Requirement-driven Scenario-based Testing Using Formal Stepwise Development

Qaisar A. Malik, Linas Laibinis, Dragoş Truşcan, and Johan Lilius
Turku Centre for Computer Science and Dept. of Information Technologies,
Åbo Akademi University, Turku, Finland.
Email: {Qaisar.Malik, Linas.Laibinis, Dragos.Truscan, Johan.Lilius}@abo.fi

Abstract

This article presents a scenario-based testing approach, in which user-defined abstract testing scenarios of the SUT are automatically refined based on formal specifications of the system under test (SUT). The latter are specified in a stepwise manner using the Event-B formalism until a sufficiently refined specification is obtained, which is then used to generate a Java implementation template of the system. The development of the specification is driven by the requirements of the system which are traced throughout the development and testing process. Abstract testing scenarios, provided by the user, are automatically refined following the same refinement steps used for the system specification. The sufficiently refined scenarios are then transformed into executable Java Unit Testing (JUnit) test cases, which are executed against the Java implementation of the SUT. During the described process, the requirements linked to the testing scenarios are propagated to JUnit tests. The main advantage of the proposed approach that it allows the developer to evaluate which requirements have been validated and to trace back the failed tests to corresponding elements of the formal specifications.

Index Terms

Scenario-based testing; Requirements Traceability; Event-B; Formal Refinement; JUnit;

1. Introduction

Formal development ensures that the developed systems are correct-by-construction. However, development of large and complex systems by formal methods exhibit several limitation including computation time and efforts it takes for the verification, and handling of low-level implementation details [1]. In general practice, formal methods are used to verify specifications of the system abstractly and implementation of the system is hand-coded while following formal specifications. In such cases, the implementation is written in an informal programming language. Since the implementation is no longer *correct-by-construction*, the resulting implementation needs to be tested.

Traditionally, testing has been performed manually by the tester carefully examining the implementation under test and then designing test cases. As the software became more complex, the resulting test cases have grown in numbers and complexity. This naturally has lead to the need to automate the testing process. Today, there exist several testing approaches that automate the testing process either completely or partially. These approaches try to achieve their goal by applying different means, i.e., code templates, scripts, formal and semi-formal software models etc. However, these approaches do not distinguish between different parts of the system that might be more or less important for overall system correctness. Therefore, it is important to test the functionality of the system according to user's requirements.

In this paper, we propose a testing methodology which uses user-specified testing scenarios in order to generate test cases. Our scenario-based testing approach can be seen as a kind of model-based testing where tests are generated from user-provided testing scenarios. The focus of this testing approach is on explicit identification of important behavior of the system that should be tested. The proposed methodology uses formal models of the system along with user-provided testing scenarios. These formal models and scenarios are mapped using the requirements. Later on, the formal models and scenarios are translated to Java and JUnit artifacts, respectively, while the requirements are also propagated to the JUnit test cases. The advantage of propagating requirements to executable test cases is that upon a a test case failure, it is possible to back-trace the failed requirement(s) to the corresponding parts in the model.

The work we present in this paper builds on and extends our previous work [9], [10] on scenario-based testing, where we have used formal models of the SUT based on the Event-B formalism. We have also proposed a collection of formal refinement techniques that can be used in the context of test generation. In this article, we elaborate our previous results and as well as compliment them by building a requirement traceability support. This allows us to keep track on how requirements are addressed by the specification at different abstraction levels and how they are propagated to the generated test cases.

To summarize, our proposed methodology encompasses the following:

- inclusion of requirements in the formal specification process and propagation of requirements to tests;
- traceability of requirements from tests back to formal specifications;
- identification of abstract test cases from formal scenario specifications;
- generation of Java templates of the system from sufficiently refined Event-B specifications;
- generation of JUnit tests from abstract test cases in the Communicating Sequential Processes (CSP) notation.

The organization of the paper is as follows. Section 2 provides necessary background on the modeling and programming languages used in this paper. In Section 3, we look in detail at the scenario-based testing process and present extended guidelines for modeling of Event-B specifications and of testing scenarios along with requirements. Section 4 gives overview of the tools we used for modeling, testing and measuring test coverage. In Section 5, we analyze and discuss the benefits and short-comings of our approach. Section 6, presents some related work in the area of research. Finally, Section 7 concludes the paper.

2. Background

In this section, we give overview of the languages and techniques we use for our scenario-based methodology.

2.1. Overview of Event-B

The Event-B [3] 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. The language of the B-method and Event-B is based on set theory and predicate calculus.

As an example of an Event-B model, consider the following model (also known as machine) *M* with a context *C*. A context is considered as the static part of the Event-B specifications. It contains constants, sets and properties (axioms) related to these. On the other hand, an Event-B machine describes the dynamic part of the specification in the form of events (state transitions).
The context has the following general form.

CONTEXT C
SETS $sets$
CONSTANTS $constants$
AXIOMS $axioms$
END

A context is uniquely defined by its name in the **CONTEXT** clause. The **CONSTANTS** and **SETS** clauses define constants and sets respectively. The **AXIOMS** clause describes the properties of constants and sets in terms of set-theoretic expressions.
An Event-B machine has the following general form.

MACHINE M
SEES C
VARIABLES v
INVARIANT I
EVENTS
$INITIALISATION \ = \ \ldots$
$E_1 \ = \ \ldots$
$\ldots$
$E_N \ = \ \ldots$
END

The machine is uniquely defined by its name in the **MACHINE** clause. The **VARIABLES** 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. In addition, the invariant can define other 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

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

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

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 (or a local variable) and *G(i)* restricts *i*. The occurrence of the events represents the observable behavior of the system. The event guard (e.g., *g* or *G(i)*) defines the condition under which event is enabled.

The occurrence of events represents the observable behavior of the system. The condition under which the action can be executed is defined by the guards. An event is known to be *enabled* when the guards evaluate to *true*. An event execution is supposed to take no time and no two events can occur simultaneously. When some events are enabled, one of them is chosen non-deterministically and its action is executed on the model state. When all events are disabled, i.e. their guards evaluate to false, the discrete system deadlocks. Then previous step is repeated to see if any events are enabled for execution.

The actions of an event can be either a deterministic assignment to the variables of the system or a non-deterministic assignment from a given set or according to a given post-condition. The semantics of actions are defined by their before-after (BA) predicates, where a BA predicate is a relation between *before* and *after* values of the event variables. BA predicates for specific cases of Event-B actions are given in Figure 1.

