Testing Component-Based Systems Using FSMs

Sami Beydeda[1] and Volker Gruhn[2]

[1] Federal Finance Office (Bundesamt für Finanzen)
Friedhofstr. 1
53225 Bonn, Germany
sami.beydeda@bff.bund.de
[2] University of Leipzig
Chair of Applied Telematics / e-Business
Klostergasse 3
04109 Leipzig, Germany
gruhn@ebus.informatik.uni-leipzig.de

Summary. No matter which tools, techniques, and methodologies are used for software development, it remains an error-prone process. Nevertheless, changing such important constituents of the software process surely has an effect on the types of faults inherent in the developed software. For instance, some types of faults are typical for structured development, whereas others are typical for object-oriented development.

This chapter explores the question of whether component-based software requires new testing techniques, and proposes an integrated testing technique. This technique integrates various tasks during testing component-based software: white- and black-box testing of the main component (i.e., the top level component controlling the other components), black-box testing of components, black-box testing of the middleware and integration testing of the main component with other components.

Benefits of this technique are shown using a real-world example: the technique is automatable and applicable to existing component-based software.

1 Introduction

In the last few years, component-based software development has received much attention from both researchers and practitioners. Components seemed to be the *silver bullet* software engineers have sought for decades. Researchers have developed component models together with the necessary technologies, which have been applied by practitioners in their daily work.

Unfortunately, testing of such software systems has not gained enough interest. In the opinion of both practitioners and researchers, a component once

sufficiently tested does not require testing when reused. But experience shows that this belief to be wrong, because components are often initially tested with respect to a certain application domain, thus failing in new environments [423].

The first subsection identifies the properties of components in order to understand components and component-based software. Furthermore, this subsection gives the reasons why traditional testing techniques cannot be used for testing this kind of software. The second subsection outlines several requirements for testing techniques for component-based software.

1.1 Properties of Components Affecting Testing

Although components and development of software based on components were discussed by computer scientists more than 30 years ago [262], there is no common definition of these terms available to date. Different authors define the term *component* differently, although all have roughly the same concepts in mind [48, 49]. Another reason for this is that developers of component models like Sun Microsystems with its JavaBeans[3] and Enterprise JavaBeans models[4], and Microsoft Corporation with COM[5], have a diverse technical understanding of the notion of a component. However, we do not try to define these terms. Rather, we establish a list of properties most components have in common. Although this list cannot be complete, it is sufficient to investigate the implications of component-based development on testing. Components typically share the following properties:

- Components are (nearly) independent and replaceable parts of a system. A component conforms to and provides a set of interfaces [48].
- Usually, a component possesses an internal state which affects results delivered by its methods and its dynamic behavior.
- Special components, called *commercial-off-the-shelf* (COTS), can be purchased on a component market. Often, these types of components are delivered without their source code.
- Since components may be distributed over several computers, communication among components requires middleware technologies like CORBA[6] and DCOM[7].

Obviously, software development will remain an error-prone process even if component-based development contributes to an improvement of software quality. However, an important question at this point is *do the above properties of a typical component affect testing?*

Predictably, component-based software requires techniques for testing other than those for traditional software. There are several reasons for this:

[3] `http://www.javasoft.com/beans/`
[4] `http://java.sun.com/products/ejb/`
[5] `http://www.microsoft.com/com/`
[6] `http://www.omg.org/corba/`
[7] `http://www.microsoft.com/com/`

- In contrast with in-house developed software, white-box testing cannot be conducted for COTS, since their source code is usually not available.
- Traditional testing techniques do not consider middleware technologies. But the middleware of component-based software is an integral part of it, and has also to be tested.

1.2 Requirements for Testing Techniques

Having identified the reasons as to why traditional testing techniques cannot be applied to component-based software, the next question is *what requirements do a testing techniques have for component-based software?*

With respect to the reasons explained above, such testing techniques have to fulfill the following requirements:

White- and black-box testing of the main component. Generally, software has to be tested using several techniques. Specifically, these techniques have to include both white- and black-box techniques [24]. Since few techniques combine the white- and black-box approach [38,65], the main component has to typically be tested first using test cases generated on the basis of the source code, and then using test cases generated on the basis of its specification.

