SUMMARY Software programs often include many defects that are not easy to detect because of the developers' mistakes, misunderstandings caused by the inadequate definition of requirements, and the complexity of the implementation. Due to the different skill levels of the testers, the significant increase in testing person-hours interferes with the progress of development projects. Therefore, it is desirable for any inexperienced developer to identify the cause of the defects. Model checking has been favored as a technique to improve the reliability earlier in the software development process. In this paper, we propose a verification method in which a Java source code control sequence is converted into finite automata in order to detect the cause of defects by using the model-checking tool UPPAAL, which has an exhaustive checking mechanism. We also propose a tool implemented by an Eclipse plug-in to assist general developers who have little knowledge of the model-checking tool. Because source code is generally complicated and large, the tool provides a step-wise verification mechanism based on the functional structure of the code and makes it easy to verify the business rules in the specification documents by adding a user-defined specification-based model to the source code model.

key words: model checking, UPPAAL, Eclipse

1. Introduction

Software programs often include many defects that are not easy to detect because of the developers' mistakes, misunderstandings caused by the inadequate definition of requirements, and the complexity of the implementation. When failures are found after the service has been deployed, a significant number of person-hours are often required to detect the defects and the causes. If the experienced developers always attend to these tasks, they will not be able to attend to other important tasks, thereby impeding the progress of projects. Thus, any general developer must be able to reliably identify the cause of defects, even if he/she is not a specialist. Searching for the correct location of the defects in the source code is difficult owing to the complicated control structure.

Model checking has been favored as a technique to improve the reliability earlier in the software development process. We propose a verification method to identify the correct location of the defects by using the model-checking tool UPPAAL. We also developed a support tool implemented as an Eclipse plug-in, which a non-specialist in model-checking can use to investigate a control flow thoroughly and to easily identify the cause of the specified failure.

The rest of the paper is organized as follows. Section 2 discusses how a developer who is not a specialist in model-checking can use a model-checking tool. Section 3 explains how to generate a system model so that developers can easily verify specified properties using the verification support tool Source2UPPAAL. Section 4 demonstrates the effectiveness of our method by using it to verify business rules. Section 5 discusses the applicability of our method to practical software. Finally, Sect. 6 discusses related works.

2. Problems in Model Checking

Figure 1 shows our verification method and the support tool Source2UPPAAL. A developer who is not a specialist in model-checking can easily detect the defect in their code by using a model-checking tool as the verification method.

2.1 UPPAAL

The model-checking tool UPPAAL [1] uses temporal logic. A target system is modeled as a network of automata that is extended with integer variables, structured data types, user defined functions, and channel synchronization. We can define a system model with query expressions that specify properties to be checked. When the specified properties are not satisfied, the tool gives a counter-example showing what may happen. The simulator helps us to detect the cause of defects by tracing the process in which the counter-example happens. Model checking is a technique that automatically verifies a model by checking all paths exhaustively, so that it can detect properties that developers often miss. A query...
expression consists of a path formula and several state formulas. Path formulae can define such properties as Reachability, Safety, and Liveness. Reachability means that the specified state will be reached eventually. Safety means that properties will not have undesirable values. Liveness means that expected properties will eventually happen. State formulae need to be defined by expressions related to several process IDs or variables in the state. However, because the path formulae and state formulae must be defined by items that are used in the system model, developers generally have difficulty defining the appropriate system model and query expressions.

2.2 Problems with Building a Verification Method

To verify the faulty programs, a system model is automatically generated by converting the control sequences of the target source code into finite automata. To perform the verification for developers who have little knowledge of model checking, two problems need to be solved.

First, the complicated and large source code needs to be well abstracted to avoid “state explosion,” where the number of states become unmanageably large, so that the developers can select the appropriate candidate functions to verify, based on their knowledge of the specifications.

Second, a non-specialist developer in model-checking must be able to verify the specified properties using the converted model. In this paper, we assume that the target source code is written in Java.

2.3 Generation of System Model from Source Code

A UPPAAL model consists of several locations and transition arrows between them, as shown in Fig. 2. A location expresses a state of a system, and a transition arrow expresses a change in the states. A transition consists of the following four labels. Guard provides the different conditions of the change. Update is an expression of a sequential process during the change. Select provides a state variable with a non-deterministic value by selecting from a list of values. Synchronization provides channels among several automata. In Fig. 2, START, LOC1, and LOC2 are the names of the locations, “i1=0” and “i1>0” are Guard expressions, and “flg=true” and “flg=false” represent Update expressions.