Action	Before-after (BA) predicate	Explanation
$x := F(x,y)$	$x' = F(x,y) \wedge y' = y$	*standard assignment*
$x :\in Set$	$\exists t.\ (t \in Set \wedge x' = t) \wedge y' = y$	*non-deterministic assignment from set*
$x :\mid P(x,y,x')$	$\exists t.\ (P(x,y,t) \wedge x' = t) \wedge y' = y$	*non-deterministic assignment by given post-condition*

Figure 1. The actions and before-after predicate

In Figure 1, x and y are disjoint lists of state variables, and x', y' represent their values in the after state. The $F(x,y)$ represents a function that provides a deterministic value for x' while y does not change its value. The Set represents any defined set while $P(x,y,x')$ is a post-condition relating initial values of x and y to the final value x'. The $:\in$ and $:\mid$ represent non-deterministic assignment operators operating on sets and predicates respectively.

To check consistency of an Event B machine, we should verify two types of properties: event feasibility and invariant preservation. Formally,

$$Inv(x,y) \wedge g_e(x,y) \ \Rightarrow\ \exists v'.\ BA_e(x,y,x')$$
$$Inv(x,y) \wedge g_e(x,y) \wedge BA_e(x,y,x') \ \Rightarrow\ Inv(x',y)$$

The main development methodology of Event B is *refinement* – the process of transforming an abstract specification to gradually introduce implementation details while preserving its correctness. Refinement allows us to reduce non-determinism present in an abstract model as well as introduce new concrete variables and events. The connection between the newly introduced variables and the abstract variables that they replace is formally defined in the invariant of the refined model. For a refinement step to be valid, every possible execution of the refined machine must correspond to some execution of the abstract machine.

Further details about modeling and verification in Event-B can be found in [3].

2.2. Overview of Communicating Sequential Processes (CSP)

In the following, we present a brief overview of Communicating Sequential Processes (CSP) [6] which is needed to model scenarios in our approach. In CSP, a system is modeled as a process, which interacts with the environment via a number of events whereas the occurrence of events is atomic.

In CSP, there are two basic processes: $STOP$ is a deadlocked process, and $SKIP$ is the terminating process. The process $a \rightarrow P$ can perform an event a and then behave as P. There are two choice operators used in CSP, namely, *external choice* ($\Box$) and *internal choice* ($\sqcap$) operators. In the case of external choice, $P_1 \Box P_2$, either process P_1 or P_2 is executed based on which event occurs first. On the other hand, the internal choice operator is used to model non-determinism, e.g., $P_1 \sqcap P_2$ can arbitrarily choose to behave as either P_1 or P_2.

The processes can be combined together in parallel or in sequence. For sequential composition, $';'$ operator is used. For instance, $P_1; P_2$ ensures that P_1 process executes before P_2. By parallel composition, we allow processes to interact/communicate with each other through the events they engage in. For parallel composition, $\|$ operator is used, e.g., $P_1 \| P_2$. Further details about CSP operators, its semantics and refinements can be read from [6].

2.3. Unit Testing

Unit testing aims at testing units of the program code, e.g., methods or modules, separately from each other. This kind of testing is performed by writing programming methods (called *unit tests*) that invoke the corresponding implementation methods under test. A unit test provides the needed input to the unit under test and evaluates its output before assigning any verdict about its success or failure. Unit testing ensures that the functionality of individual units are tested before these units are integrated to form a larger system.

In order to facilitate unit testing during the system development, unit testing frameworks have been developed for almost every programming language. A unit testing framework provides helper methods, reporting and debugging features to aid unit testing. In our scenario-based approach, we use Java Unit Testing (JUnit) [7] frameworks.

3. Scenario-based Testing Process

In our scenario-based testing process (Figure 2), the system is specified in a stepwise manner using the Event-B formalism until a *sufficiently refined specification* is obtained. The formal models are refined manually based on a set of guidelines which we will discuss in the following section. The sufficiently refined specification is then used to generate a Java implementation template of the system. The development of the specification is driven by the requirements of the system which are traced throughout the process, including to the generated Java code.

The testing scenarios are gradually developed from requirements. The first abstract scenario is provided by the user. This scenario represents a valid behavior of the abstract model present on the same level of abstraction. In short, we say that the abstract model *conforms to* or formally *satisfies* the abstract scenario. Later on, we refine this abstract scenario along the refinement chain of the system models until a sufficiently detailed scenario is obtained. In fact, this detailed scenario represents an abstract test case.

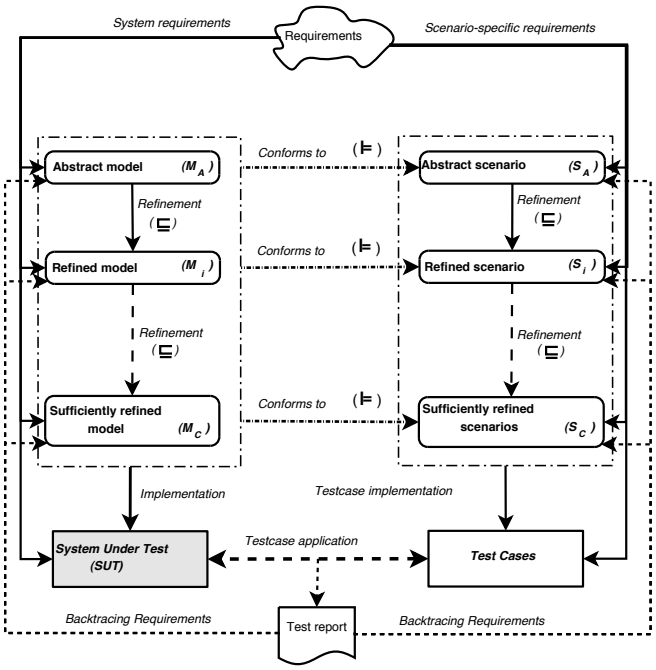

Figure 2. Overview of our scenario-based approach

3.1. Modeling Requirements

The sufficiently refined scenarios are then transformed into executable JUnit test cases, which are executed against the Java implementation. During the process, requirements linked to the testing scenarios are propagated to JUnit tests, where they are used for producing a test report. The approach allows us to evaluate which requirements have been validated and to trace back the failed tests to the formal specifications.

Usually the software systems are built according to informal requirements provided by user. The link between informal requirements and formal models is quite important in software development. The requirements are used for creating the initial specification of the SUT and also for refining this specification on the next level of abstraction.

In our approach, a stand-alone document specifying the requirements of the SUT in a structured manner is used. In this document, the requirements are described using *ID*, *Category*, *Title*, *Priority*, and *Description*, as shown in Figure 3. The hierarchy of the requirements is implemented using the requirement ID. For instance, requirement $\texttt{REQ1}-1$ is a sub-requirement of requirement $\texttt{REQ}-1$.

