Qaisar A. Malik | Johan Lilius | Linas Laibinis

Generating Test Cases from Scenario-based Formal Development

Turku Centre *for* Computer Science

TUCS Technical Report
No 895, October 2008

Generating Test Cases from Scenario-based Formal Development

Qaisar A. Malik
Turku Centre for Computer Science,
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
Qaisar.Malik@abo.fi
Johan Lilius
Turku Centre for Computer Science
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
Johan.Lilius@abo.fi
Linas Laibinis
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
Linas.Laibinis@abo.fi

TUCS Technical Report
No 895, October 2008

Abstract

In this paper, we present an extension of our model-based testing approach that is based on formal models and user-provided testing scenarios. In this approach, the user provides a testing scenario on the level of an abstract model. When the abstract model is refined to add or modify features, the corresponding testing scenarios are automatically refined to incorporate these changes. Often, due to the abstraction gap between a formal model and the implementation, it is not always feasible to generate implementation code from the formal models. As a result, the implementation is not demonstrated to be correct by its construction but instead it is hand-coded by programmer(s). To validate the correctness of the implementation, testing is performed while using user-provided scenarios. The testing scenarios are unfolded into test cases containing the required inputs and expected outputs. To automate this test-case generation process, we provide guidelines for the formal development of system models. We use Event-B as our formal framework. We also propose a methodology for automatic generation of an implementation template in Java and its corresponding JUnit test cases from Event-B specifications and testing scenarios respectively.

Keywords: Scenario-based, Testing, Model-based Testing, Event-B, Formal Methods

TUCS Laboratory
Embedded Systems Laboratory
Distributed Systems Laboratory

1 Introduction

Testing is an important but expensive activity in the software development life cycle. With advancements in the model-based approaches for software development, new ways have been explored to generate test-cases from existing software models of the system, while cutting the cost of testing at the same time. These new approaches are usually referred to as *model-based testing*. A software model is a specification of the system which is developed from the given requirements early in the development cycle [12]. In model-based development (MBD), this model is then refined until the required abstraction level is reached, from which the implementation code can be generated, or written by hand. The same idea is advocated by formal methods (FM), e.g., refinement calculus [9], B [4], Event-B [7]. The principal difference between MBD and FM is the fact that a formal method has a mathematically defined semantics and the refinement steps can be *proven* correct. Application of formal methods result in the system that are "correct-by-construction" thus mitigating the need for testing. However, as we will discuss later, there still remains a gap in the formal development of systems that needs to be bridged by tests.

Model-based testing (MBT) is an approach for deriving tests from software models using automated techniques. The intended cost reductions arise because

1. the tests are generated by tools, without hand coding,
2. the model changes do not imply extensive re-writing of test-code, and
3. test coverability is improved because we measure the coverability from the model.

However, MBT can be seen as a "dumb" testing method, in the sense that it does not address the problem of *what* needs to be tested, e.g., what the *important* parts of the system are. This problem, called *test selection* is an active research area where both formal and informal approaches exist. In [15], we proposed a solution to this problem using user-provided testing scenarios. In this approach, the testing scenarios are derived from the initial requirements provided by the user and are thus intended to describe the important features of the system. Since they are constructed at the initial level of the development they are defined in relation to the initial abstract model that is the starting point of the development process. As the model is further developed in the refinement process, the testing scenarios are also refined to be in sync with the correct model. In Section 2, we will explain how the test scenarios are kept in sync with the model development by refinement.

At the end of the development process the final model is obtained. The assumption is that this model has most of the implementation details incorporated. The goal of formal methods as well as model-based design approaches is that one should then be able to generate the final implementation automatically. If the system has been developed using a formal method, then under the assumption that

the code-generator is proven correct, the final implementation is provably correct too.

However, in practice often quite a lot of implementation details are left open and the implementation has to be still written by hand. Thus there remains the problem of how to test the implementation against user requirements. take our scenarios and transform them into concrete test cases, that can be used to validate and verify the implementation.

A number of tools exist that address this problem in the context of formal methods (e.g., see [10, 19]). But they are based on the coverage graph, which is obtained from symbolic execution of the model. Thus, they suffer from the same problem as traditional model-based test generators, namely, the generated tests have do not distinguish between different parts of the system that might be more or less important for the overall correctness. The whole process is based on the coverage criterias, such as transition coverage, state coverage or any other combinations of these. The problem of these approaches is that they can lead to long test cycles where the test-generator is not distinguishing between different parts of the system, that might be more or less important for the overall correctness of the system (the system might e.g. have been partially generated automatically, and the tester is wasting efforts while testing the "correct-by-construction" part of the system). Our scenario based approach can be seen as an attempt to bring guidance to this process, by explicitly describing important behavior of the system that we want to make sure is tested.

In our previous work [20, 15], we have explored the idea of scenario based testing combined with formal model development. We have shown how we can refine the scenarios based on a model-refinement either automatically [20] or by assuming that the refinement has a structure [15]. In this paper we extend our approach to cover the concrete test generation from the final test scenario to a set of test cases.

The main contributions of the paper are:

- We provide guidelines for stepwise development of *testable* Event-B models,
- We show how requirements (scenarios) are transformed into test-cases and how these test-cases are represented,
- We show how the inputs and expected outputs for a test-case are derived from an Event-B model,
- We outline a methodology showing how an implementation template can be generated from a sufficiently refined specification.

The organisation of the paper is as follows. In Section 2, we give an overview of our scenario-based testing methodology. In Section 3, we provide basic mathematical preliminaries for testing process, while Section 4 covers introduction to

modeling with the Event-B formalism. In Section 5, we describe modeling and refinement of scenarios. In Section 6, we illustrate automatic generation of implementation template for Java. In Section 7, automatic test case generation process for JUnit 4 is described. Finally, Section 8 contains some discussion and concluding remarks.

2 Scenario Based Testing

Our model-based testing approach [20, 15] is based on stepwise system development [9] using behavioral models of the system. By a behavioral model, we mean that the system behavior is modelled as a state transition system with operations (events) used to describe transitions. In the stepwise development process, an abstract model is first constructed and then further refined to include more details (e.g., functionalities) of the system. Generally, these models can be either formal, informal, or both. In this work we only consider formal models.

In the development process, we start with an abstract model M_A and gradually, by a number of refinement steps, obtain a sufficiently detailed concrete model M_C. The final system, the system under test (SUT), is an implementation of this detailed model. Ideally, the implementation should be automatically generated from this model, which would make it *correct by construction* under the assumption that the code generator is correct. However, in practice, the models do not take into account the low-level implementation details. Due to this *abstraction gap* between formal models and executable implementations, automatic generation of implementation code is not always possible. As a result, an implementation is often hand-coded, while consulting the formal models. The left hand-side of the Figure 1 graphically presents this process.