Black-box testing of components. Since source code may not be available for some components, they can only be black-box tested. In fact, black-box testing of components is usually sufficient, because at this level there is often no need for low-level testing techniques based on source code. Harrold et al. [170] distinguish two different perspectives during testing in component-based development. One of these perspectives is called the *component-provider perspective* and refers to the testing of components by their developers on the basis of the source code. Since developers have detailed knowledge about the internal structure of their components, they can test their components more effectively than users. The other perspective, called the *component-user perspective*, refers to the testing of components by their users, without access to their source code. Even if source code is available, components should be only black-box tested, because of the missing knowledge about the internal structure of the component and the danger of losing oneself in too much detail. In this article, we assume that components, except for the main component, are tested by their users. Thus, testing is performed with the component-user perspective.

Testing only the required component functionality. Often the main component requires only a subset of the functionality a component provides. Thus, testing techniques for component-based software have only to test these required subsets [170,350,422]. Experience shows that these subsets have to actually be tested, because components are often tested with respect to a special application domain, and thus subsequently fail in new environments [423].

Testing the middleware. Test cases have to be provided for testing the middleware used. It is obvious that failures of middleware are likely to influence the behavior of the entire system. Therefore, testing of component-based software has also to aim at ensuring the absence of faults in the middleware layer.

Testing the interaction of the main component. Another important issue for such testing techniques concerns integration testing. Integration testing consists of testing the interaction of the main component with other components. Even if all components are free of faults, a system consisting of them may fail due to wrong interactions.

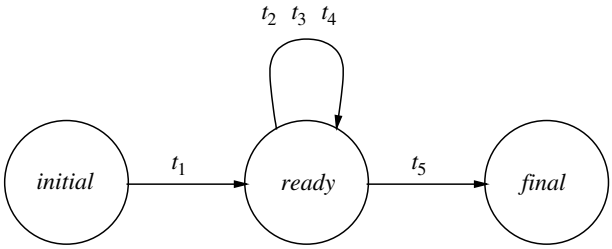
The requirements explained above make clear that component-based software cannot be sufficiently tested with existing techniques. Therefore, research in this area has to aim at developing new techniques suited for these requirements.

2 Demonstrative Example

In this section, an example from the banking area, used in the remainder of this article, is introduced. The example used to demonstrate our testing technique is a system called `BankApp`, used for making deposits and withdrawals on bank accounts. The `BankApp` system consists of the main component, a component called `account`, and a middleware component, required for interaction of the main component with the `account` component. Each of these constituents is described in detail below.

2.1 The `account` Component

As its name suggests, the `account` component simulates the bank customers' account. It encapsulates ID and account balance and provides suitable methods to check and alter the balance. Figure 1 shows the specification of this component. In our approach, the specification for each component is given as a special finite state machine, called *component state machine* (CSM), which corresponds to *class state machines* described by Hong et al. [183]. In fact, the component state machines used in our technique originate from their class state machines. The main difference between a class state machine and a component state machine is that transitions within a component state machine are augmented with Java code in order to allow automatic generation of an executable frames for event types and executable *oracle* for testing. An oracle determines the result a program should compute for a particular input, with respect to the specification, and compares this result with that obtained by actually executing the program for the same input. Executable oracles have been used, for instance, by Hoffman and Strooper [179, 180]. However, automatic generation of executable oracles is left for further study.



	source	target	event	guard	action
t_1	*initial*	*ready*	`Account(accountId,initialBalance)`	`true`	`Id=accountId; balance=initialBalance;`
t_2	*ready*	*ready*	`deposit(amount)`	`true`	`balance+=amount;`
t_3	*ready*	*ready*	`withdraw(amount)`	`amount<=balance`	`balance-=amount;`
t_4	*ready*	*ready*	`balance()`	`true`	`return balance;`
t_5	*ready*	*final*	`remove()`	`true`	`home.remove(accountId);`

Fig. 1. Specification of the `account` component as a component state machine.

In Fig. 1, abstract states of the component are represented by circles, while each transition is depicted as an arrow leading from its source state to its target state. These transitions are formally specified through 5-tuples $(source, target, event, guard, action)$. A transition consists also – in addition to a *source* and a *target* state – of an *event* causing the transition, a predicate *guard* which has to be fulfilled before the transition can occur, and the *action* defining the operations on the component variables during the transition. As explained above, the *guard* and the *action* of a transition are defined using Java statements for automation purposes.