Throughout this paper we will use excerpts from a *Hotel Booking System*. For the sake of the understanding, we will briefly go through the main functionalities of the system which will be used for exemplification later on in the paper. The four main functional requirements of the system are: the system should allow the user to search for a room in the room database ($\texttt{REQ}-1$), to reserve the room ($\texttt{REQ}-2$), to allow him to pay for the reserved room ($\texttt{REQ}-3$) or to cancel an existing reservation ($\texttt{REQ}-4$).
The requirement $\texttt{REQ}-1$ is described as

$\texttt{Requirement}:\texttt{REQ}-1$
`The system should be able to find a room of given type if it is available in the database and connection to the database is successfully established. In case of failed connection, an exception is reported.`

Each requirement can be divided into several sub-requirements. For instance, ($\texttt{REQ1}-1$) and ($\texttt{REQ1}-2$) are given in the following.

$\texttt{Requirement}:\texttt{REQ1}-1$
`The system should be able to find a room of given type if it is available in the database and connection to the database is successfully established.`

$\texttt{Requirement}:\texttt{REQ1}-2$
`The system should return an error message if connection to the database is not established successfully.`

The requirement ($\texttt{REQ1}-1$), is further divided as

$\texttt{Requirement}:\texttt{REQ1.1}-1$
`The system should be able to accept room type as an input.`

$\texttt{Requirement}:\texttt{REQ1.1}-2$
`The system should be able to connect to the database.`

$\texttt{Requirement}:\texttt{REQ1.1}-3$
`The system should be able to retrieve results.`

These sub-requirements serve as basis for refining the Event-B model.

3.2. Using Event-B for Scenario-based Testing

In our approach, we create formal descriptions of the SUT starting from the requirements as shown in Figure 2. Subsequent refinements of the specification are preformed based on the sub-requirements of a given requirement. In order to be able to generate executable test cases, one needs to have available sufficient information regarding the inputs and outputs of the system. For this purpose, we structure the information about the inputs and outputs based on set of guidelines, following the basic refinement types we suggested in [10]. These basic refinement types are also

Requirement ID	Category	Title	Pr...	D	Description
REQ-1	Functional	FindRoom	1		The system should be able to find a room of given type if it is av
REQ1-1	Functional	FindSuccess	1		The system should be able to accept room type as an input.
REQ1-2	Functional	FindException	1		The system should return an error message if connection to the
REQ1.1-1	Functional	ValidRoomType	1		The system should be able to accept room type as an input.
REQ1.1-2	Functional	ConnectionSuccess	1		The system should be able to connect to the database.
REQ1.1-3	Functional	RetrieveSuccess	1		The system should be able to retrieve results.
REQ-2	Functional	ReserveRoom	1		The system should allow the user to reserve an available room.

Figure 3. Requirements specification excerpt

referred to as *controlled refinements*. The guidelines are used in a similar way for the development of both Event-B models and corresponding user scenarios.

3.2.1. Classification of Events. In order to identify information about the inputs and outputs of the system we classify the Event-B event types into *input*, *output*, and *internal* events, as follows:

Definition 1: 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, accept inputs from user or environment. Apart from their input behavior, these events may take part in the normal functioning of the system. However, the input events do not produce externally visible output. The *output events* ε^O produce externally visible outputs. Finally, the *internal events* do not take part in any input/output activity. These events however, may produce intermediate results used by the events in ε^I and ε^O. The motivation of this classification is explained in next section, where we further divide our system into logical functional units.

3.2.2. Logical Units. As we develop our system in a stepwise manner, the main functional units of a system are already identified at the abstract level. Each of these abstract functional units are modeled as a separate logical unit, called *IOUnit*, in our Event-B models.

Definition 2. 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 an *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 possibly one or more *internal* events in between. Moreover, an *IOUnit* can not contain more than one input or output event.

An *IOUnit* takes input and produces output, as the presence of the input and output events indicates. The classification of events, defined previously, helps us in identifying the inputs and outputs of each unit, and when combined, of the whole system. The motivation for this approach is the following. The developer of the SUT may decide to implement the system independently of the structure of an Event-B model. Indeed, it is sometimes hard to construct 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 interfaces of the model and implementation, i.e., the sequence of the inputs and outputs, should remain the same.

3.2.3. Example. Reserving a room in such the hotel booking system can be modeled as a sequence of events that occur in a specific order. On the abstract level, we may have only a few events, representing some particular functionalities of the system. For example, if we model requirements $\texttt{REQ} - 1$ to $\texttt{REQ} - 4$, each top-level user requirement will be implemented as one IOUnit. Consequently, there are four main IOUnits namely, *Finding* a room, *Reserving* it, *Paying* for it, and *Canceling* a reserved room. After we structure our model according to the guidelines described in Section 3.2.1, the resulting events and their sequence of execution can be seen in Figure 4(a).

As it can be observed, the main functional events are wrapped with the input and output events. For example,

the *Find* event is wrapped around with the *InputForFind* and *OutputForFind* events, where *InputForFind* and *OutputForFind* are the input and output events, respectively.

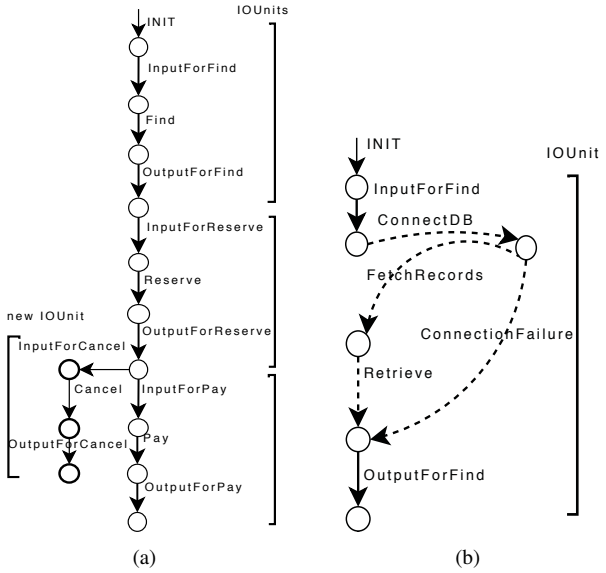

Figure 4. (a) Abstract System (b) Refined System

Within an IOUnit, we treat our main functional events as *internal events* (e.g., *Find, Reserve, Pay* and *Cancel*). Such events can be further refined, in one or more steps, consequently adding more *internal* events within the input-output unit. The refinement is performed according to the sub-requirements of the requirement that was the source of the IOUnit. For instance, the *Find* IOUnit in Figure 4(a) has been refined by applying successive refinements based on the requirement $\texttt{REQ} - 1$ and its sub-requirements, introduced earlier in Section 3.1, into four *internal* events depicted graphically with dashed line pattern in Figure 4(b).