Since the implementation is no longer *correct-by-construction*, there is a need to *test* the implementation. Such tests could be done by hand or generated through automatic test generation. In this paper, we use scenario-based testing [15] to generate tests. We start from the requirements and gradually construct testing scenarios. The right hand side of the Figure 1 depicts this process. In the literature, one can find several definitions of the term *scenario*. In the field of software engineering, scenarios have been used to represent various concepts like system requirements, analysis, user-component interactions, test cases etc. [16]. We use the definition from [1], which defines a scenario as a description of possible actions and events in the future. It can also be thought of as one of the expected functionalities of the system. The abstract scenarios are further refined until a sufficiently refined scenario is obtained. Then, in the final step, tests are generated from these scenarios.

The tests can usually be divided into at least 3 different major kinds:

1. *Unit tests* are the tests that check the functionality of a simple program component, e.g., a function or a class. In our case, a unit test would test the

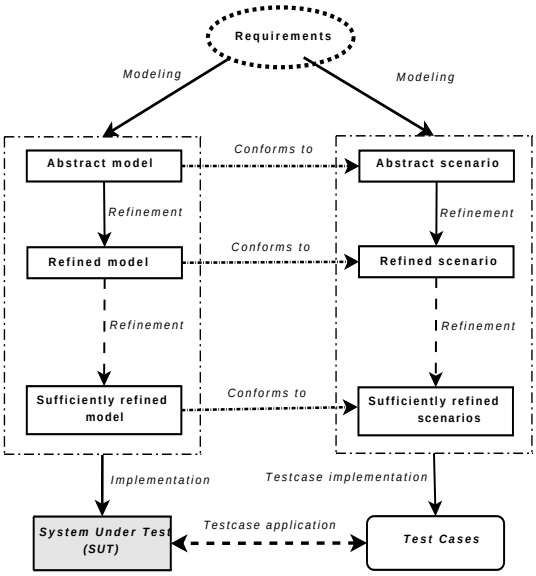

Figure 1: Overview of our Model-based testing approach

correct implementation of an Event-B event.

2. *Integration tests* are the tests that test the combined behavior of several units.

3. *Acceptance tests* are used to validate the system against the requirements.

Basically, integration and acceptance tests are very similar since both types of tests can be thought of as *scenarios*. We use the term *test scenario* to emphasize the intended use of these scenarios.

The challenge is now how to refine the test scenario S_A, along the refinement path from the abstract model to the concrete model, into a concrete test scenario S_C such that S_C covers the same behavior as S_A does. The general structure of the problem is given in the figure 2. In this process, an abstract model M_A is refined by M_i (denoted by $M_A \sqsubseteq M_i$). This refinement ($\sqsubseteq$) is so called a controlled refinement, as will be discussed in section 4.1. Scenario S_A is an abstract scenario, formally satisfiable ($\models$) by specification model M_A, is provided by the user. In the next refinement step, scenario S_i is constructed automatically from M_A, M_i and S_A in such a way that S_i is formally satisfied or conformed by the model M_i. The automatically generated scenario S_i represents functionalities, in part or whole, of the model M_i.

In some cases, the model M_i may contain some extra functionalities or features, such as incorporated fault-tolerance mechanisms, which were omitted or out of scope of scenario S_A. These *extra features*, denoted by S_{EF}, can be added in the scenario S_i manually. The modified scenario $S_i \cup S_{EF}$ must be checked (by means of available tools) to be satisfied/conformed by the model M_i. We can

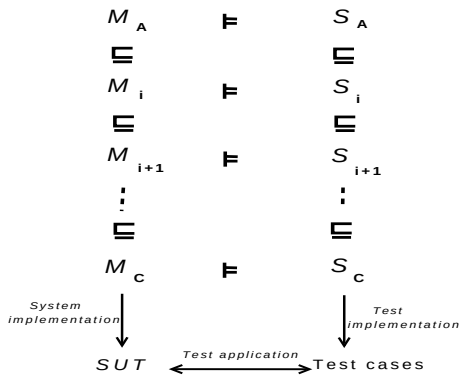

Figure 2: Refinement of Models and Scenarios

follow the same refinement process, now starting with $S_i \cup S_{EF}$, until we get a sufficiently refined scenario at level M_C. In [20], we showed how to derive the S_C given a refinement $M_A \sqsubseteq M_C$ and the scenario S_A. This approach works for any refinement but is exponential in nature. In [15], we proposed a more efficient approach that uses *controlled* refinement. In this case, the scenario S_i can be easily generated by a transformation that mirrors the refinement step. In section 4.1, we will describe this approach more in detail. After the final refinement, the system is implemented while consulting the model M_C. This implementation is called *system under test (SUT)*. Since this implementation is hand-coded, there is no guarantee for its correctness. Going from the scenario S_C to concrete test cases poses a similar problem. We would like to generate test cases automatically from the concrete scenario. In the later sections we will discuss how we can approach this task. First, in the next Section, we present some mathematical preliminaries needed for our model-based testing approach.

3 Mathematical Preliminaries

The formal models that we use in this work are *labelled transition systems*. These are formally defined in the following:

Definition 1

A *labelled transition system* (LTS) is a 4-tuple $\langle S, L, T, s_0 \rangle$ where

- *S* is countable, non-empty set of *states*;
- *L* is a countable set of *labels*;
- $T \subseteq S \times L \times S$ is the *transition relation*
- $s_0 \in S$ is the *initial state*.

The labels in L represent the events in the system. Let $l = \langle S, L, T, s_0 \rangle$ be an LTS with $s, s^{'}$ in S and let $\mu_{(i)} \in L$.

$$\begin{array}{lll} s \xrightarrow{\mu} s' & =_{def} & (s, \mu, s') \in T \\ s \xrightarrow{\mu_1 \ldots \mu_n} s' & =_{def} & \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\ s \xrightarrow{\mu_1 \ldots \mu_n} & =_{def} & \exists s' : s \xrightarrow{\mu_1 \ldots \mu_n} s' \end{array}$$

The behavior of an LTS is defined in terms of *traces* where a *trace* is a finite sequence of events in the system. The set of all traces over L is denoted by L^*. For an LTS $l = \langle S, L, T, s_0 \rangle$, the behavior function, denoted by *beh(LTS)*, is defined as

$$beh(l) \quad =_{def} \quad \{\sigma \in L^* \mid s_0 \xrightarrow{\sigma} \quad \}$$

□

Definition 2

1. A *test sequence*, denoted by *t*, is a finite sequence of events, $\mu_1, \mu_2, \ldots \mu_n$, in the system defined as

   $$s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n$$

   where $n \in \mathbb{N}$ and s_i are system states.

2. A *test scenario* , denoted by *ts*, is collection of *test sequences* present in the *behavior* of the LTS *l*,

   $$ts \subseteq beh(l)$$

□

In the context of Figure 2 and any model M_i as well as the SUT generate an LTS. We can then define

Definition 3

1. The *System Under Test* (SUT) is an executable implementation of the models. Abstractly, an SUT can be viewed as a Labelled Transition System (LTS) having states and events.

2. A *test case* denoted as *tc*, is a finite *test sequence* to be tested on SUT. Moreover, each test case also includes the expected result(s) of the test case execution. This result is used to compute the *verdict function*.