As depicted in Fig. 1, a CSM includes two special circles, labeled *initial* and *final*. These two circles represent the state of a component before its creation and after its destruction, respectively. Thus, they represent states in which the component variables and their values are not defined, meaning that these two states are abstract states of a component. Furthermore, Hong et al. [183] have proposed introducing an *error* state representing the state of a class after an error has occurred. In the following figures, the *error* state has been omitted for the sake of clarity. However, components are supposed to enter the *error* state after the occurrence of an event that is either not specified for a particular state or does not fulfill one of the guards.

After initialization, the `account` component enters the *ready* state. The *ready* state indicates that the account has been initialized and is ready to receive deposit and withdrawal requests. After each deposit request, the `account` component remains in the *ready* state. However, withdrawal requests can imply a change to the *error* state. The *error* state is entered when clients of the `account` component try to overdraw from the account. The `account` component also provides an observer method, called `balance()`, for checking the account balance. A change to the *final* state is triggered by invoking the

`remove()` method. This method does not correspond to a finalize (destructor) method. Rather, it invokes the appropriate method of the middleware which then destroys the `account` component. In the specification of the `account` component in Fig. 1, an account is modeled by a Java object having the two attributes `Id` and `balance`.

2.2 The Middleware Component

The `BankApp` system is implemented using the Enterprise JavaBeans technology[8] of Sun Microsystems. The Enterprise JavaBeans specification defines a component architecture for building distributed, object-oriented applications in Java. Each component in the Enterprise JavaBeans technology, called *bean*, is encapsulated in a server which addresses multithreading, resource pooling, clustering, distributed naming, automatic persistence, remote invocation, transaction boundary management, and distributed transaction management. More information can be found on the Internet[9].

For testing the `BankApp` system, a CSM is used that specifies only a small subset of features an Enterprise JavaBeans server provides. The CSM depicted in Fig. 2 models only the resource pooling functionality of an Enterprise JavaBeans server. To keep the example simple, other important features such as automatic persistence and remote invocation have not been addressed.

The simple server used in the example possesses two main states, namely, *capacityAvailable* and *capacityLimit*. Assuming that this simple server does not provide an automatic persistence functionality, it enters, after its invocation and initialization, the *capacityAvailable* state. The server maintains a pool that can store several `account` beans. Since this pool has a limited capacity, the state of the server changes to the *capacityLimit* state after the creation of a certain number of beans[10]. The server also provides a method which does not change its state. The `findByPrimaryKey()` selects a particular account, identified by its ID, from the pool. The specification in Fig. 2 uses a `Hashtable` object as a pool defined in the `java.util` package.

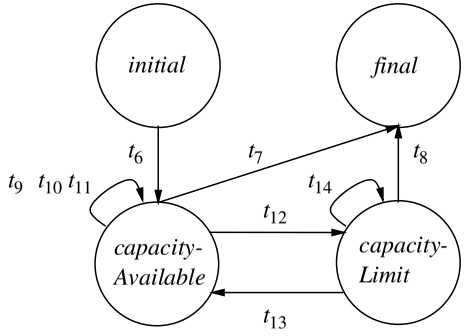
2.3 The Main Component

Figure 3 contains the specification of the main component. Assuming that persistence is not addressed, the main component enters the *accountNotAvailable* state after its initialization . The *accountNotAvailable* state indicates that a certain account referred by `ac` does not exist. This state can be changed by creating a new account using

[8] `http://java.sun.com/products/ejb/`

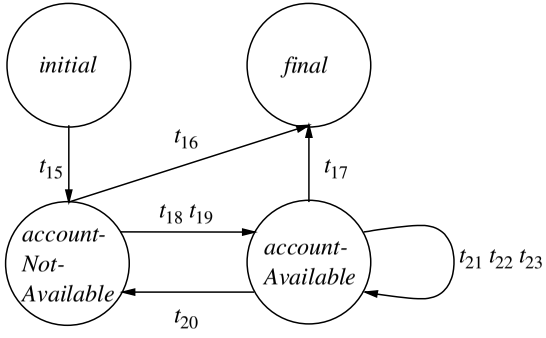
[9] `http://www.javasoft.com/`

[10] Note that in the simple example used, *activation* and *passivation* of beans are not considered.