To solve the first problem of abstracting the code, we propose a phased conversion from the Java source code to a UPPAAL model, based on the method calls, including all user-defined and standard APIs. Not all methods have to be converted into the UPPAAL model at one time; tester can convert one or more methods selectively.

First, the converted UPPAAL model is verified based on the expressions converted from the original source code within the same level. If a tester cannot detect the defects, he/she can include the selected methods in the next iteration.

However, not all symbols in the source code are converted into symbols in the UPPAAL model. In UPPAAL model, states are distinguished each other by using locations and global variables with type Integer and Boolean. Thus, we define the conversion rules as follows.

- Each statement in the source code is converted to a location in the UPPAAL model.
- Each declared variable with type Integer and Boolean is converted to a global variable in the UPPAAL model.
- Each expression that assigns its resolved value to an Integer or Boolean variable is converted into an Update expression in the UPPAAL model.
- Each method call is manually assigned a single value or several non-deterministic values by using a Select expression in the UPPAAL model, so that each value expresses a possible state of the location.

Each Guard, which decides the condition of a transition from one location to another, is expressed by the abovementioned variables. Such conversion creates a system model that reproduces the behavior of the source code, including the failures.

2.4 Generation of Query Expression from Failures

Generally, a tester can recognize failures during operations, including observable phenomena, such as an infinite loop, as well as the difference between the behavior of the program and the specifications. These failures are caused by undesirable values of the properties in the running system. UPPAAL can verify the Safety property by using a path formula to ensure that properties will not have undesirable values.

If a model identifies the properties with undesirable values, it can decide which state formula to use, so that query expressions are automatically generated. We have shown an example of detecting a defect caused by an infinite loop [2]. In this case, we defined a model in which the same process is repeated more than the expected number of times. The user-defined model is integrated into a selected set of doubtful control flows by using Synchronization in UPPAAL. Consequently, we could detect the cause of the infinite loop by verifying the Safety property, which ensures that an infinite loop will not happen.

In this case, a point where a user-defined model will be inserted is decided by a loop structure and a label of the expected defect. This is independent of the application specifications. Because a state formula is generally defined by expressions dependent on the application itself, a tester needs to recognize the correspondence of the target program expressions to the specifications.

Generally, programs are developed based on the spec-
ification documents written in natural language. We assume that a tester can understand the business rules specified in the document. This understanding will guide a tester in deciding the appropriate correspondence of a target system model to the query expressions by defining the business rules as a decision table.

2.5 Support for Non-specialist in Model-Checking

When failures occur during operations, a significant number of person-hours are required to detect the defects and the causes. The main difference between inexperienced and experienced developers may be that the former is less capable than the latter in developing a suitable hypothesis to detect a cause of the failure. A model-checking tool is a promising approach to help inexperienced developers to detect the cause of failure, because it provides an automatic mechanism that performs formal exhaustive verification.

However, two problems need to be solved for many developers who have little knowledge of model checking to detect defects in a program using a tool. First, a developer needs to be able to select the appropriate candidate methods to be verified based on his/her own knowledge of the specifications. Secondly, he/she also needs to define the appropriate query expressions to verify the properties.

A phased conversion helps a tester even if he/she has little knowledge of model checking, because several methods implement use cases and become the main functions of the system. Therefore, a tester gets useful hints to help recognize the correspondence between the programs and the specifications. Moreover, the core business rules could make it easy to recognize the correspondence between their descriptions and the method names.

If we can model the properties with undesirable values, several automata can be integrated using Synchronization. A tester can get the query expressions in UPPAAL automatically.

Therefore, we propose a verification support tool Source2UPPAAL with the following functions:

- It provides a stepwise checking mechanism based on the method structure.
- A tester can set a list of values for non-determinant execution at the selected location to a statement or a method call, if necessary.
- A tester can define a model for the selected method based on the original specifications, if necessary. Consequently, we can verify the specifications by integrating it into the UPPAAL model.

3. Verification Method and Source2UPPAAL

3.1 Conversion Process from Source Code to UPPAAL

A target Java source code consists of several packages, including several classes. Based on the conversion rules mentioned in Sect. 2.3, a system model is generated semi-automatically as described below. Figure 3 shows the conversion process, and we explain it using an example source code shown in Fig. 4.

First, the source code shown in Fig. 4 is translated into Abstract Syntax Tree (AST) using the AST Parser [3]. Figure 5 shows a graph generated by the AST in which a statement is converted into a node of AST. Each node is comprised of a method and an if-statement with a start node and an end node.