3. A *verdict function* ν is defined, in terms of *Labelled Transition System (LTS)* with *test sequence* (ts), as

$$\nu(LTS, ts) = Passed \qquad iff \qquad ts \in beh(LTS)$$

Similarly, in the context of *System Under Test* (SUT), the verdict function is used to check if the test case execution has given expected results or not.

$$\nu(SUT, tc) = Passed \qquad iff \qquad tc \in beh(SUT)$$
$$Failed \text{ otherwise}$$

□

4 Modeling in Event-B

The Event-B [6, 5] is a recent extension of the classical B-method [4] formalism. Event-B is particularly well-suited for modeling event-based systems. The common examples of event-based systems are reactive systems, embedded systems, network protocols, web-applications and graphical user interfaces.

In Event-B, the specifications are written in Abstract Machine Notation (AMN). An abstract machine encapsulates state (variables) of the machine and describes operations (events) on the state. A simple abstract machine has following general form

MACHINE AM
VARIABLES v
INVARIANT I
EVENTS
$INITIALISATION \;=\; \ldots$
$E_1 \;=\; \ldots$
$\ldots$
$E_N \;=\; \ldots$
END

A machine is uniquely defined by its name in the **MACHINE** clause. The **VARIABLE** clause defines state variables, which are then initialized in the **INITIALISATION** event. The variables are strongly typed by constraining predicates of the machine invariant *I* given in the **INVARIANT** clause. The invariant defines essential system properties that should be preserved during system execution. The operations of event based systems are atomic and are defined in the **EVENT** clause. An event is defined in one of two possible ways

$$E \;=\; \textbf{WHEN } g \textbf{ THEN } S \textbf{ END}$$

$$E \;=\; \textbf{ANY}\ i\ \textbf{WHERE}\ C(i)\ \textbf{THEN}\ S\ \textbf{END}$$

where *g* is a predicate over the state variables *v*, and the body *S* is an Event-B statement specifying how the variables *v* are affected by execution of the event. The second form, with the **ANY** construct, represents a parameterized event where *i* is the parameter and *C(i)* contains condition(s) over *i*. The occurrence of the events represents the observable behavior of the system. The event guard (*g* or *C(i)*) defines the condition under which event is enabled.

Event-B statements are formally defined using the weakest precondition semantics [11]. The defined semantics is used to demonstrate correctness of the system. To show correctness of an event-based system it is necessary to formally prove that the invariant is true in initial state and every event preserves the invariant:

$$wp(INITIALISATION, I) = true, \text{ and}$$
$$g_i \wedge I \Rightarrow wp(E_i, I)$$

In the following, we will see how Event-B specifications are developed in our scenario-based testing methodology.

4.1 Controlled Refinement

In our approach, we use Event-B formalism to model the behaviour of the system. Referring back to Figure 2, in order to automatically refine a scenario from its previous level, we need to identify each and every refinement step taken for the refinement of the corresponding models. This identification is only possible if we follow a controlled and structured approach for the refinement of models. In our earlier work [15], we presented the supported refinement types, for the Event-B, for our testing approach. These types are

- *Atomicity Refinement*
  Where one event operation is replaced by several operations, describing the system reactions in different circumstances the event occurs. Intuitively, it corresponds to a branching in the control flow of the system as shown in Figure 3(a).

- *Superposition Refinement*
  Where new implementation details are introduced into the system in the the form of new events that were invisible in the previous specification. These new events can be non-looping or looping as depicted in Figure 3(b) and (c) respectively.

4.2 Classification of Events

In order to identify the inputs and the outputs of the system, we classify the events of our Event-B models as of *input*, *output* and *internal* types. In the following is given the details of these typing.

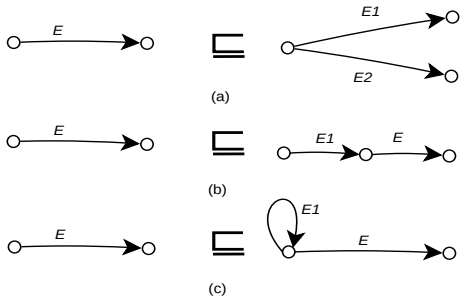

Figure 3: Basic refinement transformations

Definition 4

The Events
Set of all events in the system, denoted by Σ, is divided into following subsets of:

- *Input* events denoted by ε^I
- *Output* events denoted by ε^O
- *Internal* events denoted by ε^τ

□

The *input events*, ε^I, accepts inputs from user or environment. Apart from their input behavior, these events may take part in the normal processing of the system. However, the input events do not produce externally visible output. The *output events* ε^O produce externally visible output. The *internal events* do not take part in any input/output activity. These may produce intermediate results used by other events in ε^I and ε^O. The motivation of this classification is explained in next section where we divide our system into logical functional units.

4.3 Logical Units

As we develop our system in a stepwise manner, the main functional units of a system are identified on abstract level. Each of these abstract functional units are modelled as a separate logical unit, called *IOUnit*, in our Event-B model.

Definition 5

An IOUnit *U* consists of a finite sequence of events and has the following form.

$$U =< \varepsilon^I, \varepsilon^{\tau+}, \varepsilon^O >$$

Here ε^I and ε^O denote the input and output events respectively and $\varepsilon^{\tau+}$ represents one or more occurrences of *internal* events.

□

It can be observed from the above definition that *IOUnit* consists of the sequence of events occurring in such an order that the first event in the unit is always an *input* event and the last event is always an *output* event with one or more *internal* events in between them. Moreover, *IOUnit* can not contain more than one input or output events.

IOUnit takes input and produces output, as the presence of input and output events indicates. The classification of events, from the previous section, helps us in identifying inputs and outputs of each unit, and when combined, of the whole system. The motivation for this approach is the following. The developer(s) of the system under test (SUT) may decide to implement the system independently of the structure of the Event-B model. Indeed, it is sometimes hard to follow the strict one to one mapping between the events of the model and corresponding programming language units. For example, two events in a model can be merged to form one programming-language operation or the functionality of an event in the model may get divided across multiple operations or classes in the implementation. However, for successful execution of the system, the interface of the model and implementation, i.e., the sequence of inputs and the outputs, should remain the same.

4.4 Example

We illustrate our approach by a small example of *hotel reservation system*. Reserving a room in such a system consists of a sequence of events that occur in a particular order. On the abstract level, we may have only a few events which represent some particular functionality of the system. For example, in the *BookingSystem*, the room reservation functionality can be divided into three functional units, namely, *Finding* a room, *Reserving* and *Paying* for it. As we structure our model according to the guidelines described in Section 4.2, the resulting events and their sequence of execution can be seen in Figure 4(a). It can be observed that the main functional events are wrapped with the input and output events. For example, the *Find* event is wrapped around with *InputForFind* and *OutputForFind* events where the events *InputForFind* and *OutputForFind* are the input and output events respectively.

