intermediate component to measure execution times of the services of the real tested component, for which the intermediate component acts as its proxy [5].

An example of three component types (except the core components) of the SimCo simulation tool is depicted in Figure 1.

## V. Related Work

The generation of the simulation models of software components, whose implementation is at our disposal, but their source code is not, is not common and it is difficult to find description of similar approaches in the literature. However, it is related to the black box testing of software components, since the inspection of the component behavior is performed in both cases. The relevant techniques are mentioned in the following sections.

### A. Reverse Engineering

Reverse engineering is a process of analyzing a system (a software component in our case) and creating the representation of the system in another form or at a higher level of abstraction [12]. Reverse engineering can be used even when the source code is known (i.e. during white box testing) for example for creation of UML (Unified Modeling Language) diagrams or control-flow graphs. For this purposes, the static or dynamic analysis can be used [12]. In situations when the source code is not known (i.e. during black box testing) the reverse engineering can be used for the extraction of the source code from the binary representation of the application [13]. In our case, the application is the software component (an OSGi bundle) and the binary representation is the bytecode of its `.class` files.

There are several disadvantages of reverse engineering used for the obtaining of the source code. The names of the methods and parameters are lost, since they are irrelevant (and thus not present) in the binary representation. Similarly, the comments cannot be restored for the same reason [13]. So, it may be difficult for a human programmer to understand the obtained source code.

Although the reverse engineering of the Java bytecode is possible and, using specialized tools, quite straightforward, all mentioned issues stand. Moreover, our intention is not to replicate the exact inner functioning of the components. The simulated components should exhibit the same external behavior as their original components, but their inner functioning can be significantly different (e.g., in order to reduce the computation time). So, even with a successful reverse engineering of the bytecode, the analysis and modifications of the obtained source code by a programmer is still necessary. Due to these issues, a significant amount of manual work may be required.

### B. Interface Probing

Interface probing is a technique that utilizes the public interface of the software component for the examination of its behavior and its features. This technique can be readily utilized during the black box testing, since the source code is not necessary. Using the interface probing, the public interface with the services of the tested software components is identified first. Then, the input values for the services are generated, the particular services are invoked with the generated inputs, and the outputs of the services of the component are observed [13][14]. For this purpose, the component can be wrapped in an encasing object, which controls the input and output data flows [15]. This is similar to the intermediate components used by the SimCo simulation tool (see Section IV.B).

The main disadvantage of the interface probing is the problematic generation of the input values. They can be created randomly or manually. In both cases, it is possible to omit values, which are in fact important for the uncovering of hidden features or a hidden behavior of the tested component and its services [13]. In order to reduce the problem, it is possible to use typical values, such as 0, -1, 1, minimal value, maximal value for an integer input or all possible values for a character input [13]. However, these examples are very general and can be useless (in the sense of the uncovering of a hidden behavior) for a specific service. So, it is necessary to use other typical and border input values for the given service, if possible. These values can be provided by a programmer based on textual or other description of the software component and its services, which can be (but does not have to) at the disposal. At the very least, the programmer can use the name of the service as a clue for the service's probable functioning.

### C. Other Techniques

Other common techniques include the generation of the graphical user interface for the software component, which enables instant access to the particular services of the component. This graphical user interface then enables quick ad hoc testing of the particular services of the component using various input values without the necessity to create a client application for the component [16].

There are also more exotic approaches such as proposal for the extended component interface specification, which would enable easier black box testing [17] or the utilization of a genetic algorithm for the generation of the input data for the testing [18].

## VI. Simulated Components Generation

As it was mentioned before, so far, the simulated components of the SimCo simulation tool are created manually. If the source code of the component, for which the simulated component is created, is known, the simulated component can be created using analysis of this source code. If the source code of the original component is not known (e.g., the component was created by another manufacturer), the situation is even worse, because we have only the public interface of the component and its bytecode (and the Javadoc or other documentation in some cases) at our disposal. In this case, it is difficult to analyze the behavior of the component.

Therefore, we have developed a semi-automated approach for the generation of the simulated components. To

 ISBN: 978-1-61208-371-1 

avoid confusion, the real component, for which the simulated component (i.e. its simulation model) will be generated, will be referenced only as the *original component* for short from now on.