The complete Event-B specifications of this example have been developed and proved using the RODIN [5] platform. In the final refined system, there was a total of 42 proof obligations. Out of these, 38 proof obligations were automatically discharged by the tool, while the remaining 4 needed manual assistance.

3.3. Modeling the Testing Scenarios

As previously mentioned, we use CSP to represent testing scenarios. 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, we can associate these Event-B refinements with syntactic transformations of the corresponding CSP expressions. For instance, the abstract scenario S_A in Figure 2 is refined into scenario S_i, while considering the controlled refinement steps involved in refining the abstract model M_A to the refined model M_i. Similarly, this process continues until we get a sufficiently detailed, concrete testing scenario S_C to which the model M_C conforms.

3.3.1. Testing Scenarios. We define a testing scenario as 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. This means that the scenarios will include the same events as in the corresponding Event-B model. However, the scenarios must follow the same rules that were set for constructing IOUnits in the 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 the sequence without any output event in between them, i.e., the following sequence in a CSP expression is not allowed;
$$< \cdots \rightarrow \varepsilon^I_k \rightarrow \varepsilon^I_{k+1} \rightarrow \cdots >$$
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.
$$< \cdots \rightarrow \varepsilon^O_k \rightarrow \varepsilon^O_{k+1} \rightarrow \cdots >$$

Since the scenarios are defined on the abstract level, they lack details about the system inputs and outputs. The input details can be 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 the inputs can be retrieved from the Event-B model since the model specifies the type, initial value and invariant properties for all variables.

The expected outputs are generated after the model is animated using the 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 then used to compare the values while testing the real implementation. The ProB model checker can only produce 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, we need to define a mapping relation between the abstract and concrete data types. At the moment this mapping is provided manually. However, it is possible to automate its generation for the commonly used types, e.g., boolean and integers.

3.3.2. Example. In the case of the previously discussed *Hotel Booking System* example, there can be many possible testing scenarios. For example, if we want to test the *room*

finding, *reservation* and *paying* functionality, the corresponding abstract scenario expressed as a CSP expression would be as follows.

$$S_{0(A)} = \texttt{InputForFind?roomType} \rightarrow \texttt{Find} \rightarrow \texttt{OutputForFind!(roomId, anyException)} \rightarrow \texttt{InputForReserve?roomId} \rightarrow \texttt{Reserve} \rightarrow \texttt{OutputForReserve!reserveId} \rightarrow \texttt{InputForPay?reserveId} \rightarrow \texttt{Pay} \rightarrow \texttt{OutputForPay!payId} \rightarrow \texttt{SKIP}$$

After a number of successive refinements of event *Find*, we achieve the following scenario. For keeping the example simple, we only show the refinement of *Find* event which is also shown graphically in Figure 4(b).

$$S_0 = \texttt{InputForFind?roomType} \rightarrow \texttt{ConnectDB} \rightarrow ((\texttt{FetchRecords} \rightarrow \texttt{Retrieve}) \sqcap \texttt{ConnectionFailure})) \rightarrow \texttt{OutputForFind!(roomId, anyException)} \rightarrow \texttt{InputForReserve?roomId} \rightarrow \texttt{Reserve} \rightarrow \texttt{OutputForReserve!reserveId} \rightarrow \texttt{InputForPay?reserveId} \rightarrow \texttt{Pay} \rightarrow \texttt{OutputForPay!payId} \rightarrow \texttt{SKIP}$$

The variable `roomType` is the input for this IOUnit, whereas `roomId`, `anyException` are possible outputs. The variable `anyException` specifies if there was any exception, e.g., a connection failure.

Often, the subsequent event depends on the results of the previous ones. For example, the event `Reserve` takes `roomId` as an input from the previous event. It can be noticed that the refinement of the `Find` event has created two branches, one leading to successful case and the other to a database connection failure exception. When the above scenario is checked for conformance with the ProB model checker, it will be found that one can not proceed to `Reserve` if an exception occurred at the previous step. Therefore, this scenario will be split into two scenarios S_0 and S_1 given in the following.

$$S_0 = \texttt{InputForFind?roomType} \rightarrow \texttt{ConnectDB} \rightarrow \texttt{FetchRecords} \rightarrow \texttt{Retrieve} \rightarrow \texttt{OutputForFind!(roomId, anyException)} \rightarrow \texttt{InputForReserve?roomId} \rightarrow \texttt{Reserve} \rightarrow \texttt{OutputForReserve!reserveId} \rightarrow \texttt{InputForPay?reserveId} \rightarrow \texttt{Pay} \rightarrow \texttt{OutputForPay!payId} \rightarrow \texttt{SKIP}$$

$$S_1 = \texttt{InputForFind?roomType} \rightarrow \texttt{ConnectDB} \rightarrow \texttt{ConnectionFailure} \rightarrow \texttt{OutputForFind!(roomId, anyException)} \rightarrow \texttt{SKIP}$$

These scenarios, when sufficiently refined, are transformed into JUnit tests which will be discussed later in Section 3.5.

In the next section, we will discuss how Event-B model is used to generate an implementation template in Java.

3.4. Generating Java Implementation Templates

Once developed, we use the Event-B models of the SUT to generate Java implementation templates. We start by translating a (sufficiently refined) Event-B model into a Java class. As a result, Event-B events are translated to the corresponding Java methods. For our *Hotel Booking System* example, the excerpts of the respective Event-B machine and its implementation template are shown as follows.

