Interface-based Semi-automated Generation of Scenarios for Simulation Testing of Software Components

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 the component-based software development, an application is constructed from individual reusable components providing particular functionalities. The testing of their functionality, their quality of services, and their extra-functional properties is important, especially when each component can be created by a different manufacturer. We developed the SimCo simulation tool, which enables simulation testing of software components directly without the necessity to create their (potentially incorrect) models. The course of the testing is described in so-called scenarios. In this paper, the semi-automated generation of the scenarios based on the analysis of the interfaces of the particular software components and their mutual interactions is described. The described ideas demonstrate the usefulness of the simulation for the component-based software development.**

Keywords-software components testing; scenarios generation; component interface; component interaction.

I. INTRODUCTION

The component-based software development is an important trend in contemporary software engineering. Using this approach, an application is constructed from individual reusable software components, which provide particular functionalities as services through their public interfaces. Each service of a component can be utilized by multiple other components of the application and the invocation of a service is a basic interaction between the components. A single component can be utilized in multiple applications, which implies the reuse of the existing program code. At the same time, an application can be constructed from components, which were created by different developers [1].

The testing of software components is very important. Their functionality should be ensured by their developers. However, the potentially different developers of the particular components of a component-based application reinforce the need for testing of the interactions and mutual cooperation of the components. Moreover, besides the functionality of the components, it is also necessary to test their extra-functional properties (e.g., memory consumed by a service invocation) and quality of services (e.g., time required for a service invocation or the performance in general) [2][3].

During our previous research, we have developed the SimCo simulation tool, which enables the simulation testing of a single software component, a set of software components, or an entire component-based application [4] [5]. The components can be tested in the SimCo directly, without the necessity to create their (potentially incorrect) simulation models [5]. The course of the simulation testing is described in so-called *scenarios*, which are basically XML files loaded by the SimCo. The manual creation of the scenarios is a lengthy and error-prone process [6]. Therefore, we are now working on several approaches, which would provide a semi-automated generation of the scenarios.

In this paper, we describe one such approach, which is based on the analysis of the public interfaces of the particular software components and on the observation of their mutual interactions. The approach is focused on the situation when the software components are created by a third party and their source codes or any other descriptions of their behavior are unavailable to us.

The paper is structured as follows. The basic notions are briefly discussed in Section II. The SimCo is described in Section III. The existing approaches to generation of testing scenarios are discussed in Section IV. The interface-based scenarios generation, which is the main contribution of this paper, is described in Section V and demonstrated on a case study in Section VI. The paper is concluded in Section VII.

II. BASIC NOTIONS

In order to make the further reading more clear, the basic notions of the component-based software development and the simulation testing will be now briefly described.

A. *Software Components*

In the component-based software engineering, a software component is a black box entity with a well defined public interface and no observable inner state. Its particular functionalities are provided as services accessible via its public interface [1]. An application is then constructed from multiple components, which interact using their interfaces (e.g., invoking particular services). The particular components of an application can be created by different developers, which can be also different from the developer of the resulting application. At the same time, a component can be utilized in multiple applications. Although this definition is very abstract, a more specific definition is problematic, since the view of software components is very diverse [1].

A component model specifies the features, behavior, and interactions of software components. A component framework is then a specific implementation of a component model [5]. So, there can be (and often are) multiple implementations of a single component model [7]. Software components of different component models often have very different form. Hence, it is very difficult to transform an existing software component from one component model to another. Thus, the SimCo utilizes only one component model – the OSGi [8][9]. However, the ideas behind it can be used in other component models as well.

The OSGi is a dynamic component model for Java programming language widespread in both the academic and the industrial spheres including automotive industry, cell phones, and so on [7]. There are multiple implementations of the OSGi component model (i.e. OSGi component frameworks) [8], for example the Equinox [10]. For its widespread use, the OSGi is used in the SimCo [5].

Using the OSGi, the particular software components are referred to as *bundles*. The dynamic nature of the OSGi means that it is possible to install, start, stop, and uninstall any bundle without the necessity to restart the OSGi framework assuming that all bundle dependencies are satisfied [11]. The OSGi bundle has the form of a `.jar` file with additional OSGi-specific meta-information regarding the provided and required services and so on [9]. Since the `.jar` file can contain any number of Java classes, each bundle can provide arbitrary complex functionality [7]. Both provided and required services are described by standard Java interfaces. Hence, the particular services of a bundle have the form of standard Java methods [8].