It is also possible to introduce new IOUnits in the model. The new IOUnit must also strictly follow the input-output strcuture and controlled refinement constraints discussed in the previous sections. In the refined model, as shown in Figure 4(b), the new cancellation functionality is introduced as a new IOUnit. Within an input-output unit, we treat our main functional event as an *internal event* (e.g., *Find, Reserve and Pay*). Such events can be further refined and this may add more *internal* events within the input-output unit. This is shown in the refined model where *Find* event is refined into four events. Graphically, these new *internal* events are shown with dashed arrows in Figure 4(c). The full Event-B specifications for both abstract and refined models are given in the Appendix.

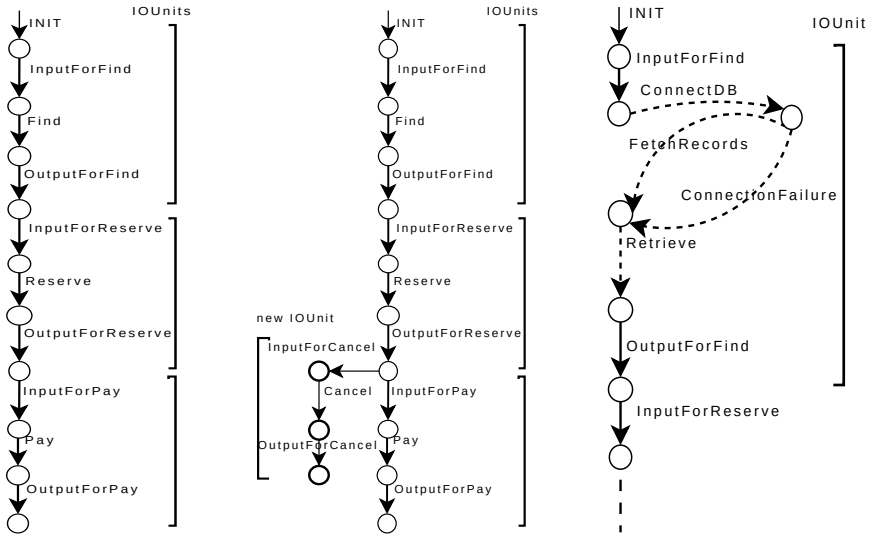

Figure 4: (a) Abstract System (b) First Refinement (c) Second Refinement

5 Modelling and Refinement of Scenarios

In the field of software engineering, scenarios could be represented in several formal, semi formal and informal ways. Some of common representations are tables, scripts, prototypes, structured texts and state charts.

In this paper, we need a simple approach to represent a scenario as sequence of model events. We use Communicating Sequential Process (CSP) [13] for this purpose. The advantage of using CSP is twofold. First, a CSP expression is a convenient way to express several scenarios in a compact form. Second, since we develop our system in a controlled way, i.e. using the basic refinement transformations described in Section 4.1, we can associate these Event-B refinements with syntactic transformations of the corresponding CSP expressions. Therefore, knowing the way model M_i was refined by M_{i+1}, we can automatically refine scenario S_i into S_{i+1}. To check whether a scenario S_i is a valid scenario of its model M_i, i.e., model M_i satisfies $(\models)$ scenario S_i, we use the ProB [14] model checker. ProB supports execution (animation) of Event-B specifications, guided by CSP expressions. In fact, the available tool support is another motivating reason for representing scenarios as CSP expressions. The satisfiability check is performed at each refinement level as shown in the Figure 2. The refinement of scenario S_i is the CSP trace-refinement [17] denoted by $\sqsubseteq_T$. The details and examples for scenario refinements can be found in our earlier work [15].

As described earlier, a scenario is a finite sequence of events occurring in some particular order. Since we have grouped the events in the form of logical IOUnits, our scenarios will also include a finite sequence of IOUnits on the logical level. It means that scenarios will include the same events as there are in the Event-

B model. However, the scenarios must follow the same rules that were set for constructing IOUnits in previous section, i.e.,

1. The first event in the scenario is always an *input* event.
2. The last event in the scenario is always an *output* event.
3. There can not be two input-type events in sequence without any output event in between them, i.e. the following sequence in a CSP expression is not allowed.

$$< \cdots \rightarrow \varepsilon_k^I \rightarrow \varepsilon_{k+1}^I \rightarrow \cdots >$$

4. There can not be two output-type events in sequence without any input event in between them i.e. the following sequence is also not allowed.

$$< \cdots \rightarrow \varepsilon_k^O \rightarrow \varepsilon_{k+1}^O \rightarrow \cdots >$$

We will see the scenario examples later in this section.

5.1 Input and Output

Since the scenarios are defined on abstract level, they lack details about inputs and outputs. Therefore, to construct concrete test cases we need to identify details about inputs for the test cases. These input details are identified from the input event(s) of each IOUnit. For example, if an input event reads three input variables then these three variables become the inputs for the unit that the input event belongs to. The details about inputs can be retrieved from an Event-B model as the model specifies the type, initial value and invariant properties for all variables.

The expected outputs can be generated after the model is animated using ProB model checker. For a given input of a test case, the ProB can animate the model and return the result, which is then saved as the *expected* output of the test case. This expected output can be used to compare the values while testing the real implementation. The ProB model checker can only provide output values based on the available abstract values. For example, to test whether a room is available in the *Hotel Booking System*, ProB can check the expected result for a pre-defined set of inputs, while in the actual implementation this result might be retrieved from the database. Therefore, it is the responsibility of the user to setup the test case accordingly.

5.2 Examples

In the case of the previously discussed *Hotel Reservation System* example, there can be many possible testing scenarios. For example, if we want to test the *room finding* functionality, the scenario in the form of CSP expression, with inputs and outputs, would be as follows.

$$\begin{aligned} \mathrm{S}_0 = \texttt{inputForFind}?\texttt{roomType} \rightarrow \texttt{connectDB} \rightarrow \\ (\texttt{FetchRecords} \sqcap \texttt{ConnectionFailure}) \rightarrow \texttt{retrieve} \rightarrow \\ \texttt{outputForFind}!(\texttt{roomId}, \texttt{ifException}) \end{aligned}$$

where $\sqcap$ is an internal choice operator in CSP. The variable `roomType` is the input for this IOUnit and `roomId`, `ifException` are the possible outputs. The variable `ifException` specifies if there was any exception, e.g., a connection failure. To test the *reservation*, the corresponding CSP expression would be

$$\mathrm{S}_1 = \texttt{inputForReserve}?\texttt{roomId} \rightarrow \texttt{reserve} \rightarrow \texttt{outputForReserve}!\texttt{reserveId}$$

Often, the subsequent scenario step depends on the results of the previous ones. Therefore, to test finding and reservation IOUnits in a sequence, a scenario would be

$$\mathrm{S}_3 = \mathrm{S}_0; \mathrm{S}_1$$

In Section 7, we will see how a scenario is used to generate a template for JUnit test-cases.

6 Java implementation template for Event-B models

Event-B specifications, developed, as described in previous sections, can be used to generate Java implementation template. We start by translating a (sufficiently refined) Event-B model into Java class. Similarly, the Event-B events are translated as the Java methods. For our *Hotel Booking System* example, the excerpts of Event-B machine and its implementation template are shown in the following and in the Listing 1 respectively. For the complete listing of the specifications, please refer to the Appendix.