### A. Approach Description

The skeleton of the component (i.e. methods representing services of the component) can be generated easily using the analysis of the public interface of the original component. The basics of the behavior of the simulated component can be then generated using the analysis of the behavior of the original component while it is running among other real components.

The approach does not require the knowledge of the source code of the original component, but there are some limitations. First, in the majority of the cases, the approach is unable to determine the complete behavior of the original component, but can provide clues for this behavior. These clues can be then used by the programmer during the finishing of the simulated component. Thus the "semi-automated" attribute of the approach. Second, it is necessary to have the neighboring components of the original component available. These components require the services or are required by the services of the original component.

### B. Case Study: Traffic Crossroad Control

The approach will be demonstrated using a case study – Traffic crossroad control. It represents a component-based application for the control of road traffic in a crossroad using traffic lights. It is expected to run on a specific hardware and operate a variety of hardware sensors and control units [4]. The components involving any direct contact with sensors and control units were replaced by (manually created) simulated components in order to enable running of the application on a standard desktop computer [2].

The Traffic crossroad control application consists of eight components (see Figure 2). The `TrafficCrossroad` component provides the information about the structure of the traffic crossroad. Moreover, its simulated version incorporates a nanoscopic road traffic simulation replacing the real traffic [4]. The `OpticDetection` and

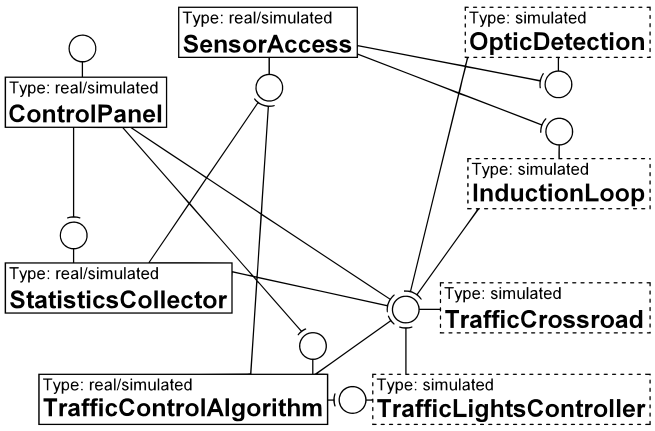


Figure 2. Software components of the Traffic crossroad control application

```
int getQueueLength(String)
boolean isVehicle(String)
DetectorType getCurrentDetectorType()
void setCurrentDetectorType(DetectorType)
```

Figure 3. List of services of the `SensorAccess` component

the `InductionLoop` components handle specific hardware sensors and provide measured data from these sensors. The simulated versions of these components utilize the data from the nanoscopic road traffic simulation of the `TrafficCrossroad` component. The `TrafficLightsController` component ensures the activation of particular traffic lights. The `ControlPanel` component is a user interface of the entire application. The `TrafficControlAlgorithm` component incorporates an algorithm for the control of the traffic lights. This algorithm can (but does not have to) require data from the sensors mediated by the `SensorAccess` component. This component is also used by the `StatisticsCollector` component, which periodically collects the data from it and provides various traffic statistics accessible via the `ControlPanel` component [4].

In the following sections, we will describe our approach for semi-automated generation of a simulated component for the `SensorAccess` component.

### C. Generation of Component's Skeleton

The skeleton of the simulated component is generated using the analysis of the public interface of the original component. For this purpose, a service of the OSGi can be used, since it provides functionality to determine the public interface (i.e. a Java interface) of a component (bundle) [19]. For the determination of the particular services (i.e. Java methods) of this interface, the Java reflection can be used [20]. This way, we obtain a list of services provided by the component including the types of their parameters and their return values. The information obtained for the `SensorAccess` component using this approach is depicted in Figure 3.

Using the obtained list of services, the skeleton of the simulated component is generated. The skeleton of each simulated component is represented as one Java class with methods along with their parameters and return values corresponding to the particular services of the original component. The bodies of the generated methods are not completely empty, the logic necessary for adding of the events corresponding to the invocation of the services to the calendar are automatically added. The standard OSGi `Activator` class ensuring the connection of the generated simulated component to the particular components it requires [19] is automatically added to the simulated component as well. The generated skeleton of the simulated component with its services is depicted in Figure 4.