	source	target	event	guard	action
t_6	*initial*	*capacityAvailable*	`AccountHome()`	`true`	`int capacity=3;` `p=new Hashtable(capacity);`
t_7	*capacityAvailable*	*final*	`finalize()`	`true`	
t_8	*capacityLimit*	*final*	`finalize()`	`true`	
t_9	*capacityAvailable*	*capacityAvailable*	`create(id,balance)`	`!p.containsKey(id)` `&& p.size()<capacity-1`	`ac=new Account(id,balance);` `p.put(id,ac);`
t_{10}	*capacityAvailable*	*capacityAvailable*	`remove(id)`	`p.containsKey(id)`	`p.remove(id);`
t_{11}	*capacityAvailable*	*capacityAvailable*	`findByPrimaryKey(id)`	`p.containsKey(id)`	`return p.get(id);`
t_{12}	*capacityAvailable*	*capacityLimit*	`create(id,balance)`	`!p.containsKey(id)` `&& p.size()==capacity-1`	`ac=new Account(id,balance);` `p.put(id,ac);`
t_{13}	*capacityLimit*	*capacityAvailable*	`remove(id)`	`p.containsKey(id)`	`p.remove(id);`
t_{14}	*capacityLimit*	*capacityLimit*	`findByPrimaryKey(id)`	`p.containsKey(id)`	`return p.get(id);`

Fig. 2. Component state machine of a simple Enterprise JavaBeans server.



	source	**target**	**event**	**guard**	**action**
t_{15}	*initial*	*accountNotAvailable*	`main()`	`true`	`home=new AccountHome();` `ac=null;`
t_{16}	*accountNotAvailable*	*final*	`exit()`	`true`	
t_{17}	*accountAvailable*	*final*	`exit()`	`true`	
t_{18}	*accountNotAvailable*	*accountAvailable*	`createAccount(id,balance)`	`true`	`ac=home.create(id,balance);`
t_{19}	*accountNotAvailable*	*accountAvailable*	`lookUpAccount(id)`	`true`	`ac=home.findbyPrimaryKey(id);`
t_{20}	*accountAvailable*	*accountNotAvailable*	`removeAccount()`	`ac.balance()==0.0`	`ac.remove();` `ac=null;`
t_{21}	*accountAvailable*	*accountAvailable*	`lookUpAccount(id)`	`true`	`ac=home.findbyPrimaryKey(id);`
t_{22}	*accountAvailable*	*accountAvailable*	`depositAccount(amount)`	`true`	`ac.deposit(amount);`
t_{23}	*accountAvailable*	*accountAvailable*	`withdrawAccount(amount)`	`true`	`ac.withdraw(amount);`

Fig. 3. Specification of the main component as a component state machine.

`createAccount()`. In this case, the *accountAvailable* state is entered. Similarly, the *accountNotAvailable* state is entered again when the account referred to as `ac` is removed using the `removeAccount()` method. Moreover, the main component provides the `lookUpAccount()` method for selecting accounts from the pool maintained by the Enterprise JavaBeans server. After the invocation of this method, the state of the main component either changes from the *accountNotAvailable* state to the *accountAvailable* state, if the account considered before has been removed, or remains in the *accountAvailable* state. Although the state does not change in the latter case, the considered account changes. Other methods provided by the `BankApp` component do not change its state. `depositAccount()` and `withdrawAccount()` can be used to change the balance of an existing account. Since the balance of the referenced account does not influence the state, these two methods do not affect the state. The *error* state can be entered in various situations. These situations can be distinguished into two groups. An error can occur either when trying to operate on an account which does not exist or when trying to remove an account which is not empty.

2.4 Implementation of the `BankApp` System

The `account` component is implemented as a *stateful session bean*[11]. The persistence of the component is ensured by the Enterprise JavaBeans server using a database via JDBC[12]. The Enterprise JavaBeans server used in the example is the BEA WebLogic Server 4.0.3, available as a trial version on the Internet[13]. However, the proposed technique is not tailored to a special technology. In the first section of this article, we have described the notion of a component only by a set of properties; no technology or existing component model has been referred to for defining the notion of a component. The only information required is the specification of the middleware.

The source code of the `account` component is available together with other necessary files on Websites of the BEA WebLogic Server[14].

3 Description of the Testing Technique

This section contains a detailed description of the testing technique. The first subsection explains a graphical representation of component-based software, which facilitates test case generation. The following subsections demonstrate the generation of test cases for the various constituents of the `BankApp` system.