MACHINE BookingSystemRef1

REFINES BookingSystem

SEES BookingContext

VARIABLES

$roomType$

...

INVARIANTS

...

EVENTS

Initialisation

$act5: \ roomType := Null\_roomType$

...

Event InputForFind $\widehat{=}$

Refines InputForFind

any

tt

where

$grd1: \ tt \in RTYPES$

$grd2: \ connection = No\_connection$

then

$act1: \ roomType := tt$

$act2: \ inputForFindCompleted := TRUE$

end

...

END

An event in the Event-B specification consists of two parts. The first part contains the pre-condition(s) for the event to be enabled, while the second part consists of the actions that event performs. For every event in Event-B model, we create two separate methods in the Java implementation, to represent the pre-conditions and actions respectively. The first method, which contains the pre-conditions/guards of an event, returns the evaluation result in the form of *boolean* value. The name of this method is pre-fixed with the string "guard_". The second method encapsulates the actions of the event. Since the actions update the class-level/global variables, this method returns *void*. For example, for the *InputForFind* event from our *Hotel Booking System* example, the Java implementation methods are given in the Listing 1.

Listing 1: Implementation template for HotelBookingSystem

```
public class HotelBookingSystem {

  //class-level variables
  public String roomType;
```

```
  public HotelBookingSystem(){
    //initialization  ...
  }

  /* PreConditions/Guards for InputForFind event*/
  private boolean guard_inputForFind(){
    return (roomType != null) ;
  }

  /* Implementation method for InputForFind event */
  public void inputForFind() throws PreConditionViolatedException
  {
    if(guard_inputForFind()){
      //actions ...
    }
    else{
      throw new PreConditionViolatedException("For inputForFind");
    }
  }

  //more Implementation methods for events
  ...
  ...
}
class PreConditionViolatedException extends Exception
{
  public PreConditionViolatedException(String mesg){
    super(mesg);
  }
}
```

Each implementation method, representing an event, first evaluates its pre condition(s) by calling its "guard_" method. If the pre-conditions are evaluated to *false* then an exception, `PreConditionViolatedException`, is raised, otherwise the actions of the corresponding event are executed. The variables of a machine are translated into the class variables in Java. The type information for variables can be retrieved from the **Invariant** clause of the machine. By default, the primitive types, e.g., BOOL, numerics etc., are translated into the corresponding Java types. For composite or user-defined types, the user can provide a translation mapping of types, from Event-B to Java separately. In the next Section, we will discuss how scenarios are translated into executable test cases.

7 JUnit Test case generation

In previous section, we have seen the guidelines for generating an implementation template for Java. Once such template is generated, we can generate the corresponding executable test cases from the scenarios, using JUnit [2] - Java Unit Testing framework.

Since, our events are sequenced in terms of *IOUnits*, we can write JUnit test cases to test these *IOUnits*. For example, on the basis of branching, there are two possible test-cases in the following scenario.

$$\begin{aligned} S_0 = \texttt{inputForFind}?\texttt{roomType} \rightarrow \texttt{connectDB} \rightarrow \\ (\texttt{FetchRecords} \sqcap \texttt{ConnectionFailure}) \rightarrow \texttt{retrieve} \rightarrow \\ \texttt{outputForFind}!(\texttt{roomId}, \texttt{ifException}) \end{aligned}$$

These test cases T_0 and T_1 are

$$T_0 = \texttt{inputForFind?roomType} \rightarrow \texttt{connectDB} \rightarrow \texttt{FetchRecords} \rightarrow \texttt{retrieve} \rightarrow \texttt{outputForFind!}(\texttt{roomId}, \texttt{ifException})$$

$$T_1 = \texttt{inputForFind?roomType} \rightarrow \texttt{connectDB} \rightarrow \texttt{ConnectionFailure} \rightarrow \texttt{retrieve} \rightarrow \texttt{outputForFind!}(\texttt{roomId}, \texttt{ifException})$$

For each of the test cases T_0 and T_1, a separate JUnit test method is implemented. The JUnit test method for T_0 is shown in the Listing 2.

Listing 2: JUnit Test method for T_0

```
public class HotelBookingSystemTest {

  HotelBookingSystem bSys;
  ...

  @Before
  public void setUp() throws Exception {
    bSys = new HotelBookingSystem();
  }

  @Test
  public final void T0(){
    try{
      //setting input for IOUnit
      bSys.roomType = "Single";

      //calling methods of IOUnit
      bSys.inputForFind();
      bSys.connectDB();
      bSys.fetchRecords();
      bSys.retrieve();
      bSys.outputForFind();

      //assert statements (verdict)
      assertTrue("non-Empty resultSet", bSys.resultSet.size() > 1);
      assertTrue(bSys.noException == true);
    }
    catch(PreConditionViolatedException e){
      fail(e.getMessage());
    }
  }
```

In similar way, the templates for JUnit test-methods can be generated for each test-case in the scenarios. The above example showed the test cases based on data-flow branching. However, it is possible to generate test cases based on different class of input values. If a scenario involves multiple *IOUnits* in a sequence then JUnit test would involve calls to the relevant test methods. For example as shown in the Listing 3, a JUnit test method tests two *IOUnits*.

Listing 3: Testing multiple IOUnits

```
@Test
  public final void T3(){

  //test method from first IOUnit
```

```
T0();

//test method from second IOUnit
T2();
}
```

Hence, it is possible to generate different combinations of test cases, for a scenario, which might be laborious and error-prone if done by hand.

8 Conclusions and Future work

In this paper, we presented a model-based testing approach using user-provided testing scenarios. These scenarios are first validated using a model checker and then used to generate test cases. Additionally, we have provided the guidelines for stepwise development of formal models and automatic refinement of testing scenarios. We also proposed an approach to generate Java language implementation templates from Event-B models. The abstract testing scenarios can then be used to generate templates for JUnit test cases.

Although, we do not provide complete translation of model and testing scenarios. However, we provide guidelines describing how user can benefit from our approach. The presented work is in progress and will be extended in future to fully automate the development and testing process. In particular, we plan to develop a tool that can do most of the translations automatically. This tool will be available as a plug-in for RODIN [3] platform which is a formal development platform for Event-B specifications.

At the moment, we do not support translation of some complex pre-condition and invariant expressions into Java. Namely, the existential and universal quantifiers are not covered. However, in future, we plan to support for full translation. We also intend to use a graphical notation for representing testing scenarios. A more challenging task is to automatically generate the range of data values for the inputs of the test cases. There exist various sophisticated approaches, for example, input-space-partitioning [8], boundary-value-testing [18] etc., that can be used to generate input values for the test cases. However, a mapping between abstract data and concrete data types needs to be provided by the user. On the other hand, for similar reasons, there is also a need for finding a mapping relation between the outputs generated from the models and actual outputs observed from the implementation.

References

[1] Cambridge Dictionary for English. Available online at http://dictionary.cambridge.org/.

[2] JUnit 4. http://www.junit.org.

[3] Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.