B. *Discrete-Event Simulation*

A discrete-event simulation is a widely used simulation type [12]. The simulation run of a discrete-event simulation is divided into sequence of time-stamped events, which represent an incremental change to the simulation state. The time stamp specifies the simulation time, in which this change to the simulation state should happen [12].

The simulation time between two succeeding events can be arbitrary long, but the changes of the simulation state are associated with the events only. So, the simulation time "jumps" from the time stamp of an event to the time stamp of the next event [12]. The processing of events is handled by a so-called *calendar* with the list of events ordered by their time stamps. At the start of a simulation run, the calendar removes the first event from the event list, sets the simulation time to the value of the event's time stamp, and performs the event's change of the simulation state. This can potentially lead to the creation of multiple new events, which are inserted to the event list on the positions corresponding to their time stamps. Then, the next event is removed from the event list and so on. The entire process stops when a predefined condition is fulfilled or the event list is empty [12].

C. *Simulation Testing*

Considering only the discrete-event simulation from now on, we will describe the simulation testing of software components. For this purpose, two approaches can be used.

The first approach is to develop a model of the tested software component and use it in the simulation [2]. The advantage of this approach is that the created model is suited for the simulation. The disadvantage is that the creation of the model is time-demanding and potentially error-prone. Since the model usually incorporates only features, which are considered important for the simulation and testing, it is also possible to unintentionally omit some features, which in fact are important for the results of the testing [7]. Also, the consistency of the model has to be maintained.

The second approach is to use the tested software component directly in the simulation, instead of using its model. The advantage of this approach is that there is no necessity to create the model [4]. The disadvantage is that the simulation must enable the running of the tested component in a way it would run in a non-simulation environment [7].

Regardless the approach, the simulation testing can be divided according to the knowledge of the tested software component. If its source code is known, it can be (and usually is) used for the preparation of the course of testing. This is a *white box* testing. If the source code is not known, other information such as a specification of the component's intended behavior, a description of its interface, and so on can be used for the preparation of the course of testing. This is a *black box* testing [13], on which this paper is focused.

D. *Testing Scenarios*

Regardless its type, the simulation testing lies in the subjecting of the tested software component (or its model) to a set of stimuli and observation of its corresponding reactions [14]. In most real cases, it is not feasible to provide a complete coverage of all stimuli. Hence, a subset of stimuli should be created instead. This subset should represent well all theoretical possible stimuli and, at the same time, be reasonably small [13].

The course of the simulation testing is described in so-called *scenarios*, which incorporate the particular stimuli and (optionally) the expected consequences. Using the discrete-event simulation (see Section II.B) for the simulation testing of software components, the stimuli correspond to particular events. Each event, in turn, corresponds to the invocation of a service of a tested software component [7]. The consequences can then be a return value, an exception, a subsequent invocation of a service of another component, and so on.

III. SIMCO SIMULATION TOOL

Now, as we discussed the basic notions, we can proceed with the description of the SimCo, which we developed during our previous research.

A. *Purpose and Description of SimCo Simulation Tool*

The SimCo enables testing of software components using the discrete-event simulation (see Section II.B). The SimCo itself is a component-based application, which enables its simple extensions and modifications [15]. It utilizes the OSGi component model [8][9]. Currently, it runs in the Equinox OSGi framework [10] (see Section II.A). Consequently, it is utilizable only for the testing of the OSGi bundles (OSGi software components) [7].

The tested components are directly incorporated into the simulation. Hence, the simulation environment must be constructed in a way, which enables a normal running of the tested components as they would be in a non-simulated environment. Among other things, the dependencies of the components must be satisfied. At the same time, the course of the simulation testing must be under control during the entire simulation run. These requirements are reflected in the structure of the SimCo, which consists of several software component types [15] (see Section III.B).

B. Types of Software Coponents in SimCo Simulation Tool

There are four types of software components (OSGi bundles) in the SimCo – the *core*, the *real tested*, the *simulated*, and the *intermediate* components [15].