[11] `http://java.sun.com/products/ejb/`

[12] `http://java.sun.com/products/jdbc/`

[13] `http://www.beasys.com/download/weblogic.html`

[14] `http://www.weblogic.com/docs/examples/ejb/basic/containerManaged/index.html`

3.1 Component-Based Software Flow Graph (CBSFG)

The basis of the proposed technique is a graphical representation of component-based software, called *component-based software flow graph* (CBSFG), visualizing information gathered from both specification and source code. After having generated this graphical representation, well known techniques for structural testing [24] can be applied on this representation to identify test cases. Thus, test cases for white- and black-box testing are determined simultaneously, without considering these strategies separately.

In our approach, test cases for black-box testing are generated according to the ideas of Hong et al. [183]. Their technique for black-box testing of classes requires a specification of the class in the form of a finite state machine, which they call *class state machine*. Hong et al. have proposed determining test cases for black-box testing of classes by associating definitions and uses of class variables according to a data flow criterion, and identifying those test cases that cover these def-use pairs. The important idea is to determine test cases for black-box testing by techniques for white-box testing.

Our first idea was to build a control flow graph on the basis of the source code and then to identify the definitions and uses of the class variables within this control flow graph. After the identification of definitions and uses, they can be associated with each other, and test cases can be generated to cover the associated def-use pairs – in exactly the same way as in the handling of conventional definitions and uses. Although this approach is feasible in theory, the identification of definitions, and especially of uses of the attributes, might be impossible. For instance, assume the guard of a transition looks like $a \leq b$. In the best case, this guard would appear in the source code as '`if a<=b ...;`'. But a programmer who is not restricted in his style of programming could transform this expression into '`if a<b { ...; if a==b ...; }`'. Even in this simple case, it is almost impossible to identify the guard. Of course, this problem could be solved by constraining the programmer to a certain style. But this solution has two shortcomings: the technique would not be applicable to existing software, and constraining programmers to a certain programming style would hinder acceptance of the technique.

To tackle this problem, we have elaborated the following solution:

1. A frame is generated for each event type occurring within transitions of CSMs of the component-based software,
2. the action part of each frame is marked with a label indicating the method implementing the action.

These steps are explained using the CSM of the main component in Fig. 3. During the first step, each transition $t = (source, target, event, guard, action)$ is transformed to a nested `if-then-else` construct:

```
if (predicate(source)) {   // state
  if (guard) {   // guard
    action;   // action
```

```
  }
  else throw new ErrorStateException();
}
else throw new ErrorStateException();
```

`predicate(source)` refers to the predicate of the source state, i.e., the predicate on component variables defining the occurrence of state *source*. For instance, `predicate(accountNotAvailable)` is `ac==null`.

After transforming each transition to a frame, frames of transitions having the same event type are combined. For instance, transition t_9 and transition t_{12} in Fig. 2 share the event `create()`. Their frames can be merged to the following frame:

```
createSpec(id, balance) {
  if (predicate(capacityAvailable)) {
    if (!p.containsKey(id) && p.size()<capacity-1) {
      ac=new Account(id, balance);
      p.put(id, ac);
    }
    else
      if (!p.containsKey(id) && p.size()==capacity-1) {
        ac=new Account(id, balance);
        p.put(id, ac);
      }
      else throw new ErrorStateException();
  }
  else throw new ErrorStateException();
}
```

Having represented each event type in this way, the identification of definitions and uses during a transition is trivial due to the simple and predefined structure of a frame. After the identification, test cases covering the identified def-use pairs can be generated. Note that we do not validate the frame. Test cases determined in this way also cover statements within the source code representing the definition and the use identified within the frame. The reason is that the predicate statements '`if (predicate(source)) ...`' and '`if (guard) ...`' act as a filter. They filter exactly those inputs which execute only the corresponding statements in the source code. Generally, a definition is tested by a use by executing first the definition and then the use. It is not necessary to know which statements represent the definition and the use. It is only important to ensure that both the definition and the use are executed in a certain order and that the variable is not redefined before the use occurs.