MACHINE BookingSystemRef1
REFINES BookingSystem
SEES BookingContext
VARIABLES
roomType
...
INVARIANTS
...
EVENTS
Initialisation
act5 : roomType := Null_roomType
...
Event InputForFind ≙
Refines InputForFind
any
tt
where
grd1 : tt ∈ RTYPES
then
act1 : roomT ype := tt
act2 : inputForFindCompleted := TRUE
end
...
END

An operation in an Event-B specification consists of two parts. The first part contains the pre-condition(s) for the event operation to be enabled, while the second part consists of the actions that the operation performs. For every event in an Event-B model, we create two separate methods in the corresponding Java implementation representing the pre-conditions and actions respectively. The first method, which contains the pre-conditions of an event, returns the evaluation result in the form of a *boolean* value. The name of this method is pre-fixed with the string "guard_". The second method encapsulates the actions of the event. For example, for the *InputForFind* event from our *Hotel Booking System* example, the Java implementation methods are given in the Listing 1. As one can notice, the requirements attached to different IOUnits in Event-B are preserved during the transformation and included in the generated template (see line 19 of Listing 1).

```
1 public class HotelBookingSystem {
2
3     //class-level variables
4     public String roomType;
5     ....
```

```
6
7       public HotelBookingSystem(){
8           //initialization     ...
9       }
10
11      /* PreConditions/Guards for InputForFind event
            */
12      private boolean guard_inputForFind(String
            roomType){
13          return (roomType != null);
14      }
15
16      /* Implementation method for InputForFind
            event*/
17      public boolean inputForFind(String roomType)
18              throws PreConditionViolatedException{
19          //REQ1.1-1
20
21          boolean inputForFindCompleted = false;
22          if(guard_inputForFind(roomType)){
23
24              //actions ...
25
26              this.roomType = roomType;
27              inputForFindCompleted = true;
28          }
29          else{
30              throw new
31              PreConditionViolatedException("For
                    inputForFind");
32          }
33          return inputForFindCompleted;
34      }
35
36      //more Implementation methods for events
37      ....
38  }
39
40  class PreConditionViolatedException extends
        Exception{
41
42      public PreConditionViolatedException(String
            mesg){
43          super(mesg);
44      }
45  }
```

Listing 1. Implementation template example

Each Java implementation method, representing an Event-B event, first evaluates its pre-condition(s) by calling its "guard_" method. If the pre-conditions are evaluated to *false* then the exception `PreConditionViolatedException` is raised, otherwise the actions of the corresponding event are executed. The variables of an Event-B machine are translated into the corresponding class variables in Java. The type information for these variables can be retrieved from the *invariant* clause of the Event-B machine. We assume that a mapping relation between data types in Event-B and Java is provided by the user. For non-primitive data types, Java enumeration (`enum`) type can be used, e.g., to represent a set of finite elements. While most of the Java code can be automatically translated from Event-B constructs, the user can add more code statements according to his/her requirements. This means that the generated class actually constitutes a Java template.

In the next section, we will discuss how testing scenarios are translated into JUnit test cases.

3.5. Generating JUnit test cases from Scenarios

In Section 3.4, we presented the guidelines for generating implementation templates for Java. Once such a template is generated, we can generate the corresponding executable test cases from the scenarios. These test cases are represented as JUnit test methods.

Since our Event-B events are now presented as sequences of *IOUnits*, we write JUnit test cases to test these *IOUnits*. The $Find$ IOUnit from scenario $\mathtt{S}_0$ is represented as an abstract test case $\mathtt{T}_0$ as given in the following.

$$\mathtt{T}_0 = \mathtt{InputForFind?roomType} \rightarrow \mathtt{ConnectDB} \rightarrow \mathtt{FetchRecords} \rightarrow \mathtt{Retrieve} \rightarrow \mathtt{OutputForFind!(roomId, anyException)} \rightarrow \mathtt{SKIP}$$

For scenario $\mathtt{S}_1$, the abstract test case $\mathtt{T}_1$ would be expressed as following.

$$\mathtt{T}_1 = \mathtt{InputForFind?roomType} \rightarrow \mathtt{ConnectDB} \rightarrow \mathtt{ConnectionFailure} \rightarrow \mathtt{OutputForFind!(roomId, anyException)} \rightarrow \mathtt{SKIP}$$

For each of the test cases $\mathtt{T}_0$ and $\mathtt{T}_1$, a separate JUnit test method is implemented. The JUnit test method for $\mathtt{T}_0$ is shown in the Listing 2. In a similar way, JUnit test cases are generated for each IOUnit in the scenario.

```
1  public class HotelBookingSystemTest {
2
3      HotelBookingSystem bSys;
4      ...
5
6      @Before
7      public void setUp() throws Exception {
8          bSys = new HotelBookingSystem();
9      }
10
11     @Test
12     public final void T0(){
13         //REQ1-1
14
15         String roomType = ``Single";
16
17         try{
18             boolean v1,v2,v3,v4,v5;
19             v1 = v2 = v3 = v4 = v5 = false;
20
21             //calling methods of IOUnit
22             v1 = bSys.inputForFind(roomType);
23             v2 = bSys.connectDB();
24             v3 = bSys.fetchRecords();
25             v4 = bSys.retrieve();
26             v5 = bSys.outputForFind();
27
28             //assert statements (verdict)
29             assertTrue(``Successful completion'',
30                 v1 && v2 && v3 && v4 && v5);
31
32             assertTrue(bSys.resultSet.size() > 0);
```

```
33              assertTrue(bSys.anyException == false)
                    ;
34          }
35          catch(PreConditionViolatedException e){
36              fail(e.getMessage());
37          }
38      }
39  }
```

Listing 2. JUnit Test method for T_0

In the test case example shown in Listing 2, there is only one input parameter, i.e., `roomType`. However, in practice, there can be more than one input parameters. Generating all possible values for each parameter and then making all possible combinations of these parameters values may result in combinatorial explosion. In order to handle this problem, the input space partitioning [14] approach is used for test case generation. Information about each input variable is retrieved from the *invariant* clause and the *pre-condition* part of the *input* event. The *pre-conditions* and *invariant* clauses specify the type and possible restrictions (value ranges) for each variable. Using this information, the input space for each parameter is divided into equivalent partitions. Then from each partition, one value is selected to represent the whole partition. Combining the values of different variables from different partitions reduces the total number of input combinations needed for testing.

If a scenario involves multiple *IOUnits* in a sequence and JUnit test case for that sequence is desired, then JUnit test also includes calls to the relevant implementation methods of the the *IOUnit* involved. Moreover, the JUnit assert statements are also appended in the test case.

During the test case generation, the requirements associated to CSP specifications (at this stage called abstract test cases) are propagated to JUnit test cases, as Java comments in the code (see Listing 2–line 13). In addition, each requirement present in the requirement document listed in Figure 3 will be associated with the test cases that covers it. The approach allows one to trace which requirements have been covered and validated during the test execution. The approach will be discussed in more detail in Section 4.

3.6. Backtraceability of Requirements

Once the JUnit tests are run against the SUT a test report is produced. The report will tell which requirements have been covered by the selected set of test cases, which requirements have been left uncovered, and which requirements were not validated. Having the requirements associated to test cases and in the same time to different parts of both Event-B and CSP specifications, allows us to trace at which abstraction level a requirement was introduced and how it reflected in the generated test cases. Based on this analysis, one can identify the source of the error: either in the SUT implementation or an incorrect formalization of the requirement.