The graph is converted into a UPPAAL model by mapping a node in the graph to a location in the UPPAAL model, as shown in Fig. 6. The first two field declarations such as fflg and val are converted into global variables in the UPPAAL model. Because these variables are of Integer or Boolean types, they are migrated unchanged in Update in the UP-
PAAL model. On the other hand, in statements such as “char ch1 = ‘A’” and “double d = 0.0,” variables cannot be used as global variables in the UPPAAL model. However, assignment statements, such as “ch1 = line1.charAt(1);”, and infix expressions, such as “ch1==’B’,” are abstracted into a state of true or false in the UPPAAL model. Therefore, we can manually set these statements to non-deterministic values. As a result, these statements are converted into the Select expression in the UPPAAL model.

Because such a statement as “System.out.println("Even number");” has no effect on the application logic, we can set the value true to the statement.

A user-defined method, such as chkVal, is converted into the UPPAAL model if a tester suspects that a defect occurs within the method. At the first verification step, it is set to non-deterministic values, based on the return value type. In the case of an API method such as valueOf in the Integer class, the expression is set to a list of Integer values.

However, the defects are often dependent on the abstracted state. With user-defined methods, we can get the unfolded UPPAAL model and integrate it into the original model using Synchronization. With API methods, we need to define the model for the behavior based on the specifications of the APIs.

3.2 Verification Support Tool: Source2UPPAAL

To minimize the manual tasks of a tester, we implemented Source2UPPAAL as an Eclipse plug-in [4] with the following functions.

- Select the method to be verified from the method-call tree structure using a graphical user interface (GUI).
- Select a list of values for the non-determinant execution of a statement or a method call, using the GUI.
- Register a user-defined model, based on the specifications of APIs or business rules, and set it to the selected method.
- Generate a UPPAAL system model and a set of query expressions, based on the abovementioned replacement, specification-based model, and AST data.

Replacement tasks can be easily initiated using drag-and-drop. Figure 7 shows a case of selecting a method from the tree structure of the source code shown in Fig. 4. All the statements in the specified method are displayed as the right side of Fig. 7 shows. A tester can choose a method to be verified from the tree. The left side of Fig. 7 shows the selected expressions to be converted into the UPPAAL model.

Figure 8 shows a case of selecting a list of values for the non-determinant execution of a statement or a method call. We can choose a value with type Boolean or Integer from candidate patterns, such as “true,” “false,” “true or false,” and “arbitrary numerical values.” The selected values define the Select property in the UPPAAL model, and new variable holding the value is defined as a global variable.

After the UPPAAL model in this state is checked, we suppose that a defect is found in checkVal. As the inside of checkVal needs to be checked, the definition of checkVal is unfolded by selecting a method insted of values for the non-determinant execution as Fig. 9 shows.

To register a specification-based model based on business rules, a file of the UPPAAL model must be configured in the same Eclipse project as the verification support tool. Then, it is displayed on a tool with other source code as Fig. 7 shows. Using drag-and-drop, the models are integrated into the base model by generating the Synchronization property of the transition between them.
3.3 Verification by User-Defined Specification-Based Model

Generally, programs are developed based on the specifications documents written in natural language. We explain how to verify code using a user-defined model based on the business rules in the specifications. Figure 10 shows the process.

The program to be verified was created based on the specifications. We can assume that a tester can understand the business rules in the specifications written in natural language, and that he/she can define a decision table. A decision table [5] is well-known as a precise yet compact way to model complicated business logic caused by combinations of inputs.

First, a tester extracts actions with the conditions and the result states from the documents of the business rules. These actions correspond to methods in the program to be verified. By these actions, we can determine the correspondence between the source code model and the user-defined specification-based model. Moreover, Source2UPPAAL helps a tester to select corrects methods by comparing the functionality and the action names. On the other hand, the combinations of the result states can represent some unexpected properties that will have the undesirable values, so that we can get query expressions for Safety properties automatically.

Next, a tester makes several types decision tables in stages. Finally, a tester makes decision tables which correspond to all result states and all actions. These decision tables can be automatically converted into the UPPAAL models which have Synchronization properties between these models.

There are two kinds of UPPAAL models. The first kind of model specifies a property to be satisfied by actions with the conditions in the decision tables. That is, it is the specification of the target business function. The second kind of model has a role that connects the first model with the source code model. These models are registered with Source2UPPAAL and the second kind of model will be set on the corresponding method in the source code using drag-and-drop operation. The first kind of model will be integrated with the source code model by the Synchronization mechanism. As a result, we can check the specified properties against the source code. Details will be shown in Sect. 4.