The core components provide the functionality of the SimCo itself. Primarily, they ensure the running of the simulation according to the loaded testing scenario and the collection of the results. The most important core component is the `Calendar`, which ensures the progress of the simulation run by processing the events from the event list. The core components also provide supplementary functions such as logging, visualization of the simulation, and so on [15].

The real tested components are the OSGi bundles, which shall be tested using the SimCo. The source code of these components may be known or unknown, since the components can be created by a third party [15]. If the source code is known (i.e. white box testing) it can be used for the creation of the testing scenarios. However, in this paper, we focus on the case when the source code is not known (i.e. black box testing) and only the public interfaces of the components (with or without further description) are known. The real tested components run in the simulated environment, which brings several issues (e.g., discrepancy of the simulation and real time, undesirable remote communication, etc.) solvable by various means (see [5] and [15]).

The simulated components provide services, which are required by the real tested components. They are useful when a single component or a limited set of components are tested and these components require services of other components, which are not present in the simulation. Then, the simulated components provide the missing services required by the real tested components. When an entire component-based application is tested, the simulated components are not necessary, since the requirements of all real tested components should be satisfied. Another important function of the simulated components is the ensuring that all invocations of the services of these components are performed using the events and the calendar. This allows the SimCo to maintain the control of the simulation run. Additionally, the simulated components can be used for the speedup, since they do not have to perform all the calculation they real counterparts would. Instead, they can utilize random numbers generators or prerecorded values, depending on the situation [15].

The intermediate components are inserted between all pairs of real tested components. They ensure that the invocations on the real tested components are handled using the events and the calendar. Basically, an intermediate component is a proxy for a real tested component. It has the same

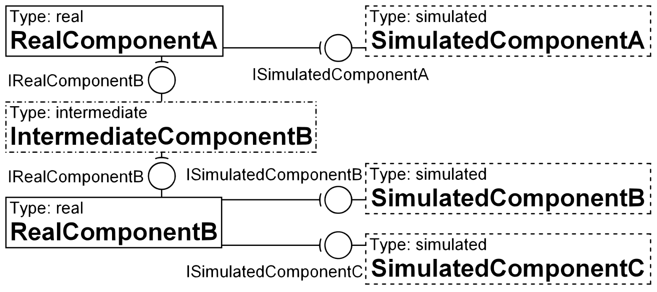

Figure 1. Relations of the three types of SimCo components

public interface (i.e. provides the same services). All invocations of the services on the real tested component are in fact handled using this proxy (see Figure 1). When a service is invoked, the intermediate component ensures the use of the events and the calendar and invokes the corresponding service on the real tested components to obtain the return value and potentially trigger further consequences (e.g., invocation of a service of another component) [15].

C. Testing Scenarios in SimCo Simulation Tool

The testing scenarios for the SimCo have the form of XML files. They contain primarily the events, which shall be processed during the simulation run [15]. The events correspond to the invocations of the particular services of particular components (see Section II.D). The scenarios also contain the configuration of the simulation and the layout of the components (e.g., information, which components are real tested, which are simulated, etc.).

Each event in the testing scenario is usually stored as a unique record containing the time stamp and the description of the invocation that shall be performed – the component, on which the invocation shall be performed, the service, which shall be invoked, and the parameter values of the invocation. The expected consequences of the invocation can be stored as well. They are useful for evaluation of the simulation tests – it can be easily compared whether the consequences of the invocation during the simulation run are the same as the expected ones. This is particularly useful during the testing of the correct functionality of the components. Nevertheless, the expected consequences are not an obligatory part of the testing scenarios, since it is also possible to use logging and probes for evaluation of the simulation tests, separately from the scenarios. This can be useful during the testing of the extra-functional properties and quality of services of the components.

As an alternative to describing every single event separately, it is also possible to specify a random numbers generator with an appropriate probabilistic distribution [15]. This way, it is possible to generate multiple events (with preset parameter values or even randomly generated parameter values) with a single description within the testing scenario. However, this approach can be used only in some cases.

IV. Scenarios Generation Approaches

The idea of the semi-automated generation of the testing scenarios is not new in the software development field. The research in this area begun at least in 1970s [16] and continues till today [6][13][14][17]. However, the research focused on the semi-automated generation of the testing scenarios directly for the software components is rare. So, several existing approaches, which are briefly discussed in the following sections, are mostly not intended for the software components, but still can serve as a source of inspiration. They have in common that they do not utilize the source code of the tested application, but rather a description of its behavior and requirements.