4. Tool Support

In this section we will have a brief overview of the tool chain used to support our scenario-based testing process described in Section 3.

Tool support for Event-B modeling and verification is provided by the RODIN platform [5]. RODIN is an Eclipse-based development platform providing effective support for mathematical proof and refinement. The platform is an open-source and further extendible with plugins. RODIN comes along with several useful plugins facilitating smooth and quick development of Event-B specifications. Some of its important features include interactive prover, proof manager, requirement manager and visual modeling. Figure 4 shows a screenshot of RODIN platform with interactive prover.

The consistency of Event B models as well as correctness of refinement steps should be formally demonstrated by discharging *proof obligations*. The RODIN platform automatically generates the required proof obligations and attempts to automatically prove them. Sometimes it requires user assistance by invoking its interactive prover. However, in general the tool achieves high level of automation (usually over 90%) in proving.

The JFeature Eclipse plugin [**?**] ver. 1.2 is used for specifying and managing the requirements of the system. A screenshot was presented in Figure 3. The requirements are specified in the Eclipse GUI and exported to a textual file. The requirements file is then used by the the Requirement plug-in [13] of RODIN for creating associations to Event-B models. In the Requirement plug-in, a parser parses the requirement document and lists individual requirements. Then, any requirement can be selected to be mapped to one or more Event-B elements. This mapping information is stored in a mapping file. Similarly, a separate mapping file is used for storing the mapping between requirements and scenarios. A requirement can be associated with a model element by first selecting a requirement in the requirement manager and then choosing the "Add Association" menu option, which appears after right-clicking the element to be associated. A caption of the Requirement plugin displaying the requirements of the Hotel Booking system is given in Figure 4.

Another important tool in our tool chain is ProB [12] animator and model-checker. Once an initial abstract scenario is provide and expressed in CSP, the generation of the refined testing scenarios is automatic. ProB is used to check conformance between the models and the scenarios. This satisfiability check is performed at each refinement level as was shown earlier in Figure 2. 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. Otherwise, regular expressions could also have served the purpose.

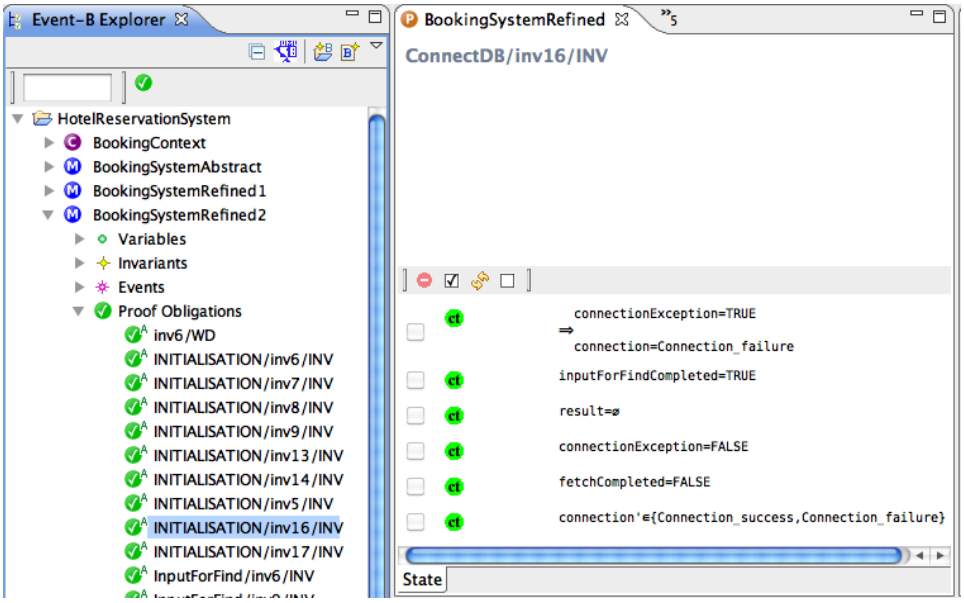

Figure 5. Screenshot of RODIN showing the formal proofs of the Hotel Booking System

Once the JUnit tests are generated they are executed using the JUnit plugin in Eclipse. The code coverage is performed by EclEmma [15], which is a freely available open source Java code coverage tool for Eclipse. With EclEmma, it is also possible to generate the test execution and coverage analysis reports. Figure 4 shows execution and code coverage of a unit test.

As mentioned previously, when the JUnit test cases are generated, the requirement document (listed in Figure 3) is updated such that each requirement is associated with the JUnit tests that cover it. Figure **??** shows a screenshot of the JFeature view in Eclipse. In the top frame, the requirements of the system are linked to one or many JUnit tests. In the left frame, the result of executing the JUnit tests is presented, whereas in the bottom frame, the JFeature test report on how different requirements have been covered by the test execution. In this concrete example, the `TestT1()` test failed, and since it was associated with requirement `REQ1-2 FindException`, the report presents the requirement as broken (with red background). The report presents also the *successful* requirements in green color and the `uncovered` requirements in yellow color.

For back-traceability of requirements for failed test cases, a manual approach is used. Basically, we examine the test reports for failed test cases. With the help of the RODIN Requirement plugin we identify in what parts of the formal specification a requirement was introduced and specified. Alternatively, we examine the Java code of the implantation and debug the code accordingly. Tracing requirements to back to code and formal specifications proved useful in identifying wrongful formalization of requirements or errors in the implementation of the SUT.

5. Discussion

In this section, we analyze our testing approach and discuss some related issues.

The presented test generation process produces test cases in JUnit, which is a well-known and widely used testing framework. The test cases are generated according to the user provided scenarios. More scenarios the user provides, the more code coverage we are likely to achieve. There are several good coverage measuring tools available that can be used with the generated test suites. We have tried EclEmma as described earlier in Section 4.

Furthermore, our approach has the distinguishing advantage that it also accommodates those changes which can not categorized and proved as formal refinement. Referring back to Figure 2, in some cases the model M_i may contain some extra functionalities or features, such as the incorporated fault-tolerance mechanisms, which were omitted or out of scope of the 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 the ProB model checker, to satisfy the model M_i. We can then follow the same refinement process, now starting with $S_i \cup S_{EF}$, until we get a sufficiently refined scenario at level of the final model M_C.

Our approach also describes how one can generate Java implementation templates and the corresponding JUnit test cases. However, if for some reason, the user does not want to use the generated template, s/he can still use the JUnit test generation part to test his/her own implementation, provided that s/he has implemented the system keeping the operation interfaces consistent with the already generated JUnit tests.