4. Case Study

4.1 Outline of the Case Study

As a case study, we applied our method to a credit management program used by the distribution industry for sales orders. If the company receives a new order that would exceed the customer’s credit limit, which is stored in the credit management master, the sales system program blocks the order. We detected defects of sample program which is defined according to the specification used in actual operations of a real company.

4.2 Definitions of Terms

We define the terms used in this case study as follows:

(1) Credit Limit
Credit Limit is the maximum amount that a company accepts as receivables. The credit limit for each customer is registered in the credit management master.

(2) Credit State
The magnitude relation between Credit Limit and the sum total of receivables is called Credit State. When a customer’s receivables is greater than their Credit Limit, Credit State is set to OVER; otherwise, it is set to UNDER.

(3) Temporary Credit Check
Temporary Credit Check is the process of confirming whether the sum total of the current receivables and the amount of the sales order exceeds Credit Limit. If the sum total exceeds Credit Limit, the Temporary Credit Check shows that Credit State is OVER; otherwise, it shows UNDER. If Credit Limit is raised, Credit State is set to UNDER again.

(4) Credit Block Flag
When Credit State is OVER, the block flag is set to ON for the applicable customer in the credit management master. This flag is called Credit Block Flag.

(5) New Sales Order Refusal State
This is the state where Credit State is OVER and Credit Block Flag is ON. In this state, new sales orders cannot be created for the customer, until the state changes.

4.3 Example Scenario

The main role of a credit management program is to control the creation of a sales order matching to the rules on a combination of multiple conditions. Figure 11 shows an ex-
ample scenario of credit block processing which follows the rules. Voucher 001 is registered normally. However, Credit Limit was exceeded by Voucher 002; therefore, Voucher 003 is blocked. After Credit Limit is raised, Voucher 004 is processed.

4.4 Specifications of Program

The sample specifications of the program are as follows:

1. In the New Sales Order Refusal State, a voucher is not processed, and an error message is shown.
2. If not in the New Sales Order Refusal State and the result of Temporary Credit Check is OVER, a voucher is processed and an error message is shown. Credit Block Flag is set to ON.
3. If not in the New Sales Order Refusal State and the result of Temporary Credit Check is UNDER, a voucher is processed. If Credit Block Flag is ON, it turns OFF.
4. If in the New Sales Order Refusal State, all creation of sales orders are denied using a dedicated verification method. If Credit State is OVER and Credit Block Flag is OFF, the method returns true; otherwise, it returns false. The amount of money does not affect the determination of Credit State, because the amount of money has not been saved yet.
5. Error message are displayed using an error message display method.
6. The Temporary Credit Check method confirms whether the sum total of existing receivables and the amount of the order is over Credit Limit. If Credit Limit is OVER, the method returns false. If Credit Limit is UNDER, the method returns true.
7. The Credit Block Flag Change method sets Credit Block Flag to ON, if the argument is true; otherwise, OFF.
8. Sales orders are created using a sales order method. The customer’s receivables increase during order verification.
9. The Credit Block Flag Change method sets Credit Block Flag to ON, if the argument is true; otherwise, OFF.

4.5 Creating the Decision Table

As mentioned in Sect. 3.3, first a tester creates three work tables by extracting the Condition, the Action, and the Result from the specifications and the definitions of terms. Specifications (1) to (3) define the main part of the program, and Specifications (4) to (9) define a method corresponding to a use case, from which the method is composed.

Table 1 shows the Condition extracted from the specifications and includes the names of the actions related to the condition, the possible values, and the numbers of the associated specifications. Likewise, Table 2 shows the Action extracted from the specifications and includes the names of the conditions, the possible values, and the numbers of the associated specifications. By these actions, the tester can determine the correspondence between the source code model and the user-defined specification-based model.

Table 3 shows the Result extracted from the specifications and includes the names of the actions related to the condition, the possible values, and the numbers of the associated specifications. Likewise, Table 2 shows the Action extracted from the specifications and includes the names of the conditions, the possible values, and the numbers of the associated specifications. By these actions, the tester can determine the correspondence between the source code model and the user-defined specification-based model.

Table 3 shows the Result extracted from the specifications and includes the names of the states and the possible values. The combinations of the result states can represent some unexpected properties that will have the undesirable values, so that we can get query expressions for Safety properties automatically.

The decision table is created based on these three work tables. For example, we explain the creation of the decision table using Specification (2). The work table of Specifica-
Table 4 Work table of Specification (2).