A. UML Behavioral Diagrams

Many approaches for the generation of the testing scenarios are based on UML behavioral diagrams (e.g., sequence diagram, activity diagram, etc.) [18].

The activity diagrams are used, for example, for a representation of concurrent activities [13]. The exhaustive exploration of the diagrams is then used for the generation of the testing scenarios. Since, for large applications, the exploration of all possible flows in the diagrams is infeasible, some constraints derived from the application domain are used to discard illegal or irrelevant scenarios [13]. The activity diagrams are also used in [17] where they are generated using multiple UML use case diagrams in order to express the concurrency of the particular use cases. The exploration of the diagrams is again used for the generation of the scenarios [17]. In [19], a layered activity diagram describing the workflow of the tested application is used for the extraction of the execution paths of the tested application. These execution paths are then used for the generation of the scenarios [19].

B. User Interface

Several approaches utilize the user interface for the generation of the testing scenarios. This is based on the assumption that the user interface provides access to the majority (if not all) functions of the tested application [20].

In [20], the testing scenarios are created by inducing inputs to the user interface and pairing them with corresponding outputs. An opposite approach, which is focused on the testing of software components created using the .NET platform, is described in [21]. In this case, the absence of the user interface of the components is stressed as a major setback for the testing. Hence, reverse engineering is used for determination of the classes, methods and attributes of the particular components. Using this data, a basic graphical user interface is generated for each component [21].

C. Natural Language Specification

Using the specification of the tested application written in natural language is an appealing approach for creating of the testing scenarios. However, this approach is still very difficult to implement due to the ambiguity, poor understandability, incompleteness, and inconsistency of the natural language [22]. Hence, some restrictions are often used to overcome these difficulties.

An example can be found in [23] where a restricted form of natural language is used for descriptions of the use cases. From these descriptions, a control-flow-based state machine is generated for each use case. Then, these state machines are combined into a global system level state machine. The testing scenarios can be then generated by exploration of this state machine [23].

A similar approach, based on the use case descriptions written in natural language, is considered for the SimCo as well. However, it employs different restrictions and different course of generation of the testing scenarios. The use-case descriptions are analyzed and transformed into an overall behavioral automaton (OBA) using the FOAM tool [24]. From this automaton, the testing scenarios can be generated [7]. However, the use cases descriptions must be written using the rules described in [25] and (manually) enriched by annotations describing the flow of the program and its temporal dependencies. For more details, see [7].

V. Interface-based Scenarios Generation

The approach to semi-automated generation of the testing scenarios, which is described in this paper in detail, is based on the analysis of the interfaces of the particular software components and observation of their interactions. The basic idea has been described already in [7], but its extension, the formulation of the algorithm, and its specific application using a case study is the main contribution of this paper.

A. Main Ideas, Features, and Limitations

The approach is intended for the situation when the source code of the particular software components is not known, but their implementations are at our disposal. The generated scenarios can then be used for testing of the extra-functional properties and the quality of services of the components. The specific requirements (e.g., maximal duration of an invocation) are not generated, but can be easily added by the user. The generated scenarios are also particularly useful when a new component shall replace its older version and we want to test whether the new component exhibits the same external behavior as the old one.

Assume now that there is an entire component-based application, for which the testing scenarios shall be generated, and the source code is not known for some or all of its components. This entire application can be then imported to the SimCo. During the import, the intermediate components are placed between each pair of the components, since all of them are real. Alternatively, some of the components can be replaced by their simulated counterparts as long as they exhibit the same external behavior as the original real components.

Once the import is complete, the public interfaces of the components (OSGi bundles) along with their services and their parameters are determined. Then, the parameter values of the particular invocations can be partially generated automatically and partially provided by a user. Using these parameter values, the services of the particular components are sequentially invoked and the consequences of the invocations are observed using the intermediate components (and simulated components if they are present). The consequences can be an exception, a return value, a subsequent invocation of a service or services on another component or other components, or solely a change of the inner state of the component. With the exception of the unobservable inner state, all consequences can be recorded.