The second step consists of adding a label to the action parts of the frames, indicating the method implementing the event type. Note that this is where the integration of the white- and the black-box approach takes place. For instance, during the second step, the action parts of the frame of method

`create()` are augmented with a label '`implementing method: create()`' which refers to the appropriate method:

```
ac=new Account(id, balance);
p.put(id, ac);
// implementing method: create()
```

A frame can be generated for the entire component-based software under test by repeating this procedure for every event type, including also those of the components. After generating the frames for each event type, a control flow graph showing the overall structure of the component-based software can be generated. The CBSFG of the `BankApp` system is shown in Fig. 4.

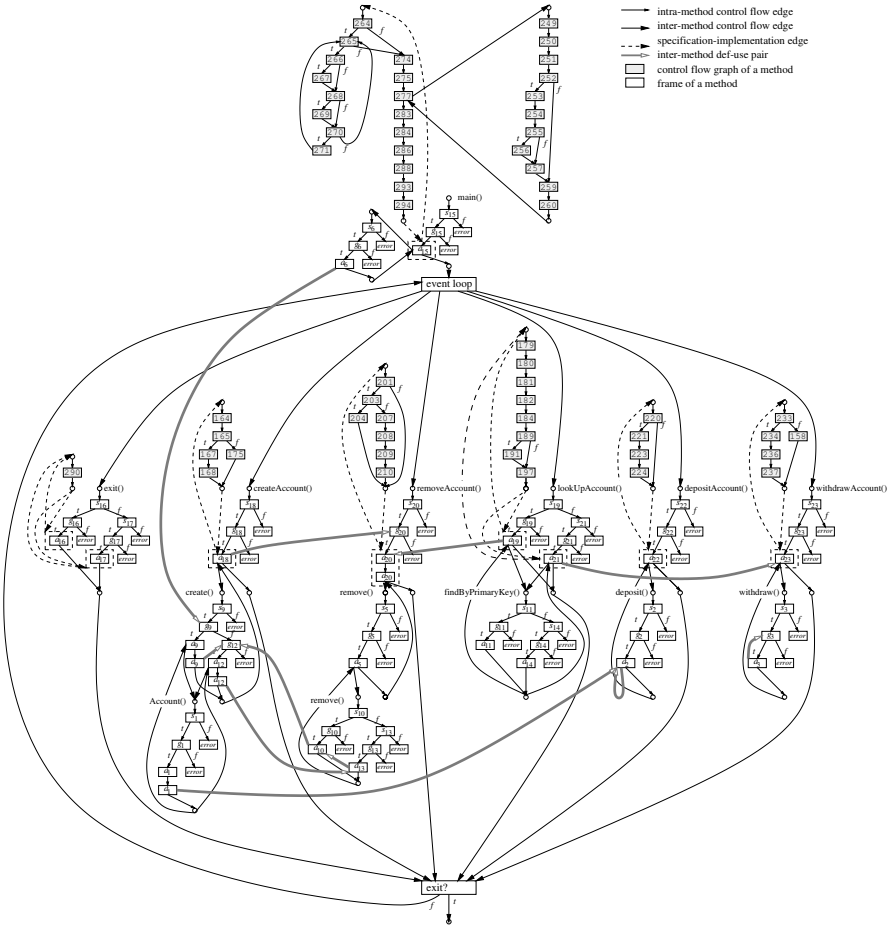


Fig. 4. Component-based software flow graph of `BankApp`.

As explained before, a CBSFG is a directed graph visualizing both control and data flow within component-based software. Each method of the main component is represented by two subgraphs. One of these subgraphs represents control flow of the frame generated on the basis of the specification, whereas the other represents control flow determined using the source code of the method. Contrary to a method within the main application, a method in another component is represented by a flow graph visualizing only its frame. The reason is obvious: a control flow graph of a method can only be built if its source code is available.

These subgraphs are interlinked with each other by control and data flow edges. Three types of control and data flow edges can be distinguished:

Intra-method control and data flow edges. Intra-method control and data flow edges visualize control and data dependencies within a single subgraph. For instance, an intra-method data flow edge connects a node representing a definition of a variable with another node representing a use in the same method. Intra-method control flow edges are drawn as thin arrows, whereas intra-method data flow edges have been omitted in Fig. 4.