<table>
<thead>
<tr>
<th>No</th>
<th>Item</th>
<th>Classification</th>
<th>Specified Value</th>
<th>Related Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>New Sales Order Refusal State</td>
<td>Branch Condition</td>
<td>FALSE</td>
<td>Related Specification</td>
</tr>
<tr>
<td>02</td>
<td>Result of Temporary Credit Check</td>
<td>Action</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>03</td>
<td>Credit Block Flag</td>
<td>Branch Condition</td>
<td>OFF</td>
<td>Related Specification</td>
</tr>
<tr>
<td>04</td>
<td>Credit State</td>
<td>Branch Condition</td>
<td>UNDER</td>
<td>Related Specification</td>
</tr>
</tbody>
</table>

Table 5 Decision table of Credit Management.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Create Sales Order</td>
<td>Action</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>02</td>
<td>Error Message Display</td>
<td>Action</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>03</td>
<td>Temporary Credit Check</td>
<td>Action</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>04</td>
<td>New Sales Order Refusal State Check</td>
<td>Action</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>05</td>
<td>Credit Block Flag Check</td>
<td>Action</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>06</td>
<td>Credit State</td>
<td>Condition</td>
<td>OVER</td>
<td>UNDER</td>
<td>UNDER</td>
<td>UNDER</td>
</tr>
<tr>
<td>07</td>
<td>Credit Block Flag</td>
<td>Condition</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>08</td>
<td>Credit State</td>
<td>Condition</td>
<td>OVER</td>
<td>UNDER</td>
<td>UNDER</td>
<td>UNDER</td>
</tr>
<tr>
<td>09</td>
<td>Credit Block Flag</td>
<td>Condition</td>
<td>UNDER</td>
<td>UNDER</td>
<td>UNDER</td>
<td>UNDER</td>
</tr>
<tr>
<td>10</td>
<td>Credit Block Flag</td>
<td>Result</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>11</td>
<td>Credit State</td>
<td>Result</td>
<td>OVER</td>
<td>UNDER</td>
<td>UNDER</td>
<td>UNDER</td>
</tr>
</tbody>
</table>

If Action is related to the selected Condition, Action is judged to be also related to Specification (2). Because New Sales Order Refusal State consists of Credit State and Credit Block Flag, the combination of Credit State and Credit Block Flag is determined automatically. Depending on the item, the possible value can be obtained from specifications or from the definition of terms.

We also applied this procedure to Specifications (1) and (3). The three work tables were combined to create the decision table, shown in Table 5. The defects that our method should catch are in two Result: Credit Block Flag of Item No.11, and Credit State of Item No.12.

In these states, Credit Block Flag should turn ON and Credit State should be set to OVER, or Credit Block Flag should turn OFF and two patterns of Credit State should be set to UNDER. We also modeled the subroutine separately. However, the error message display method is not related to state changes; therefore, it is not included in the model.

4.6 Creating Verification Model (User-Defined Specification-Based Model)

To define the specified properties on Result states, a tester select the related rows from the decision table shown in Table 5.

Table 6 shows the selected decision table that presents the state transitions of Credit Block Flag. The created model is shown in Fig. 12.

By reading the decision table, we understand that Credit Block Flag has two states, ON/OFF. Because these states toggle to each other, the transition is two-way. A transition is activated by an operation that changes a state. The transition is triggered by the execution of the Credit Block Flag Change method in the model corresponding to the action.

Fig. 12 Model of Credit Block Flag.

The synchronization channel name is arbitrary. For example, by receiving “Credit_Block_Flag Change_TRUE” message, a state transition is started. The sending side must wait until the receiving side has completed processing. In order to return processing to the sending side, the synchronization message “RETURN!” is sent.

To communicate the state of other models to this model, the state is stored in the variable vBlock, which can be ON (true) and OFF (false). Because the state is not known when verification starts, it is initialized using nondeterminacy at location START, and later changes to ON or OFF. The initial state is stored in the variable vInitBlock.

The model for this table is shown in Fig. 12.

The decision table that represents state transition of a Credit State is shown in Table 7. The created model is shown in Fig. 13. The New Sales Order Refusal State verification method is not described in a model to prevent a change of state. The Temporary Credit Check method is described in a model to change Credit State to UNDER or OVER based on the amount of the sales order. Because the sales order create method can also change Credit State from UNDER to OVER, that method is described.

The model for this table is shown in Fig. 13. Credit State could be one of two states: UNDER and OVER. The initial state is set up using non-
Table 7  Decision table of Credit State.