From all the types of the consequences, the subsequent invocations are the most important ones. The reason is that, with each subsequent invocation of a service, its parameter values are observed. These parameter values can be then added to the parameter values, which were automatically generated or provided by the user (see previous paragraph), but, unlike them, the parameter values from the subsequent invocations are genuine, provided directly by the component, which invokes the particular service. Moreover, the parameter values can contain instances of objects, which are difficult to generate automatically. This useful feature is possible, because the parameters of the subsequent invocations are present in the components, but are hidden from us, since the source code is not known.

From the invocations and their consequences, the scenarios can be directly generated. It should be noted that, if there is an erroneous behavior in the component-based application (e.g., a subsequent invocation, which in fact should not occur), the errors will propagate into the generated scenarios. Therefore, the user is encouraged to check the generated scenarios. However, in most cases, we assume that the component-based application is working correctly.

B. *Algorithm*

The algorithm of our approach to the testing scenarios generation consists of three steps – the preparation of a tree data structure of all the components and all their services, the generation of invocations of particular services along with their parameter values, and the performing the particular invocations in order to determine their consequences and supplement additional invocations and their parameter values.

The tree data structure can be initiated easily using the analysis of the public interfaces of the software components. For this purpose, a service of the OSGi framework, which enables to determine the public interface (i.e. a Java interface – see Section II.A) of a component (OSGi bundle), can be used. With the Java interfaces of the particular components known, the Java reflection can be used for the determination of their services (i.e. Java methods – see Section II.A) with types of their parameters and return values.

For each service of each component in the initiated tree data structure, the list of invocations is added. Each invocation contains the set of parameter values. These parameter values can be partially generated and partially supplemented by the user, depending on the type of the parameters. For the `enum` type, all possible values including the `null` value are generated. For the `boolean` types, both possible values are generated. For the `char` type, all single-byte values are generated. For the number types, a set of representative values are generated (e.g., -128, -100, -10, -1, 0, 1, 10, 100, and 127 for the `byte` type). For an object type, only the `null` value is generated. If there are multiple parameters for a single service, all combinations are generated. In order to keep the number of various invocations feasible, the user can provide restrictions for the generation of the parameters. Moreover, the user can manually add parameters, which are difficult to be generated automatically (e.g., specific instances for an object type). The scheme of the tree data structure with filled lists of invocations is depicted in Figure 2.

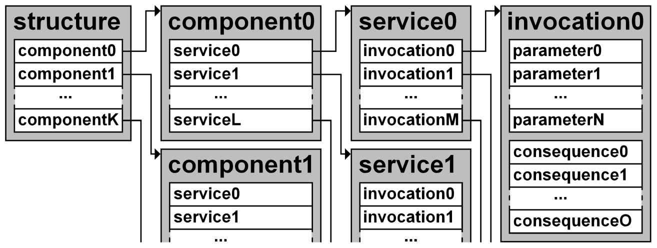

Figure 2. Scheme of the tree data structure

Once the tree data structure is initiated and filled with the invocations (see Figure 2), it is sequentially explored component by component, service by service, invocation by invocation. Each invocation from the tree data structure is performed on the corresponding component and its consequences are observed using the intermediate components (and the simulated components if they are present). These consequences (a return value, an exception, a subsequent invocation) are added to the tree data structure if they are not already present. If the consequence is a subsequent invocation on another component and it is not already present, this invocation along with its parameter values is added to the tree data structure as well.

```
//Initialization of the structure of components
structure.components = simco.getComponents();

//Initialization of their services
for (c: structure.components) {
  c.services = simco.getServicesOfComponent(c);

  //Generation and filling (by the user) of the
  //invocations
  for (s: c.services) {
    s.invocations = simco.generateAndReadInvocations();
  }
}

//Exploration of invocations and consequences
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) {
          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;
            if (!structure.contains(sc, ss, si)) {
              structure.addInvocation(sc, ss, si);
              structure.setChanged(true);
            }
          }
        }
      }
    }
  }
}
```

Figure 3. The pseudocode of the entire algorithm

Once the invocation is completed and all its consequences are added to the tree data structure, the next invocation is performed and so on. When the entire structure is explored (i.e. all invocations were performed), it is checked whether new invocation consequences or subsequent invocations were added to the structure during the just finished exploration. If so, the exploration of the entire structure repeats. Otherwise, the algorithm ends (see Figure 3).