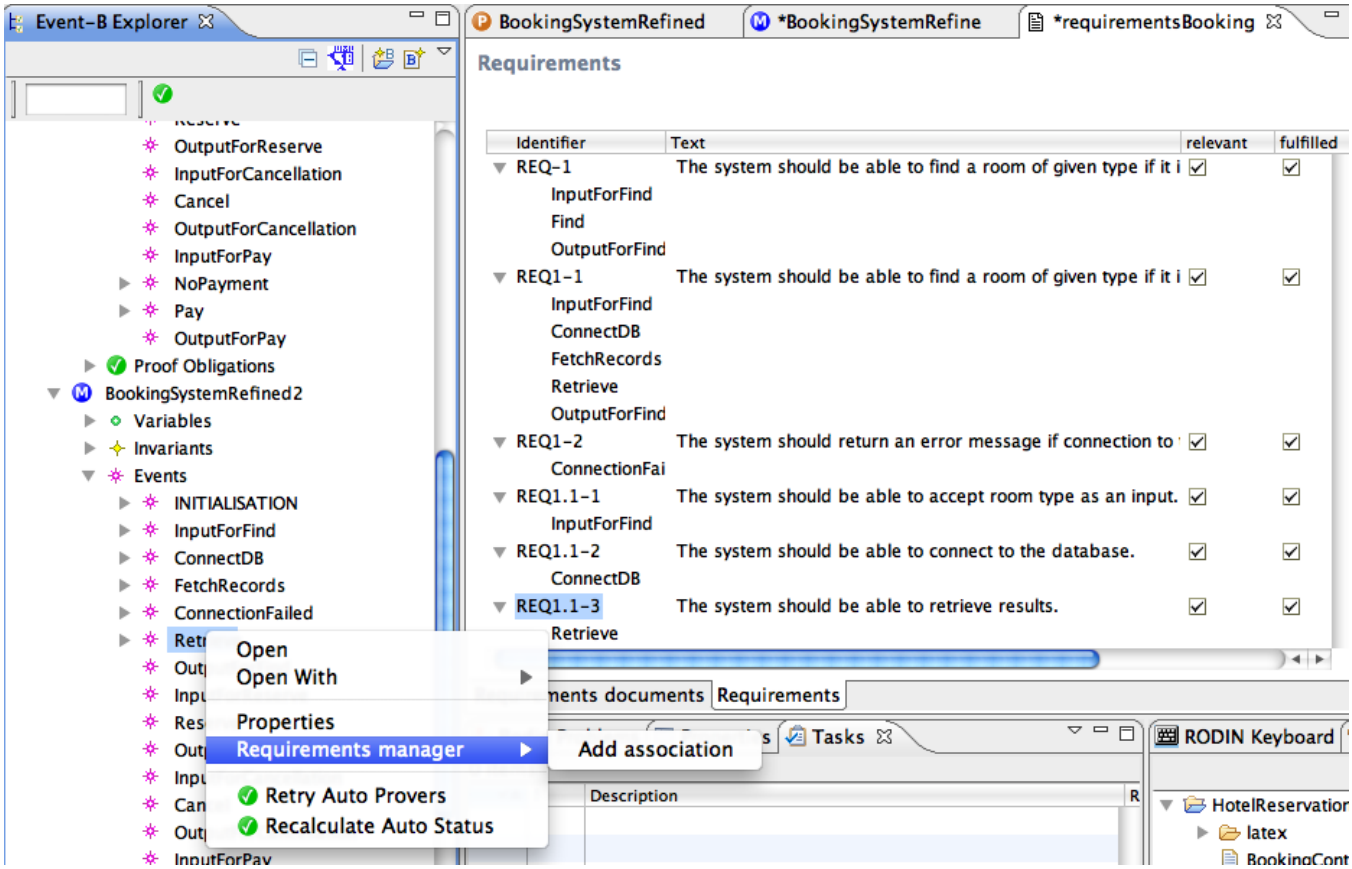

Figure 6. Screenshot of the Requirement plugin in RODIN showing requirements mapping

At the moment, we do not support translation of more complex pre-condition and invariant expressions from Event-B to Java. Namely, the existential and universal quantifiers are not covered. However, this can be achieved by using an approach similar to the one used in JML [16].

We do not explicitly support testing for *negative* scenarios i.e., the behavior that should not exist in the SUT. However, this kind of testing can also be accommodated if we model such *negative* behavior in our Event-B models as events and then provide testing scenarios covering those events. In order to show correctness, the JUnit tests, generated from these negative scenarios, should fail when applied on SUT.

6. Related Work

The jSynoPSys tool [17] performs scenario-based testing using symbolic animation of the B machines. This work defines a scenario-description language used to represent scenarios. However, authors do not provide any guidelines for the refinement of the specifications or scenarios. It is also not mentioned how scenarios will be transformed into executable test cases.

Nogueira et al. in [18] present a test generation approach based on the CSP formalism. The CSP models are constructed from use cases described in a pre-defined subset of natural language. The test scenarios are then incrementally generated as counter-examples for refinement verifications using a model checker. The main difference between their work and our approach is that we use Event-B to represent our system models and use CSP to represent testing scenarios. A model checker in our case is used to check the conformance between models and scenarios.

Stotts et al. in [19] describe a JUnit test generation scheme based on the algebraic semantics of Abstract Data Types (ADTs). The developer codes ADT in Java, while tests are generated for each ADT axiom. One of the advantages of this approach is that the formalism is hidden and the developer only needs to know Java to use this method. However, unlike our approach, in their case it would not be possible to mathematically prove any safety properties or to find deadlocks in the specifications.

In our earlier work [20], we presented the scenario-based testing approach for B models, where we designed an algorithm for constructing test sequences across different refinement [21] models. However, this algorithm is exponential in its nature thus limiting its practical applicability.