[4] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[5] J.-R. Abrial. Event Driven Sequential Program Construction. 2000. Available at http://www.matisse.qinetiq.com.

[6] J.-R. Abrial and L.Mussat. Introducing Dynamic Constraints in B. Second International B Conference, LNCS 1393, Springer-Verlag, April 1998.

[7] Jean-Raymond Abrial. A system development process with event-b and the rodin platform. In *ICFEM*, pages 1–3, 2007.

[8] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[9] Ralph-Johan Back and Joakim von Wright. Refinement calculus, part i: Sequential nondeterministic programs. In *REX Workshop*, pages 42–66, 1989.

[10] Eddy Bernard, Bruno Legeard, Xavier Luck, and Fabien Peureux. Generation of test sequences from formal specifications: Gsm 11-11 standard case study. *Softw. Pract. Exper.*, 34(10):915–948, 2004.

[11] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[12] Dalal S.R. et al. Model Based Testing in Practice . Proc. of the ICSE'99,Los Angeles,pp 285-294, 1999.

[13] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[14] M. Leuschel and M. Butler. ProB: A model checker for B. Proc. of FME 2003, Springer-Verlag LNCS 2805, pages 855-874., 2003.

[15] Qaisar A. Malik, Johan Lilius, and Linas Laibinis. Model-based testing using scenarios and event-b refinements. Proceedings of Methods and Models for Fault-Tolerance(MeMoT), Oxford, UK, 2007.

[16] Leila Naslavsky, Thomas A. Alspaugh, Debra J. Richardson, and Hadar Ziv. Using Scenarios to support traceability. Proc of 3rd int. workshop on Traceability in emerging forms of software engineering, 2005.

[17] A.W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998 amended 2005.

[18] Philip Samuel and Rajib Mall. Boundary value testing based on uml models. In *ATS '05: Proceedings of the 14th Asian Test Symposium on Asian Test Symposium*, pages 94–99, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Manoranjan Satpathy, Michael Leuschel, and Michael J. Butler. ProTest: An automatic test environment for B specifications. *Electr. Notes Theor. Comput. Sci.*, 111:113–136, 2005.

[20] Manoranjan Satpathy, Qaisar A. Malik, and Johan Lilius. Synthesis of scenario based test cases from b models. In *FATES/RV*, pages 133–147, 2006.

APPENDIX

An Event-B Specification of HotelBookingSystem. Developed and proved using RODIN Platform

CONTEXT BookingContext

SETS

$ROOMS$

$RTYPES$

$REQUESTS$

$CONNECTIONS$

CONSTANTS

$Reservation$

$Payment$

No_room

$rtypes$

$Null\_roomType$

db

$No\_connection$

$Connection\_success$

$Connection\_failure$

$Cancellation$

AXIOMS

$axm1:\ REQUESTS = \{Reservation, Payment, Cancellation\}$

$axm2:\ Reservation \neq Payment$

$axm13:\ Reservation \neq Cancellation$

$axm12:\ Cancellation \neq Payment$

$axm3:\ No\_room \in ROOMS$

$axm4:\ rtypes \in RTYPES \rightarrow \mathbb{P}(ROOMS)$

$axm5:\ Null\_roomType \in RTYPES$

$axm6:\ db \in RTYPES \rightarrow \mathbb{P}(ROOMS \setminus \{No\_room\})$

$axm7:\ db = rtypes$

$axm8:\ CONNECTIONS = \{No\_connection, Connection\_success, Connection\_failure\}$

$axm9:\ No\_connection \neq Connection\_success$

$axm10:\ No\_connection \neq Connection\_failure$

$axm11:\ Connection\_success \neq Connection\_failure$

END

MACHINE BookingSystemAbstract

SEES BookingContext

VARIABLES

$bookedRoom$

$searchResult$

$reservedRoom$

$roomType$

$inputForFindCompleted$

$inputForReserveCompleted$

$reservationCompleted$

$paymentCompleted$

$inputForPayCompleted$

$roomID$

$reservationID$

$sessionCompleted$

$findCompleted$

INVARIANTS

$inv1 : roomType \in RTYPES$

$inv3 : inputForFindCompleted \in BOOL$

$inv4 : inputForReserveCompleted \in BOOL$

$inv5 : inputForPayCompleted \in BOOL$

$inv6 : reservationCompleted \in BOOL$

$inv7 : inputForReserveCompleted \in BOOL$

$inv9 : paymentCompleted \in BOOL$

$inv8 : searchResult \in ROOMS$

$inv10 : bookedRoom \subseteq ROOMS$

$inv13 : reservedRoom \in ROOMS$

$inv11 : roomID \in ROOMS$

$inv14 : reservationID \in ROOMS$

$inv12 : sessionCompleted \in BOOL$

$inv15 : findCompleted \in BOOL$

EVENTS

Initialisation

begin

$act1: bookedRoom := \varnothing$
$act3: searchResult := No\_room$
$act4: reservedRoom := No\_room$
$act5: roomType := Null\_roomType$
$act7: inputForFindCompleted := FALSE$
$act8: reservationCompleted := FALSE$
$act9: inputForReserveCompleted := FALSE$
$act11: paymentCompleted := FALSE$
$act12: inputForPayCompleted := FALSE$
$act10: roomID := No\_room$
$act13: reservationID := No\_room$
$act2: sessionCompleted := FALSE$
$act14: findCompleted := FALSE$

end

Event InputForFind $\widehat{=}$

any

tt

where

$grd1: tt \in RTYPES$

then

$act1: roomType := tt$
$act2: inputForFindCompleted := TRUE$

end

Event Find $\widehat{=}$

any

rr

where

$grd1: rr \subseteq ROOMS$
$grd3: rr \cap bookedRoom = \varnothing$
$grd4: rr \subseteq rtypes(roomType)$

$grd2:\ inputForFindCompleted = TRUE$

$grd5:\ No\_room \notin rr$

then

$act1:\ searchResult :|(rr = \varnothing \Rightarrow searchResult' = No\_room)$
$\wedge\ (rr \neq \varnothing \Rightarrow searchResult' \in rr)$