Once the algorithm is complete, its result is the tree data structure filled with the invocation consequences and the subsequent invocations. This filled data structure can be visualized as a tree (see Figure 2) or as a multilevel list (see Figure 5) for the user, if required. The structure can be then directly transcribed to the resulting scenario (an XML file). Should the extra-functional properties and the quality of services be tested, their descriptions must be added to the scenario by the user, since they cannot be determined from the components or they behavior without further description. The scenario can then be loaded by the SimCo, which performs the invocations with the parameters specified in the scenario and observes their consequences.

VI. TRAFFIC CROSSROAD CONTROL CASE STUDY

The utilization of the described approach to the semi-automatic generation of testing scenarios will be demonstrated on a case study – the Traffic crossroad control.

A. *Description of Traffic Crossroad Control*

The Traffic crossroad control is a component-based application for the control of road traffic in a crossroad using traffic lights. It is expected to run in an OSGi framework on a specific hardware and operate a variety of hardware sensors and control units [5]. Since these sensors and control units are irrelevant for this paper, the components involving any direct contact with them were replaced by manually created simulated components in order to enable the running of the application on a standard desktop computer [7].

The scheme of the entire application is depicted in Figure 4. The application consists of eight components, from which four components are real and four components are simulated. Moreover, each real component has its own intermediate component (see Figure 4).

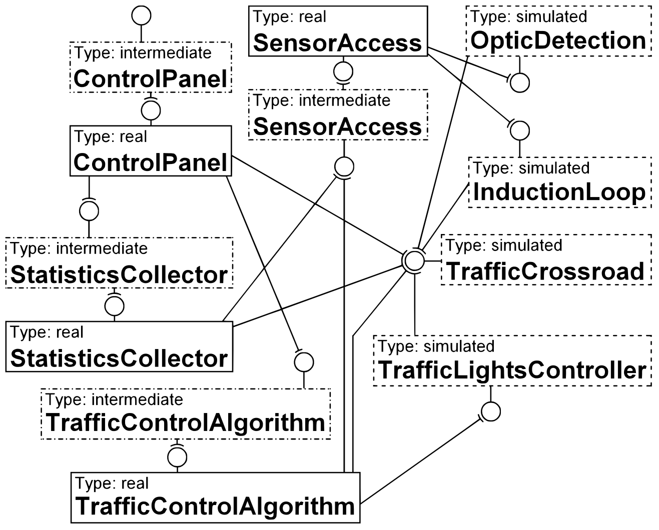

Figure 4. The scheme of the Traffic crossroad control application

The `TrafficCrossroad` component provides the information about the structure of the crossroad. Its simulated version also incorporates a nanoscopic road traffic simulation replacing the real traffic [7]. The `Induction-Loop` and the `OpticDetection` components provide measured data from corresponding hardware sensors. Their simulated versions extract the data from the road traffic simulation instead. The `TrafficLightsController` component ensures desired settings of the traffic lights using corresponding hardware control units. Its simulated version sets the traffic lights in the road traffic simulation instead. The `ControlPanel` component provides the user interface for the entire application. The `TrafficControlAlgo-rithm` component contains an algorithm for the control of the traffic lights. This algorithm can require information from the sensors mediated by the `SensorAccess` component. From this component, the `StatisticsCol-lector` component periodically collects data and provides various road traffic statistics [7].

B. *Scenarios Generation for Traffic Crossroad Control*

Assume now the utilization of the algorithm described in Section V.B on the components of the Traffic crossroad control application. During the initiation of the tree data structure, the user provides eight sets of parameter values for two services of the `SensorAccess` component (string IDs of the particular sensors) and eight sets of parameter values for three services of the `TrafficControlAlgorithm` component (instances with parameters of the traffic control algorithm). The remainder of the parameter values is generated. The tree data structure is then explored and the invocation consequences and subsequent invocations are added to this structure.

The parts of the resulting filled structure are depicted in Figure 5. The components, services, invocations, and invocation consequences are marked with C, S, I, and IC, respectively. The value in the parentheses of I denotes the originator of the invocation (G – generated at the beginning, U – provided by the user, E*x* – added automatically during the *x*th exploration). The value in the parentheses of IC denotes the number of the exploration of the structure (starting with 1), in which the invocation consequence was added. The overall statistics of the filled tree data structure are summarized in Table I.