### D. Generation of Component's Behavior

The more difficult part of our approach is the generation of the component behavior. For this purpose, the original

```
public class SimulatedSensorAccess implements
    ISensorAccess {
  private BundleContext bc;
  private IOpticDetection od;
  private IInductionLoop il;
  private ICalendar calendar;

  public SimulatedSensorAccess(BundleContext bc,
      IOpticDetection od, IInductionLoop il,
      ICalendar calendar) {
    this.bc = bc;
    this.od = od;
    this.il = id;
    this.calendar = calendar;
  }

  public int getQueueLength(String arg0) {
    calendar.insertEvent();
    return 0;
  }

  public boolean isVehicle(String arg0) {
    calendar.insertEvent();
    return false;
  }

  public DetectorType getCurrentDetectorType() {
    calendar.insertEvent();
    return null;
  }

  public void setCurrentDetectorType(
      DetectorType arg0) {
    calendar.insertEvent();
  }
}
```

Figure 4. Skeleton of the simulated `SensorAccess` component

component, for which the simulated component shall be created, is imported to the SimCo simulation tool along with other components, which are required by the services of the original component or require the services of the original component. It is also possible to import entire component-based application, whose part the original component is. The components, which require the services of the original component, are not necessary, but can provide information about the parameters of the services of the original component.

It is assumed that all the neighboring components of the original components are real, but they can also be simulated, assuming they exhibit the same behavior as the real components they are mimicking. Between all pairs of the neighboring real components, there are intermediate components designed to observe the invocations of the services. The behavior generation then works as follows.

A tree data structure incorporating the original component and all components, which can invoke services of the original component, and all the services of these components is initiated. Each service of each component contains a list of possible invocations with various parameters. These invocations are automatically generated based on the type of their parameters, similarly to [13].

For the `enum` parameters, all possible values (including `null`) are generated, since there are usually only several values. For the `char` parameters, the basic 256 characters and several higher values are generated (in Java, the size of char type is 2 bytes). For the number parameters, a set of representative and border values are generated (e.g., -128, -64, -16, -1, 0, 1, 16, 63, 127 can be generated for a `byte` parameter). It is not feasible to generate all possible values of the number parameters. For object parameters, only the `null` value is generated. If the service has multiple parameters, all combinations are generated. Of course, this can lead to the exponential increase of the number of generated invocations. Hence, the programmer can restrict the generated parameters to values, which he or she considers representative. Moreover, the programmer should provide values, which cannot be automatically generated, for example filled data objects, which are parameters of the invocations. The programmer can use Javadoc of the original component or other documentation for this purpose, if they are at the disposal.

Once the structure is initiated, it is explored component by component, service by service, invocation by invocation. Each invocation from the tree data structure is performed on the corresponding component and it can have various consequences – an exception, a return value, a change of the inner state of the component, and a subsequent invocation of a service of another component. These consequences, with the exception of the change of inner state, which is unobservable from outside, are captured by the intermediate components (and also by the simulated components if they are present) and inserted to the tree data structure, but only if they are not already present in the structure and they are related to the components, which are present in the structure. Simultaneously, the flag indicating that a change of the structure occurred is set. If the invocation consequence is a subsequent invocation, and this invocation is not already present in the structure, the invocation is added to the structure as well and the flag is again set. The exploration of the structure repeats while this flag is set. The pseudocode of the filling and exploring of the tree structure is depicted in Figure 5.

The iterative nature of the approach enables to explore multiple inner states of the components. As it was said before, the change of the inner state of a component is unobservable. However, the inner state of the component can be changed by invocation of its services (if the component can have different inner states at all). On the other hand, the inner state of the component can influence the behavior of its services – a services invoked with the same parameters can have different outcome due to a different inner state of the component. So, the repeating of the invocations, which were already performed, can bring new invocations and invocation consequences, which can be useful for a better exploration of the original component behavior. When no new invocation consequences are generated in the iteration, the process is stopped in order to avoid to be stuck in an infinite loop.