$act2:\ findCompleted := TRUE$

end

Event OutputForFind $\widehat{=}$

when

$grd1:\ findCompleted = TRUE$

then

$act2:\ roomID := searchResult$

end

Event InputForReserve $\widehat{=}$

any

req

where

$grd3:\ req \in REQUESTS$

$grd4:\ req = Reservation$

$grd2:\ roomID \neq No\_room$

then

$act1:\ inputForReserveCompleted := TRUE$

end

Event Reserve $\widehat{=}$

any

req

where

$grd5:\ roomID \neq No\_room$

$grd2:\ req \in REQUESTS$

$grd1:\ req = Reservation$

$grd4:\ inputForReserveCompleted = TRUE$

then

$act1: reservedRoom := roomID$

end

Event OutputForReserve $\widehat{=}$

when

$grd1: reservedRoom \neq No\_room$

then

$act1: reservationCompleted := TRUE$

$act2: reservationID := roomID$

end

Event InputForPay $\widehat{=}$

any

req

where

$grd5: req \in REQUESTS$

$grd4: req = Payment$

$grd1: reservationCompleted = TRUE$

$grd2: reservationID \neq No\_room$

then

$act1: inputForPayCompleted := TRUE$

end

Event Pay $\widehat{=}$

any

$reservedRooms$

where

$grd1: inputForPayCompleted = TRUE$

$grd3: reservedRoom \neq No\_room$

$grd2: reservedRooms \subseteq \{reservedRoom\}$

then

$act2: bookedRoom := bookedRoom \cup reservedRooms$

$act3: reservedRoom := No\_room$

$act1: paymentCompleted := TRUE$

end

Event OutputForPay $\widehat{=}$

when

$grd1: \ paymentCompleted = TRUE$

then

$act1: \ sessionCompleted := TRUE$

end

END

MACHINE BookingSystemRefined1

REFINES BookingSystemAbstract

SEES BookingContext

VARIABLES

$bookedRoom$

$searchResult$

$paidRoom$

$reservedRoom$

$roomType$

$inputForFindCompleted$

$inputForReserveCompleted$

$reservationCompleted$

$paymentCompleted$

$inputForPayCompleted$

$roomID$

$reservationID$

$inputForCancellationCompleted$

$cancellationCompleted$

$roomCancelled$

$sessionCompleted$

$findCompleted$

INVARIANTS

$inv1 : inputForCancellationCompleted \in BOOL$

$inv2 : cancellationCompleted \in BOOL$

$inv3 : roomCancelled \in BOOL$

$inv4 : paidRoom \subseteq ROOMS$

$inv5 : paidRoom \subseteq bookedRoom$

$inv6 : sessionCompleted \in BOOL$

$inv7 : findCompleted \in BOOL$

EVENTS

Initialisation

begin

$act1: bookedRoom := \varnothing$
$act2: paidRoom := \varnothing$
$act3: searchResult := No\_room$
$act4: reservedRoom := No\_room$
$act5: roomType := Null\_roomType$
$act7: inputForFindCompleted := FALSE$
$act8: reservationCompleted := FALSE$
$act9: inputForReserveCompleted := FALSE$
$act11: paymentCompleted := FALSE$
$act12: inputForPayCompleted := FALSE$
$act10: roomID := No\_room$
$act13: reservationID := No\_room$
$act14: inputForCancellationCompleted := FALSE$
$act15: cancellationCompleted := FALSE$
$act16: roomCancelled := FALSE$
$act17: sessionCompleted := FALSE$
$act18: findCompleted := FALSE$

end

Event InputForFind $\widehat{=}$

Refines InputForFind

any

tt

where

$grd1: tt \in RTYPES$

then

$act1: roomType := tt$
$act2: inputForFindCompleted := TRUE$

end

Event Find $\widehat{=}$

Refines Find

any

rr

where

$grd1: rr \subseteq ROOMS$

$grd3: rr \cap bookedRoom = \varnothing$

$grd4: rr \subseteq rtypes(roomType)$

$grd2: inputForFindCompleted = TRUE$

$grd5: No\_room \notin rr$

then

$act1: searchResult :| (rr = \varnothing \Rightarrow searchResult' = No\_room) \wedge (rr \neq \varnothing \Rightarrow searchResult' \in rr)$

$act2: findCompleted := TRUE$

end

Event OutputForFind $\widehat{=}$

Refines OutputForFind

when

$grd1: findCompleted = TRUE$

then

$act2: roomID := searchResult$

end

Event InputForReserve $\widehat{=}$

Refines InputForReserve

any

req

where

$grd3: req \in REQUESTS$

$grd4: req = Reservation$

$grd2: roomID \neq No\_room$

then

$act1: inputForReserveCompleted := TRUE$

end

Event Reserve $\widehat{=}$

Refines Reserve

any

req

where

$grd5: roomID \neq No\_room$

$grd2: req \in REQUESTS$

$grd1: req = Reservation$

$grd4: inputForReserveCompleted = TRUE$

then

$act1: reservedRoom := roomID$

end

Event OutputForReserve $\widehat{=}$

Refines OutputForReserve

when

$grd1: reservedRoom \neq No\_room$

then

$act1: reservationCompleted := TRUE$

$act2: reservationID := roomID$

end

Event InputForCancellation $\widehat{=}$

any

req

where

$grd1: req = Cancellation$

$grd2: req \in REQUESTS$

$grd3: reservationID \neq No\_room$

$grd4: reservationCompleted = TRUE$

then

$act1: inputForCancellationCompleted := TRUE$

end

Event Cancel $\widehat{=}$

when

$grd1: inputForCancellationCompleted = TRUE$

$grd2: \ reservationID \neq No\_room$

then

$act2: \ roomCancelled := TRUE$

end

Event OutputForCancellation $\widehat{=}$

when

$grd1: \ roomCancelled = TRUE$

then

$act1: \ cancellationCompleted := TRUE$

end

Event InputForPay $\widehat{=}$

Refines InputForPay

any

req

where

$grd5: \ req \in REQUESTS$
$grd4: \ req = Payment$
$grd1: \ reservationCompleted = TRUE$
$grd2: \ reservationID \neq No\_room$

then

$act1: \ inputForPayCompleted := TRUE$

end

Event NoPayment $\widehat{=}$

Refines Pay

when

$grd1: \ inputForPayCompleted = TRUE$
$grd3: \ reservedRoom \neq No\_room$
$grd2: \ cancellationCompleted = TRUE$

with

$reservedRooms: \ reservedRooms = \varnothing$

then

$act2:\ paymentCompleted := TRUE$

$act1:\ reservedRoom := No\_room$

end

Event Pay $\widehat{=}$

Refines Pay

when

$grd1:\ inputForPayCompleted = TRUE$

$grd3:\ reservedRoom \neq No\_room$

$grd2:\ cancellationCompleted = FALSE$

with

$reservedRooms:\ reservedRooms = \{reservedRoom\}$

then

$act1:\ paidRoom := paidRoom \cup \{reservedRoom\}$

$act2:\ bookedRoom := bookedRoom \cup \{reservedRoom\}$

$act3:\ reservedRoom := No\_room$

$act4:\ paymentCompleted := TRUE$

end

Event OutputForPay $\widehat{=}$

Refines OutputForPay

when

$grd1:\ paymentCompleted = TRUE$

then

$act1:\ sessionCompleted := TRUE$

end

END

MACHINE BookingSystemRefined2

REFINES BookingSystemRefined1

SEES BookingContext

VARIABLES

$bookedRoom$
$searchResult$
$paidRoom$
$reservedRoom$
$roomType$
$inputForFindCompleted$
$inputForReserveCompleted$
$reservationCompleted$
$paymentCompleted$
$inputForPayCompleted$
$roomID$
$reservationID$
$inputForCancellationCompleted$
$cancellationCompleted$
$roomCancelled$
$sessionCompleted$
$connection$
$result$
$fetchCompleted$
$retrieveCompleted$
$connectionException$
$ifAnyException$
$findCompleted$