<table>
<thead>
<tr>
<th>No.</th>
<th>Item</th>
<th>Classification</th>
<th>Specification</th>
<th>Specification</th>
<th>Specification</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create Sale Order</td>
<td>Action</td>
<td>Not Execute</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>01</td>
<td>Temporary Credit Check</td>
<td>Action</td>
<td>Not Eligible</td>
<td>Eligible</td>
<td>Eligible</td>
<td>Eligible</td>
</tr>
<tr>
<td>02</td>
<td>New Sales Order Eligible Check</td>
<td>Action</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>Recap of Temporary Credit Check</td>
<td>Action</td>
<td>Not Eligible</td>
<td>Eligible</td>
<td>UNDER</td>
<td>UNDER</td>
</tr>
<tr>
<td>04</td>
<td>Credit State</td>
<td>Result</td>
<td>OVER</td>
<td>UNDER</td>
<td>UNDER</td>
<td>UNDER</td>
</tr>
</tbody>
</table>

The transition UNDER -> OVER could happen after receiving the synchronous processing message “Create_Sales_Order_call?”, if the sum total of the amount of the sales order and the existing receivables exceed Credit Limit and Credit State is set to OVER.

We have defined a model that presents a state. However, these user-defined specification-based models are not related to a source code model. Therefore, we created a model that is connected to a source code model.

If a model is connected to a source code model, it can be synchronously called from the source code model to start a transition. The execution result, if any, is acquired and processing returns to the source code model.

For example, we modeled the Credit Block Flag Change method by creating a decision table based on Specification (9). The model of the Credit Block Flag Change method (Fig. 14) was created from the decision table in Table 8.

When a synchronous call from a source code model (Method_Block_Flag_call_call?) is received, the transition (START -> Change) is started. The conditional branch process is changed based on the value of the argument passed from the source code model.

Table 8  Decision table of the Credit Block Flag Change method.

<table>
<thead>
<tr>
<th>No.</th>
<th>Item</th>
<th>Classification</th>
<th>Specification</th>
<th>Specification</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Credit Block Flag Change Method</td>
<td>Action</td>
<td>Execute</td>
<td>Execute</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
| 01  | Credit Block Flag | Result | ON | OFF | }

The user-defined specification-based model is related to the method that corresponds to source code by Source2UPPAAL. Therefore, the user-defined specification-based model is called from the source code model. Sending an argument and receiving a return value also become possible.

In Fig. 12 and Fig. 13, a global time variable named “t” is used to match the timing of running the source code model and initializing the Result states.

4.7 Model Checking

As mentioned in Sect. 3.2, a tester selects a list of values for the non-determinant execution of an API method call using Source2UPPAAL. Moreover, the specification-based models mentioned in Sect. 4.6 are registered to Source2UPPAAL, and the models are integrated into the base model by drag-and-drop operation as Fig. 15 shows. As a result, a tester can start the model checking as follows.

Model checking verifies whether a system will be in an unexpected state. After the program is performed, the state will be one of the following two cases:

- Credit Block Flag ON and Credit State OVER
- Credit Block Flag OFF and Credit State UNDER

If the state is neither of these two cases, a defect exists in the code. A query expression can be created, because the item to inspect is transposed with the location identifier or variable on the state model of Fig. 12 and Fig. 13. The query expression is as follows.

A[] not ((Initial_Credit_Block_Flag_var==true && Rst_Credit_Block_Flag.OFF)) || (Rst_Credit_State.OVER && Rst_Credit_Block_Flag.ON))

The query expression means that the state at the end of the source code is one of the following: Credit Block Flag is OFF and Credit State is UNDER, or Credit Block Flag is ON and Credit State is OVER.

The result was “Property is NOT satisfied,” which proved that a problem exists. After Credit State was set to UNDER and Credit Block Flag was set to ON, the state did not change. Even if the process started with the same initial state, it was inspected to determine whether it ends as expected. The query expression is as follows.

A[] (voidmain.END imply (Rst_Credit_State.UNDER && Rst_Credit_Block_Flag.OFF)) || (Rst_Credit_State.OVER && Rst_Credit_Block_Flag.ON))
The query expression means that, if Credit State is OVER and Credit Block Flag is ON at the initial state, then the process will never get to the state where Credit State is UNDER and Credit Block Flag is ON. The result of verification becomes “Property is NOT satisfied.” In the counter example, even when the initial state is the same, Credit Block Flag turns OFF and the process ends normally.

We look for the place where the different transition is started in two counter examples. The place can be pointed on a source code model.

When Credit Limit is raised, the state at the first stage becomes such that Credit State is UNDER and Credit Block Flag is ON. Depending on the amount of the sales order, Temporary Credit Check may be suddenly set to OVER, immediately after raising Credit Limit. In that case, the sales order was not processed at this time.