Consider now the Traffic crossroad control case study and the generation of the simulated component for the `SensorAccess` component. This component is then the original component and the `StatisticsCollector` and the `TrafficControlAlgorithm` components are the

 ISBN: 978-1-61208-371-1 

```
//Initialization of the tree data structure
structure.addComponent(originalComponent);
structure.addComponents(
  simco.getInvokingComponents(originalComponent));

for (c: structure.components) {
  c.services = simco.getServicesOfComponent(c);
  for (s: c.services) {
    //Generate parameters and read parameters from
    //the user for the invocation
    s.invocations =
      simco.generateAndReadInvocations();
  }
}

//Exploration of invocations and consequences
structure.setChanged(true);
while (structure.isChanged()) {
  structure.setChanged(false);
  for (c: structure.components) {
    for (s: c.services) {
      for (i: s.invocations) {
        invocationConsequences =
          simco.performServiceInvocation(c, s, i);
        for (ic: invocationConsequences) {
          //Only consequences, which are not
          //already present
          if (!i.consequences.contains(ic)) {
            i.consequences.add(ic);
            structure.setChanged(true);
          }
          if (ic.type == SUBSEQUENT_INVOCATION) {
            sc = ic.subsequentComponent;
            ss = ic.subsequentService;
            si = ic.subsequentInvocation;

            //Only for components contained in
            //the structure and only invocations,
            //which are not already present
            if (structure.contains(sc) &&
                !ss.contains(si)) {
              structure.addInvocation(sc, ss, si);
              structure.setChanged(true);
            }
          }
        }
      }
    }
  }
}
```

Figure 5. Pseudocode of the tree data structure exploration

invoking components (see Figure 2). The user did not provide any parameters of the invocations. The filled explored tree data structure is then depicted in Figure 6. There, the components, services, invocations, and invocation consequences are marked with C, S, I, and IC, respectively. The value in the parentheses of I expresses the originator of the invocation – G for "generated at beginning", U for "provided by user", and A*x* for "added automatically during the *x*th iteration". The value in the parentheses of IC denotes the index of the iteration, in which the invocation consequence was added to the tree data structure (starting with 0).

As can be seen in Figure 6, the invoking components add invocations of the services of the original component and provide `String` parameters, which would not be generated

```
C: StatisticsCollector //Invoking component #1
 S: Statistics collectStatistics()
  I(G): collectStatistics()
   IC(0): subsequent invocation
    [SensorAccess.getQueueLength("E_01")]
. . .
   IC(0): subsequent invocation
    [SensorAccess.getQueueLength("E_04")]
   IC(0): return value [statistics]
   IC(1): return value [statistics]

C: TrafficControlAlgorithm //Invoking component #2
 S: void updateTrafficLightsState()
  I(G): updateTrafficLightsState()
    IC(0): subsequent invocation
     [SensorAcces.isVehicle("E_01")]
. . .
    IC(0): subsequent invocation
     [SensorAcces.isVehicle("E_04")]

C: SensorAccess //Original component
 S: int getQueueLength(String)
  I(G): getQueueLength(null)
   IC(0): exception [NullPointerException]
  I(A0): getQueueLength("E_01")
   IC(0): subsequent invocation
    [OpticDetection.getVehiclesCount("E_01")]
   IC(0): return value [2]
   IC(1): subsequent invocation
    [InductionLoop.isVehicle("E_01")]
   IC(1): return value [1]
. . .
  I(A0): getQueueLength("E_04")
   IC(0): subsequent invocation
     [OpticDetection.getVehiclesCount("E_04")]
   IC(0): return value [0]
   IC(1): subsequent invocation
     [InductionLoop.isVehicle("E_04")]
   IC(1): return value [0]
 S: boolean isVehicle(String)
  I(G): isVehicle(null)
   IC(0): exception [NullPointerException]
  I(A0): isVehicle("E_01")
   IC(0): subseqeuent invocation
    [OpticDetection.getVehiclesCount("E_01")]
   IC(0): return value [true]
   IC(1): subseqeuent invocation
    [InductionLoop.isVehicle("E_01")]
   IC(1): return value [true]
. . .
  I(A0): isVehicle("E_04")
   IC(0): subseqeuent invocation
    [OpticDetection.getVehiclesCount("E_04")]
   IC(0): return value [false]
   IC(1): subseqeuent invocation
    [InductionLoop.isVehicle("E_04")]
   IC(1): return value [false]
 S: DetectorTypes getDetectorType()
  I(G): getDetectorType()
   IC(0): return value [DetectorTypes.OPTIC]
   IC(1): return value [DetectorTypes.INDUCTION]
 S: void setDetectorType(DetectorTypes)
  I(G): setDetectorType(null)
   IC(0): exception [NullPointerException]
  I(G): setDetectorType(DetectorTypes.OPTIC)
   IC(0): nothing observable
  I(G): setDetectorType(DetectorTypes.INDUCTION)
   IC(0): nothing observable
```