TABLE I. OVERALL STATISTICS OF THE FILLED TREE DATA STRUCTURE

Feature	Count
Invocations generated at beginning (G)	39
Invocations provided by user (U)	16
Automatically added invocations in 1st exploration (E1)	4
Automatically added invocations in 2nd exploration (E2)	4
Automatically added invocations in 3rd exploration (E3)	0
Generated consequences in 1st exploration (1)	68
Generated consequences in 2nd exploration (2)	21
Generated consequences in 3rd exploration (3)	0
Explorations	3

```
C: ControlPanel
 S: boolean isTrafficLightsActivated()
  I(G): isTrafficLightsActivated()
   IC(1): return value [true]
 S: void setTrafficLightsActivated(boolean)
  I(G): setTrafficLightsActivated(true)
   IC(1): subsequent invocation
    [TrafficControlAlgorithm.isActivated()]
   IC(2): subsequent invocation
    [TrafficControlAlgorithm.setActivated(true)]
  I(G): setTrafficLightsActivated(false)
   IC(1): subsequent invocation
    [TrafficControlAlgorithm.isActivated())
   IC(1): subsequent invocation
    [TrafficControlAlgorithm.setActivated(false)]
                                  ...
C: SensorAccess
 S: int getQueueLength(String)
  I(G): getQueueLength(null)
   IC(1): exception [NullPointerException]
  I(U): getQueueLength("E_01")
   IC(1): subsequent invocation
    [OpticDetection.getVehiclesCount("E_01")]
   IC(1): return value [2]
   IC(2): subsequent invocation
    [InductionLoop.isVehicle("E_01")]
   IC(2): return value [1]
                                  ...
 S: boolean isVehicle(String)
  I(G): isVehicle(null)
   IC(1): exception [NullPointerException]
  I(U): isVehicle("E_01")
   IC(1): subsequent invocation
    [OpticDetection.getVehiclesCount("E_01")]
   IC(1): return value [true]
   IC(2): subsequent invocation
    [InductionLoop.isVehicle("E_01")]
   IC(2): return value [true]
                                  ...
 S: DetectorType getDetectorType()
  I(G): getDetectorType()
   IC(1): return value [DetectorTypes.OPTIC]
 S: void setDetectorType(DetectorTypes)
  I(G): setDetectorType(null)
   IC(1): exception [NullPointerException]
  I(G): setDetectorType(DetectorTypes.OPTIC)
   IC(1): nothing observable
  I(G): setDetectorType(DetectorType.INDUCTION)
   IC(1): nothing observable

C: StatisticsCollector
 S: Statistics getStatistics()
  I(G): getStatistics()
   IC(1): return value [statistics]
                                  ...
C: OpticDetection
 S: int getVehiclesCount(String)
  I(G): getCurrentVehiclesCount(null)
   IC(1): exception [NullPointerException]
  I(E1): getCurrentVehiclesCount("E_01")
   IC(1): return value [2]
                                  ...
C: InductionLoop
 S: boolean isVehicle(String)
  I(G): isVehicle(null)
   IC(1): exception [NullPointerException]
  I(E2): isVehicle("E_01")
   IC(2): return value [true]
                                  ...
```

Figure 5. Selected parts of the explored and filled tree data structure

As seen in Table I, the tree data structure is explored three times in this case. During the third exploration, no new invocation consequences and no new invocations are generated, which means that the structure does not change and the algorithm ends (see Section V.B).

There are four newly generated invocations of the service `getVehiclesCount()` of the `OpticDetection` component added during the first exploration and four newly generated invocations of the service `isVehicle()` of the `InductionLoop` component during the second exploration. These invocations are direct consequences of the invocations of the `getQueueLength()` service of the `SensorAccess` component – not its `isVehicle()` service, which would have the same consequences, but is invoked after the `getQueueLength()` service, so the invocations are already present. In other words, the invocations provided by the user for the `SensorAccess` component propagate automatically to other components (`OpticDetection` and `InductionLoop` in this case) and enable their better testing. This is a very useful feature of our approach.