Inter-method control and data flow edges. Edges of this type model control and data flow between subgraphs of the same type. For instance, an invocation of a method within another is modeled by an inter-method control flow edge leading from the node representing the invoking statement in the first method to the node representing the entry statement of the second method. Similarly, an inter-method control flow edge also models triggering an event of a CSM within the action part of a transition of another CSM. Inter-method control flow edges connect both control flow graphs of methods with each other and control flow graphs of frames with each other. Contrary to inter-method control flow edges, inter-method data flow edges connect only nodes within control flow graphs of frames. As stated above, the objective of frames is to ease identifying and associating definitions and uses of component variables. Thus, inter-method data flow edges are not required for method control flow graphs. Inter-method edges are shown by bold gray arrows in Fig. 4. Note that inter-method data flow edges which do not represent def-use pairs are omitted.

Specification-implementation edges. These type of edges visualize the connection between specification and source code by connecting the two subgraphs of the main component methods. Thus, a specification-implementation edge leads from the node representing the action within a frame to the node representing the entry node of the method referred to by the label added during the second step of the frame generation. Specification-implementation edges are drawn as dashed arrows in Fig. 4.

In Fig. 4, statements are represented by rectangles which are interlinked to each other by control and data flow edges. The number of outgoing control flow edges can be either one or two, depending on the statement represented

by a node. A node representing a predicate statement has two outgoing edges, labeled *t(rue)* and *f(alse)*, to indicate the path which is to be taken, whereas all other nodes representing other statements have only one outgoing control flow edge[15]. The number of outgoing data flow edges can vary according to the number of references to the variable defined in the node.

The graph in Fig. 4 also possesses two special nodes, labeled *event loop* and *exit?*. These two nodes take into account the event-driven nature of the example used.

3.2 Generating Test Cases for the Main Component

Our approach combines white- and black-box strategies by using a single technique for test case generation. White-box testing is conducted to test source code of individual methods, whereas black-box testing aims at testing the state-dependent behavior of a component that is tested by validating data flow among methods. Thus, test cases for black-box testing consist of a sequence of method invocations, whereas test cases for white-box testing consist of only one method invocation[16].

In the remainder of this chapter, only a data flow criterion is used for simplicity. However, the proposed technique is not restricted to a data flow criterion. Several criteria, including control flow criteria, can be used. As a first step for test case generation, definitions and uses of variables have to be identified and associated with each other. Associating definitions and uses of local variables within methods can be carried out only by considering their source code. In contrast with associating local definitions and uses, associating definitions and uses of component variables has to take into account the possible method sequences defined by the appropriate CSM. A definition and a use can be associated with each other only if the method including the use can be invoked after an invocation of the method including the definition. It is important to ensure that the component variable is not redefined.

For instance, the definition of variable `ac` within the action a_{19} of transition t_{19} cannot be associated with the use of the variable within guard g_{19}, since this would imply invoking `removeAccount()` two times to ensure the correct order of definition and use. But `removeAccount()` is not defined for state *accountNotAvailable*; thus, the main component would enter the *error* state after the second invocation. However, the definition within action a_{18} can be tested by the use in guard g_{19} by invoking (`createAccount()`, `removeAccount()`).

Figure 4 shows the def-use pairs within the main component induced by the *all-definitions* criterion [24]. These def-use pairs are (a_{18}, g_{20}),

[15]Note that a `switch` statement in C or Java can be transformed into a nested `if-then-else` construct.

[16]In some cases, initialization of the state might require invocation of a sequence prior to the invocation of the specific method.

(a_{19}, a_{20}), and (a_{21}, a_{23}). These def-use pairs can be covered by the following test cases: (`main()`, `createAccount()`, `removeAccount()`, `exit()`), (`main()`, `createAccount()`, `lookUpAccount()`, `removeAccount()`, `exit()`) and (`main()`, `createAccount()`, `lookUpAccount()`, `withdrawAccount()`, `exit()`).

3.3 Generating Test Cases for the `account` Component

Since source code for the `account` component is not available, white-box testing cannot be carried out. Black-box testing is performed in exactly the same way as in the case of the main component. Definitions and uses within control flow graphs of frames are associated with each other according to the all-definitions criterion, and test cases are generated covering those def-use pairs.

However, there is one significant difference in test case generation between the main component and the `account` component. Since the `account` component is used in the context of the main component, some of its valid method sequences cannot be tested. The `account` component is invoked through the main component, which uses a subset of the functionality the `account` component provides. Thus, some def-use pairs cannot be covered in this context, although such a data flow might occur in another context. However, we do not need to test functionality which is not required in a specific context [350, 422, 423].