Figure 6. Example of the filled explored tree data structure

if the invoking components were not present. These parameters (e.g., "E_01") are the IDs of the traffic lanes, in which the sensors survey the state of road traffic. In this case, the IDs are taken from the `TrafficCrossroad` component, which provides information about the traffic crossroad, by the `StatisticsCollector` and `TrafficControlAlgorithm` components. The invocations of the services of the `TrafficCrossroad` component are not present in the filled tree data structure, because the `TrafficCrossroad` is not present in the tree data structure, since it is not an invoking component of the original (i.e. `SensorAccess`) component. We know the origin of the IDs, only because we developed the Traffic crossroad control application. If this were not the case and the source code of the application were not available to us, the origin of the IDs would be hidden from us. Nevertheless, the IDs would still be there helping to probe the behavior of the original `SensorAccess` component.

The overall statistics of the content of the filled tree data structure is summarized in Table I. The tree data structure is filled in three iterations of the outermost (`while`) cycle (see Figure 5). In the third iteration, there are no newly added invocations or invocation consequences. So, the algorithm ends.

TABLE I. OVERALL STATISTICS OF THE FILLED TREE DATA STRUCTURE

| Feature | Count |
|---|---|
| Number of invocations generated at beginning | 8 |
| Number of invocations provided by user | 0 |
| Number of automatically added invocations in 1st iteration (A0) | 8 |
| Number of automatically added invocations in 2nd iteration (A1) | 0 |
| Number of automatically added invocations in 3rd iteration (A2) | 0 |
| Number of generated consequences in 1st iteration (0) | 31 |
| Number of generated consequences in 2nd iteration (1) | 17 |
| Number of generated consequences in 3rd iteration (2) | 0 |
| Number of iterations | 3 |

Using the part of the filled explored tree data structure containing data for the `SensorAccess` component and a set of rules, it is possible to generate limited bodies of the services of this component with clues for the programmer. The rules are following:

1. If there is a single return value for each invocation, generate an `if-else` construction depending on the parameters of the invocation for the particular return values of the service.
2. If there are multiple return values for a single invocation, generate an `if-else` construction depending on the parameters of the invocation for the return value of the service. All but the first return value for the same invocation put in the comments in the corresponding branch of the `if-else` construction.
3. If there is a single return value for a service without parameters, generate `return` statement with this value.
4. If there are multiple (different) return values for a service without parameters, this service can reflect state of the component (e.g., it is a getter). Generate an information comment for the programmer. Generate multiple `return` statements with all return values. All but the first return statements put in the comments.
5. If there is a single subsequent invocation for each invocation and the parameters of the invocation correspond by type and value to the parameters of the subsequent invocation, generate a method call corresponding to the subsequent invocation using the formal parameters of the service.
6. If there are multiple subsequent invocations of different services for a single invocation and the parameters of the invocation correspond by type and value to the parameters of the subsequent invocations, generate multiple methods calls corresponding to the particular subsequent invocations using the formal parameters of the service.
7. If there is a single subsequent invocation for each invocation and the parameters of the invocation do not correspond by type or value to the parameters of the subsequent invocation, generate an `if-else` construction depending on the parameters of the invocation for the particular methods calls corresponding to the subsequent invocations.
8. If there are multiple subsequent invocations of different services for a single invocation and the parameters of the invocation do not correspond by type or value to the parameters of the subsequent invocations, generate an `if-else` construction depending on the parameters of the invocation for the particular sets of the methods calls corresponding to the subsequent invocations.
9. If there are multiple (different) subsequent invocations for a single invocation in multiple iterations and the parameters of the invocation correspond by type and value to the parameters of the subsequent invocations, the subsequent invocation can depend on the inner state of the component. Generate an information comment for the programmer. Generate an `if-else` construction depending on the parameters of the invocation for the methods calls corresponding to the particular subsequent invocations. All but the first method call for the same invocation put in the comments in the corresponding branch of the `if-else` construction.
10. If there are multiple (different) subsequent invocations for a single invocation in multiple iterations and the parameters of the invocation do not correspond by type or value to the parameters of the subsequent invocations, the subsequent invocation can depend on the inner state of the component. Generate an information comment for the programmer. Generate multiple methods calls corresponding to the particular subsequent invocations. All but the first method call put in the comments.
11. If there is a single exception for a single invocation, generate an `if-else` construction depending on the parameters of the invocation for the throwing of the corresponding exceptions.