INVARIANTS

$inv1 : connection \in CONNECTIONS$
$inv2 : result \subseteq ROOMS$
$inv3 : fetchCompleted \in BOOL$

$inv4:\ retrieveCompleted \in BOOL$

$inv6:\ (fetchCompleted = TRUE) \Rightarrow result \subseteq rtypes(roomType)$

$inv7:\ (fetchCompleted = TRUE) \Rightarrow connection = Connection\_success$

$inv8:\ (connection = No\_connection) \Rightarrow (fetchCompleted = FALSE)$

$inv9:\ connection \in \{Connection\_success, Connection\_failure\} \Rightarrow inputForFindComplete$
$TRUE$

$inv10:\ connectionException \in BOOL$

$inv11:\ ifAnyException \in BOOL$

$inv12:\ findCompleted \in BOOL$

$inv13:\ (connection = Connection\_failure) \Rightarrow (result = \varnothing)$

$inv14:\ ((connectionException = TRUE) \Rightarrow (connection = Connection\_failure))$

$inv5:\ (fetchCompleted = TRUE \vee connectionException = TRUE) \Rightarrow$
$result \cap bookedRoom = \varnothing$

$inv16:\ (connectionException = TRUE) \Rightarrow connection = Connection\_failure$

$inv17:\ result \subseteq ROOMS \setminus \{No\_room\}$

EVENTS

Initialisation

begin

$act1:\ bookedRoom := \varnothing$

$act2:\ paidRoom := \varnothing$

$act3:\ searchResult := No\_room$

$act4:\ reservedRoom := No\_room$

$act5:\ roomType := Null\_roomType$

$act7:\ inputForFindCompleted := FALSE$

$act8:\ reservationCompleted := FALSE$

$act9:\ inputForReserveCompleted := FALSE$

$act11:\ paymentCompleted := FALSE$

$act12:\ inputForPayCompleted := FALSE$

$act10:\ roomID := No\_room$

$act13:\ reservationID := No\_room$

$act14:\ inputForCancellationCompleted := FALSE$

$act15:\ cancellationCompleted := FALSE$

$act16:\ roomCancelled := FALSE$

$act17:\ sessionCompleted := FALSE$

$act18:\ connection := No\_connection$

$act19:\ result := \varnothing$

$act20:\ fetchCompleted := FALSE$

$act21:\ retrieveCompleted := FALSE$

$act22:\ connectionException := FALSE$

$act23:\ ifAnyException := FALSE$

$act24:\ findCompleted := FALSE$

end

Event InputForFind $\widehat{=}$

Refines InputForFind

any

tt

where

$grd1:\ tt \in RTYPES$

$grd2:\ connection = No\_connection$

then

$act1:\ roomType := tt$

$act2:\ inputForFindCompleted := TRUE$

end

Event ConnectDB $\widehat{=}$

when

$grd1:\ inputForFindCompleted = TRUE$

$grd2:\ result = \varnothing$

$grd3:\ connectionException = FALSE$

$grd4:\ fetchCompleted = FALSE$

then

$act1:\ connection :\in \{Connection\_success, Connection\_failure\}$

end

Event FetchRecords $\widehat{=}$

when

$grd1:\ connection = Connection\_success$

$grd2:\ fetchCompleted = FALSE$

then

$act1: result := db(roomType) \setminus bookedRoom$

$act2: fetchCompleted := TRUE$

end

Event ConnectionFailed $\widehat{=}$

when

$grd1: connection = Connection\_failure$

$grd2: fetchCompleted = FALSE$

then

$act1: result := \varnothing$

$act3: connectionException := TRUE$

end

Event Retrieve $\widehat{=}$

Refines Find

when

$grd1: fetchCompleted = TRUE \vee connectionException = TRUE$

with

$rr: rr = result$

then

$act1: searchResult :\mid ((result = \varnothing \vee connectionException = TRUE) \Rightarrow searchResult' = No\_room) \wedge (result \neq \varnothing \Rightarrow searchResult' \in result)$

$act2: findCompleted := TRUE$

end

Event OutputForFind $\widehat{=}$

Refines OutputForFind

when

$grd1: findCompleted = TRUE$

then

$act2: roomID := searchResult$

$act3: ifAnyException := connectionException$

end

Event InputForReserve $\widehat{=}$

Refines InputForReserve

any

req

where

$grd3: req \in REQUESTS$
$grd4: req = Reservation$
$grd2: roomID \neq No\_room$

then

$act1: inputForReserveCompleted := TRUE$

end

Event Reserve $\widehat{=}$

Refines Reserve

any

req

where

$grd5: roomID \neq No\_room$
$grd2: req \in REQUESTS$
$grd1: req = Reservation$
$grd4: inputForReserveCompleted = TRUE$

then

$act1: reservedRoom := roomID$

end

Event OutputForReserve $\widehat{=}$

Refines OutputForReserve

when

$grd1: reservedRoom \neq No\_room$

then

$act1: reservationCompleted := TRUE$
$act2: reservationID := roomID$

end

Event InputForCancellation $\widehat{=}$

Refines InputForCancellation

any

req

where

$grd1: \ req = Cancellation$
$grd2: \ req \in REQUESTS$
$grd3: \ reservationID \neq No\_room$
$grd4: \ reservationCompleted = TRUE$

then

$act1: \ inputForCancellationCompleted := TRUE$

end

Event Cancel $\widehat{=}$

Refines Cancel

when

$grd1: \ inputForCancellationCompleted = TRUE$
$grd2: \ reservationID \neq No\_room$

then

$act2: \ roomCancelled := TRUE$

end

Event OutputForCancellation $\widehat{=}$

Refines OutputForCancellation

when

$grd1: \ roomCancelled = TRUE$

then

$act1: \ cancellationCompleted := TRUE$

end

Event InputForPay $\widehat{=}$

Refines InputForPay

any

req

where

$grd5: req \in REQUESTS$

$grd4: req = Payment$

$grd1: reservationCompleted = TRUE$

$grd2: reservationID \neq No\_room$

then

$act1: inputForPayCompleted := TRUE$

end

Event NoPayment $\widehat{=}$

Refines NoPayment

when

$grd1: inputForPayCompleted = TRUE$

$grd3: reservedRoom \neq No\_room$

$grd2: cancellationCompleted = TRUE$

then

$act2: paymentCompleted := TRUE$

$act1: reservedRoom := No\_room$

end

Event Pay $\widehat{=}$

Refines Pay

when

$grd1: inputForPayCompleted = TRUE$

$grd3: reservedRoom \neq No\_room$

$grd2: cancellationCompleted = FALSE$

then

$act1: paidRoom := paidRoom \cup \{reservedRoom\}$

$act2: bookedRoom := bookedRoom \cup \{reservedRoom\}$

$act3: reservedRoom := No\_room$

$act4: paymentCompleted := TRUE$

$act5: result := result \setminus \{reservedRoom\}$

end

Event OutputForPay $\widehat{=}$

Refines OutputForPay

when

$grd1: \ paymentCompleted = TRUE$

then

$act1: \ sessionCompleted := TRUE$

end

END

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2101-9
ISSN 1239-1891