For instance, although (`deposit(), balance()`) is a valid test case for testing the definition of component variable `balance` within action a_2, this test case is not valid in the current context because the `BankApp` system does not provide a functionality to check account balances. Thus, we do not need to test the interaction of methods `deposit()` and `balance()`. However, since `balance()` is invoked by the main component within guard g_{19}, we cannot completely omit this method.

Taking into account the possible method sequences within the `BankApp` system, the following def-use pairs have to be covered in order to fulfill the *all-definitions* criterion: (a_1, a_2), (a_2, a_2), and (a_3, g_3). These def-use pairs can be tested by the following test case sequences: (`main()`, `createAccount()`, `depositAccount()`, `exit()`), (`main()`, `createAccount()`, `depositAccount()`, `depositAccount()`, `exit()`) and (`main()`, `createAccount()`, `depositAccount()`, `withdrawAccount()`, `withdrawAccount()`, `exit()`).

3.4 Generating Test Cases for the Middleware

Test case generation for the middleware component is carried out in exactly the same way as in the case of the `account` component. Test cases have to fulfill the *all-definitions* criterion, have to be valid method sequences of the

middleware component and have to be possible in the context of the `BankApp` system.

In order to test the middleware appropriately, the following def-use pairs have to be covered by test cases: (a_6, a_{18}), (a_9, g_{12}), (a_{12}, a_{13}), (a_{13}, a_{10}), and (a_{10}, g_{12}). Test case sequences executing these def-use pairs are (`main()`, `createAccount()`, `exit()`), (`main()`, `createAccount()`, `createAccount()`, `exit()`), (`main()`, `createAccount()`, `removeAccount()`, `exit()`), (`main()`, `createAccount()`, `removeAccount()`, `createAccount()`, `exit()`) and (`main()`, `createAccount()`, `createAccount()`, `removeAccount()`, `removeAccount()`, `exit()`).

3.5 Generating Test Cases for the Integration Test

Another important task is testing the interaction of the main component with the `account` component and the middleware component. The main component is tested with the same test cases that have been generated for its black-box testing. The main difference between black-box testing of the main component and integration testing is that in the latter the same test cases are repeated for every possible state of the components [55].

The integration test of the main component with the `account` component does not require new test cases because the `account` component possesses only one abstract state. This state is entered after initializing the component, done in each test case by invoking `createAccount()`.

In contrast with the `account` component, the middleware component requires generating new test cases. The middleware component can enter two abstract states, namely *capacityAvailable* and *capacityLimit*. The integration test can be performed with the test cases used for testing the main component; the only difference is in the parameters passed. To test the *capacityAvailable* state, the `capacity` parameter has to be higher than 1. For the other cases, the `capacity` parameter has to be set exactly to 1 in order to enter the *capacityLimit* state by creating one account.

3.6 Regression Test based on CBSTDs

CBSFGs can also be used for identifying test cases for regression testing on the basis of the technique of Rothermel and Harrold [351, 352]. They have proposed comparing two successive versions of an object-oriented program with respect to their graphical representations called *class control flow graph* (CCFG). This comparison identifies those statements which have to be tested due to modifications made during the last correction, or due to modified control and data dependencies from modified statements. Using the same approach, a CBSFG is generated after a modification, and is compared with the CBSFG of the prior version of the component-based software. By comparing these two graphs, those statements and def-use pairs can be identified that

have been modified or that are affected by modifications made to other statements. The modified version needs to be tested only with respect to these changed or affected statements or def-use pairs. Since CCFG and CBSFG have a similar structure, their algorithms can also be used, with some adjustments, for selecting test cases for component-based software on the basis of CBSFGs. Note that CBSFGs also permit selection of test cases for black-box testing and integration testing, whereas CCFGs do not.

4 Conclusions

In this chapter, we outlined requirements that have to be addressed by testing techniques for component-based software. After discussing these requirements, we have described a graphical representation of component-based software called *component-based software flow graph* (CBSFG) facilitating test case generation. The generated test cases cover the important features of component-based software to be tested: white- and black-box testing of the main component, black-box testing of other components (including the middleware component), and integration testing.

The applicability of this approach has been demonstrated with an example, a system for conducting deposits and withdrawals a bank account. Enterprise JavaBean technology has been used as middleware.

Our aim during the development of this technique was its automatability. Every step during testing, except test case generation, which is in fact a very hard problem, can be carried out automatically.