 ISBN: 978-1-61208-371-1 

12. If there are multiple (different) exceptions for a single invocation, generate `if-else` construction depending on the parameters of the invocation for the throwing of the corresponding exception. Additional exceptions for the same invocation put in comments in the corresponding branch of the `if-else` construction.
13. If there are multiple exceptions or a single exception for a service without parameters, generate the throwing of the corresponding exceptions. Put them into the comments.

The resulting generated bodies of the services of the simulated `SensorAccess` component are depicted in Figure 7. The handling of the service using the calendar was added already during the component skeleton generation (see Section VI.C). The particular parts of the generated bodies – invocation of services, return values, throwing of exceptions – are not directly usable. It is possible to compile the generated simulated component, but its behavior will not correspond to the behavior of the original component. However, the programmer can use the generated parts as clues for what the particular services should do. It is much easier for him or her to finish the functionality of the simulated component than when he or she should program the functionality of the simulated components from the scratch. This is the most important contribution of our semi-automated approach.

In Figure 7, it is indicated with the comments in the generated source code, which parts of the services bodies were generated using which rule. For the entire `SensorAccess` component, only the rules 1, 2, 4, 9, and 11 were utilized.

### *E. Approach Evaluation*

As it was demonstrated using the case study, the automatic part of our approach for the generation of the simulated components is working and provides clues of what the particular services of the component should do for the programmer. The programmer then finishes the functionality of the simulated component using these clues.

However, it should be noted that in a more complicated case with services with multiple parameters, the generated bodies of the services can become quite long and difficultly understandable by the programmer. In such cases, the restrictions of the generated parameters provided by the programmer are important (see Section VI.D). This restriction was not used in the presented case study, since the number of parameters of the services was low and the generated bodies of the services are easily readable.

The programmer also did not provide any parameters values for the services invocations. Despite of this, the invoking software components provided the `String` parameters, which enable a better exploration of the behavior of the services of the original component (see Section VI.D). Nevertheless, this depends solely on the nature of the component-based application. In many cases, the invoking components do not have to provide useful parameters. Then, the possibility to add the parameters of the services invocations by the programmer becomes more important.

```
public int getQueueLength(String arg0) {
  calendar.insertEvent();

  //Rule 11
  if (arg0 == null) {
    throw new NullPointerException();
  }

  //Rule 9
  if (arg0.equals("E_01"))
    od.getVehiclesCount("E_01");
    //il.isVehicle("E_01");
. . .
  else if (arg0.equals("E_04"))
    od.getVehiclesCount("E_04");
    //il.isVehicle("E_04");

  //Rule 2
  if (arg0.equals("E_01"))
    return 2;
    //return 1;
. . .
  else if (arg0.equals("E_04"))
    return 0;
  else
    return 0;
}

public boolean isVehicle(String arg0) {
  calendar.insertEvent();

  //Rule 11
  if (arg0 == null) {
    throw new NullPointerException();
  }

  //Rule 9
  if (arg0.equals("E_01"))
    od.getVehiclesCount("E_01");
    //il.isVehicle("E_01");
. . .
  else if (arg0.equals("E_04"))
    od.getVehiclesCount("E_04");
    //il.isVehicle("E_04");

  //Rule 1
  if (arg0.equals("E_01"))
    return true;
. . .
  else if (arg0.equals("E_04"))
    return false;
  else
    return false;
}

public DetectorType getCurrentDetectorType() {
  calendar.insertEvent();

  //Rule 4
  return DetectorTypes.OPTIC;
  //return DetectorTypes.INDUCTION;
}

public void setCurrentDetectorType(DetectorType
arg0) {
  calendar.insertEvent();

  //Rule 11
  if (arg0 == null)
    throw new NullPointerException();
}
```

Figure 7. Generated services bodies of the SensorAccess component

## VII. FUTURE WORK

In our future work, we will focus on improving our approach in several ways.

### A. Better Exploration of the Tree Data Structure

First of all, we will focus on a better exploration of the tree data structure, from which the simulated components are generated. This includes ordering of the particular invocations in order to explore more different inner states of the components. This can theoretically lead to acquiring of a higher number of subsequent invocations.