5. Discussion

In a business system development, based on business process analysis the required business flows are defined as a sequence of several business processes. A business process consists of several business functions and several system functions. We assume that a business function is the minimum unit of a business process and includes several system functions and the related manual work with them. Figure 16 shows a part of the business flow in the sales management system discussed in this paper.

“Sales Order Processing” consists of 6 business functions such as “Inventory Reservation” and “Credit Management” discussed in Sect. 4. Each business function is executed by executing one or more system functions. “Credit Management” consists of such two system functions as “Credit Limit Check in Sales Order” and “Credit Limit Amount Check”. The specification of this business function is the specification shown in Sect. 4.4. In this way, each specification of system functions that compose a business function allows us to do model-checking of the entire system.

This paper proposed a verification method of the business function specification for the source code by organizing the specification as a decision table and integrating the converted source code model and the specification-based model.

A decision table expresses the conditions that specify the business logic in a table form to organize the combinations of possible values for each condition. This table is often used as a convenient tool for black-box testing of the unit test that allows us to extract the test cases. In a similar way of conducting a testing for the entire system, a model-checking process needs to be planned the order considering the dependency between business functions in a business process.

In case of the specifications shown in Sect. 4.4, the requirements (1) to (3) are the basic specifications of “Credit Limit Check in Sales Order”. The requirements (4) to (7) are method specifications of “Credit Limit Amount Check”. The item (8) shows a pre-condition that the method is called by a sales order creation method of “Create Sales Order”. The item (9) is a method specification associated with (4). In this case, after completion of checking the business function “Create Sales Order”, the business function “Credit Management” can be checked.

As with creating test cases to a unit of each business function, a decision table creates to a unit of each business function. A business function consists of several system functions and manual operations of a user. Thus, we can say that conditions and actions to be specified are limited to conditions and actions which the user performs within a practical time. The size of a decision table is within the range that a developer can create within a practical time.

The number of decision tables increase as the number of the related business functions increase. However, the size of the individual decision table will not change for each business functions. Considering the dependencies of each business function allows us to do model-checking of the entire system.

6. Related Work

There are many researches of model checking to detect the defects of the source code. Java PathFinder [6] is a system to verify executable Java bytecode program, so as to detect the deadlock and assertion errors of the distributed application, user interfaces, etc. This system has “state explosion” problem against the complicated and large source code. Bandera [7] enables the automatic extraction of safe, compact finite-state models from Java program source code by using abstraction, slicing, and static analysis. The output model can be verified by such existing model-checking tools as SPIN [8] and SMV [9]. This approach solves the above-mentioned problem, however, a tester is required deep knowledge of model checking.

Markesan [10] inspected the NASA flight simulator developed by Pathfinder. He explained the technique of refactoring by reducing the number of states and by conducting a model inspection efficiently. Because he converted the source code to be suitable for model checking with the modification technique described in this paper, the converted source code is checked to determine whether the original
meaning is lost.

These techniques for minimizing a model are useful for us. However, they cannot perform verification on user-defined specification-based models, which define the observable phenomenon of a failure.

Beyer [11] proposed a technique to discover dead code by inserting an assertion in the C program and then checking its Reachability property. They used the model-checking tool BLAST. However, because the discovery of the dead code became the main purpose, the defects in the business logic could not be found.

Lei [12] inspected the brittleness of the application using BLAST by determining the Reachability of an assertion. It also cannot define and check the phenomenon to be analyzed.

It is important to elucidate properties that can be verified using model-checking. Several researchers have verified the business process using a model-checking tool.

Angelis et al. [13] proposed a method to verify a model by converting the code to a Java object that was described using BPEL. This method effectively verifies the business process for only BPEL users.

Dury et al. [14] discussed an experimental result of applying a model checking technology for verifying RBAC on workflows. A modeling method is an effective example of applying a model checking. However, it is not a wide-use modeling of business rules for a developer who is not a specialist in model-checking.

Roy et al. [15] conducted an empirical analysis of the relationships between error metrics and business process model-based BPMN. Without the use of model checking, the relationships are detected based on control flow errors and syntax errors. However, these works are insufficient for our purpose, because they require advanced knowledge of the tester.

Dwyer et al. [17] proposed an approach of property specification patterns. A property specification pattern describes the essential structure of some aspect of a system’s behavior and provides expressions of this behavior in a range of such common formalisms as LTL, CTL and etc. There are five basic kinds of scopes. For example, global means the entire program execution, before means the execution up to a given state/event. The scope is determined by specifying a starting and an ending state/event for the pattern. They collected 555 specifications and examined each specification and manually determined whether it matched a pattern. As the result, of the 555 example specifications, 511 (92%) matched one of patterns. The most common pattern in the sample is Response, with the next most common being Universality and its dual Absence. Together, these three patterns accounted for 80% of the sample.