It should also be noted that both sets of the subsequent invocations described in the previous paragraph are generated by the same invocation, but in different explorations. The reason is that the inner state of the `SensorAccess` component is being changed during the exploration of the tree data structure by its `setDetectorType()` service. So, although the inner state of the components is unobservable, it can (and often does) influence the behavior of their services. Due to this, the services, which lack any observable consequence, but presumably change the inner state of the components, can influence the generation of the invocation consequences. So, the repetitive explorations of the tree data structure maximize the number of generated invocation consequences.

As it was mentioned in Section V.B, the filled explored tree structure can be directly transcribed to the resulting scenario, which is a XML file. The part of this scenario for the structure depicted in Figure 5 is depicted in Figure 6.

```
<scenario>
  <invocation time="1" componentName="ControlPanel"
      serviceName="isTrafficLightsActivated">
    <parameters>
    </parameters>
    <consequences>
      <consequence type="RETURN_VALUE"
          dataType="boolean" value="true" />
    </consequences>
  </invocation>
  <invocation time="2" componentName="ControlPanel"
      serviceName="setTrafficLightsActivated">
    <parameters>
      <parameter dataType="boolean" name="arg0"
          value="true">
    </parameters>
    <consequences>
      <consequence type="SUBSEQUENT_INVOCATION"
          componentName="TrafficControlAlgorithm"
          serviceName="isActivated">
        <parameters>
        </parameters>
      </consequence>
      <consequence type="SUBSEQUENT_INVOCATION"
          componentName="TrafficControlAlgorithm"
          serviceName="setActivated">
        <parameters>
          <parameter dataType="boolean" name="arg0"
              value="true">
        </parameters>
      </consequence>
    </consequences>
  </invocation>
                          . . .
</scenarios>
```

Figure 6. A part of the resulting scenario

VII. Conclusion

In this paper, we described an approach to the semi-automated generation of the scenarios for the simulation testing of software components. The approach is based on the analysis of the interfaces of the particular software components and observation of their mutual interactions. The approach is intended for situations when the source code of the components is unknown, but their implementations are available. Then, it enables to partially analyze the behavior of the particular components. The resulting scenarios are useful for the testing of the functionality of new versions of the components, which shall replace their older versions, and as the basis for the testing of the extra-functional properties and quality of services of the components. In the latter case, however, the descriptions and constraints of the extra-functional properties and the quality of services must be filled into the generated scenario by the user.

The functioning of the approach was demonstrated on the Traffic crossroad control case study. The approach was designed for the SimCo simulation tool and OSGi component model, but its basic ideas are utilizable for other existing component models as well.

In our future work, we will focus on the better exploration of the structure, from which the scenarios are generated. This includes ordering of the particular service invocations and attempting to extrapolate the inner states of the components.

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] 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.

[3] P. C. Heam, O. Kouchnarenko, and J. Voinot, "Component Simulation-based Substitutivity Managing QoS Aspects," Electronic Notes in Theoretical Computer Science, vol. 260, 2010, pp. 109–123.

[4] 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.

[5] 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.

[6] D. Xu, H. Li, and C. P. Lam, "Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams," Proccedings of the 12th Asia-Pacific Software Engineering Conference, December 2005.

[7] 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.

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

[9] R. S. Hall, K. Pauls, S. McCulloch, and D. Savage, OSGi in Action: Creating Modular Applications in Java, Manning Publications Co., Stamford, 2011.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] J. Bauer and A. Finger, "Test Plan Generation Using Formal Grammars," Proceedings of the Fourth International conference on Software Engineering, Los Alamitos, September 1979, pp. 425–432.

[17] 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.

[18] M. Shirole and R. Kumar, "UML Behavioral Model Based Test Case Generation: A Survey," ACM SIGSOFT Software Engineering Notes, Vol. 28(4), 2013, pp. 1–12.

[19] Y. Yongfeng, L. Bin, L. Minyan, and L. Zhen, "Test Cases Generation for Embedded Real-time Software Based on Extended UML," 2009 International Conference on Information Technology and Computer Science, Kiev, 2009, pp. 69–74.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] V. Simko, D. Hauzar, T. Bures, P. Hnetynka, and F. Plasil, "Verifying Temporal Properties of Use-Cases in Natural Language," Springer-Verlag, vol. 7741, 2013, pp. 350–367.

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