We also want to employ analysis of the names of the services/methods. This can be useful for example for determining whether a pair of services serves as a getter and a setter of a property of the component.

### B. Expansion of the Set of Rules

The rules described in Section VI.D can be also expanded by other rules in order to provide more and/or better clues in the generated bodies of the services of the simulated component. The generation of the clues can be also refined to be clearer for the programmer. For example, an array or a table with values accessed using indices can be used instead of large `if-else` constructions.

## VIII. CONCLUSION

In this paper, we described an approach for the semi-automated generation of a simulated component, which is used as a replacement of a real component during simulation testing of other real software components. The approach is based on the observation of the original component behavior while it is interacting with other real components, which require or are required by its services.

The resulting generated simulated component incorporates all services of the original component and parts of their bodies, which provide clues for the programmer. The programmer can then use these clues for the finishing of the behavior of the services of the simulated component. The approach was designed for the OSGi component model, but the ideas behind it can be utilized also in other component models. The approach is demonstrated using a case study.

## ACKNOWLEDGMENT

This work was supported by the European Regional Development Fund (ERDF), project "NTIS – New Technologies for the Information Society", European Centre of Excellence, CZ.1.05/1.1.00/02.0090.

## REFERENCES

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

[2] 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.

[3] T. Potuzak, R. Lipka, J. Snajberk, P. Brada, and P. Herout, "Design of a Component-based Simulation Framework for Component Testing using SpringDM," ECBS-EERC 2011 – 2011 Second Eastern European Regional Conference on the Engineering on Computer Based Systems, Bratislava, September 2011, pp. 167–168.

[4] T. Potuzak, R. Lipka, P. Brada, and P. Herout, "Testing a Component-based Application for Road Traffic Crossroad Control using the SimCo Simulation Framework," 38th Euromicro Conference on Software Engineering and Advanced Applications, Cesme, Izmir, September 2012, pp. 175–182.

[5] R. Lipka, T. Potuzak, P. Brada, and P. Herout, "Verification of SimCo – Simulation Tool for Testing of Component-based Application," EUROCON 2013, Zagreb, July 2013, pp. 467–474.

[6] The OSGi Alliance, OSGi Service Platform Core Specification. Release 4, version 4.2, 2009.

[7] D. Rubio, Pro Spring Dynamic Modules for OSGi™ Service Platform, Apress, USA, 2009.

[8] R. M. Fujimoto, Parallel and Distributed Simulation Systems, John Wiley & Sons, New York, 2000.

[9] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," Journal of Systems and Software, vol. 82(1), 2009, pp. 3–22.

[10] P. G. Sapna and H. Mohanty, "Automated Scenario Generation based on UML Activity Diagrams," International Conference on Information Technology 2008, December 2008, pp. 209–214.

[11] J. McAffer, P. VanderLei., and S. Archer, OSGi and Equinox: Creating Highly Modular Java™ Systems, Pearson Education Inc., Boston, 2010.

[12] G. Canfora and M. Di Penta, "New Frontiers of Reverse Engineering," Future of Software Engineering (FOSE'07), Minneapolis, 2007, pp. 326–341.

[13] B. Korel, "Black-Box Understanding of COTS Components," Seventh International Workshop on Program Comprehension, Pittsburgh, 1999, pp. 92–99.

[14] 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.

[15] 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.

[16] F. Naseer, S. U. Rehman, and K. Hussain, "Using Meta-data Technique for Component Based Black Box Testing," 2010 6th International Conference on Emerging Technologies, Islamabad, 2010, pp. 276–281.

[17] J. Ying, Y.-N. Li, X.-D. Fu, "The Support of Interface Specifications in Black-box Components Testing," 2010 Fifth International Conference on Frontier of Computer Science and Technology, Changchun, 2010, pp. 305–311.

[18] M. Fisher and R. Tönjes, "Generating Test Data for Black-Box Testing using Genetic Algorithms," 2012 IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA), Krakow, 2012, pp. 1–6.

[19] N. Bartlell, OSGi in Practice, eBook (Creative Commons), 2009.

[20] I. R. Forman and N. Forman, Java Reflection in Action. Manning Publications Co., Greenwich 2005.

 ISBN: 978-1-61208-371-1 