The query expressions generated from a decision table in our approach correspond to most common patterns such as Response, Universality and Absence with global scope. This means that a decision table is useful to specify essential system properties formally. It is understandable that their proposed patterns are helpful to write a query expression. However, it is required to understand the meaning of the specification pattern and scope in connection with the finite automata. It is difficult for a non-expert of formal method to write a query expression in connection with finite automata correctly, because of the rigorous mathematical formalism.

Rachel et al. [18] proposed property pattern templates and the PROPEL system based the research of Dwyer et al. PROPEL aims to make the job of writing and understanding properties easier by providing templates that explicitly capture these details as options for commonly-occurring property patterns. These templates are represented using both a disciplined natural language and finite state automata, allowing the specifier to easily move between these two representations. However, difficulty in writing correct specification properties with the template still remains. Moreover, as their research does not cover all the patterns proposed by Dwyer, a developer may not be able to find an appropriate property pattern that they want to use.

Guglielmo et al. [19] proposed DDPSL (Drag and Drop Property Specification Language (PSL)) a template library and a tool which simplifies the definition of PSL formal properties by exploiting PSL-based templates. They also provide a way to reduce the effort required for formalizing complex properties. As a user can define PSL properties by dragging and dropping logical and temporal operators, he/she may write formal properties easily, even if he/she is not expert in formal methods. IBM Formal Checkers tool simulates the result rule in which some user-defined expressions combined with the selected operators. A rule corresponds to a query expression in model-checking. However, these expressions need to be written correctly by the developer.

In case of formalizing properties in model-checking of a source code, a developer not only writes a pattern of a query expression, but also he/she has to correctly write an expression in the query expression in connection with some variables or method names. It is a difficult and indispensable task to a non-specialist in model-checking. In our verification method, based on the specification written in a decision table, a query expression in connection with the source code can be automatically generated. Writing a decision table provides a verifiable correct query expression for a non-specialist in model-checking.

Salamah et al. [20] propose the Property Specification (Prospec) tool which uses patterns and scopes defined by Dwyer et al., to generate the improved formal specifications in Linear Temporal Logic (LTL) and other languages. In order to efficiently execute LTL specification, reducing the number of states is an indispensable problem. Their research can contribute to solve this problem in reducing the scale of the written LTL formal specification. However, as the scale of source code that is a target of model-checking is generally large, another measure is also required to reduce the number of states. To reduce the number of states, our verification method provides a step-wise verification mechanism based on the functional structure and setting non-deterministic values or user defined specifica-
We have proposed a verification method for Java programs to detect the defects based on the specifications by using Source2UPPAAL implemented as an Eclipse plug-in to assist general developers who have little knowledge of the model-checking tool.

Marc [16] suggested that deviation between a phenomenon of failures and the cause is one of the main factors which make it difficult to detect the defects. Generally, it is difficult to insert an appropriate assertion to be checked into the target source code at the appropriate location. Our contribution is that we can define the user-defined specification-based model using a familiar decision table without deep understanding of the target source code. We can also easily check a Safety property by query expressions automatically generated. Because the user-defined specification-based model is created manually, the accuracy of the model is affected by the developer’s skill. To increase the accuracy of an inspection, we examined the creation of a tool and identified a clear creation procedure. Moreover, we also identified the best method of finding a suitable point of contact between our model and the user-defined specification-based model, both of which were created from the source code.

We believe that this method can be used to verify specifications at the time of migration. If an old program and a new program are migrated into a verification model and the behaviors are compared, we can confirm the compatibility of the old control flow and the new control flow.

Acknowledgments

This work has been conducted as a part of “Research Initiative on Advanced Software Engineering in 2012” supported by Software Reliability Enhancement Center (SEC), Information Technology Promotion Agency Japan (IPA).

References

Saeko Matsuura received the MS degree in mathematics in 1982 from the Graduate School of Tsuda College. She was a researcher at Kanri Kogaku Kenkyuusyo Ltd. She received Dr. Information and Computer Science in 2001 from Waseda University. Since 2002, she was an associate professor at Shibaura Institute of Technology (SIT) and she is currently a professor at SIT since 2006. She is a member of IEEE computer Society, Information Processing Society of Japan (IPSJ). She received the 1993 IPSJ SIG Research Award and the 2004 FIT Paper Award from IPSJ.