In our current approach, ordering of events are enforced

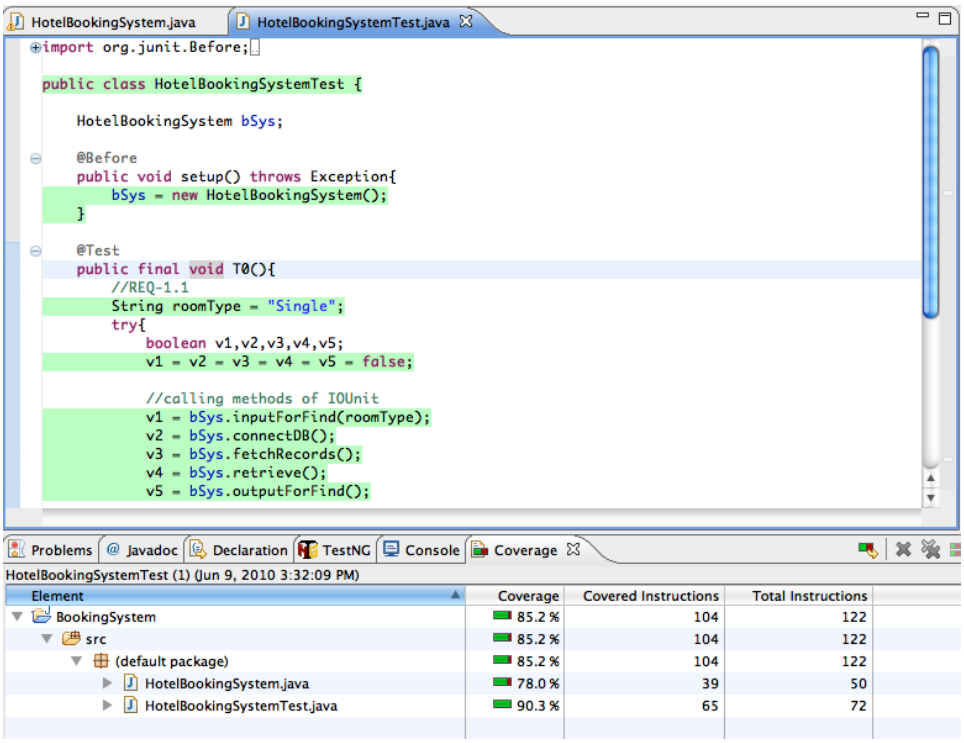

Figure 7. Executing Unit Tests and Measuring Code Coverage

by the guards and actions of the events. In [22], Iliasov has proposed a support of control flow as an explicit event ordering mechanism for Event-B models. The control flow of events resemble the notion of scenarios used by our approach. The difference between the two approaches is that we use a model-checker and an animator to verify existence of these scenarios, whereas in [22], the additional proof obligations are generated and then proved by a theorem prover. The control flow approach can be used as an alternative to the scenario-conformance steps, shown in Figure 2.

7. Conclusion 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 executable JUnit test cases. Optionally, user can map informal requirements to the formal model and testing scenarios at different refinement steps. This mapping of informal requirements is extended till concrete test cases so that upon test case failure, these unfulfilled requirements can be back-traced into the model.

We believe that our approach is very scalable. It can help developers and testers to automatically generate large number of executable test cases. Generating these test case by hand would be very laborious and error-prone process.

As future work, we aim at providing graphical representation for the testing scenarios and their refinements. Moreover, at the moment, the mapping between abstract and concrete data types needs to be provided manually by the user. An automatic translation would be very helpful and time-saving in this respect.

In addition to that, we also intend to use the UML-B [23] formalism as the main modeling language in our scenario-based testing approach. UML-B is a new graphical language, which combines certain UML features with Event-B. UML-B is similar to UML but has its own meta model. The main advantages of using UML-B is that it provides an UML-like front-end to Event-B, which make the modeling language familiar to the majority of the developers. Moreover, it provides additional structuring of Event-B models in the form of UML classes and state-machines.

Another direction for our future work is to use the Next Generation Java Testing (TestNG) framework [8] for implementing the test cases. TestNG provides some important extensions to JUnit 4 framework, e.g., parameterized test methods.

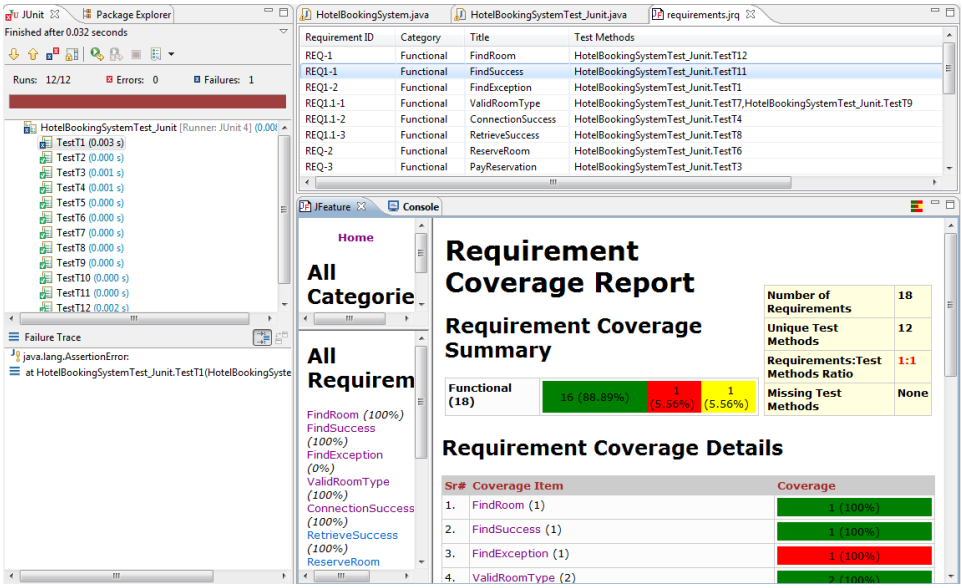

Figure 8. Requirement coverage report in JFeature.

References

[1] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria, "Software engineering and formal methods," *Commun. ACM*, vol. 51, no. 9, pp. 54–59, 2008.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing*. Morgan Kaufmann Publishers, 2006.

[3] J. R. Abrial, *Modeling in Event-B: System and Software Design*. Cambridge University Press, 2010.

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

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

[6] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[7] "JUnit 4," http://www.junit.org.

[8] C. Beust and H. Suleiman, *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley, 2007, http://www.testng.org/.

[9] Q. A. Malik, J. Lilius, and L. Laibinis, "Scenario-Based Test Case Generation Using Event-B Models," in *International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)*. IEEE Computer Society, 2009, pp. 31–37.

[10] Q. A. Malik, J. Lilius, and L. Laibinis, "Model-Based Testing Using Scenarios and Event-B Refinements," in *Methods, Models and Tools for Fault Tolerance, LNCS Vol. 5454*. Springer-Verlag, 2009, pp. 177–195.

[11] A. Roscoe, *The Theory and Practice of Concurrency*. Prentice Hall, 1998 amended 2005.

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

[13] "Requirement Management Plug-in for Rodin Platform," home page : http://wiki.event-b.org/index.php/Category:Requirement_Plugin.

[14] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[15] "EclEmma - Java Code Coverage for Eclipse," http://www.eclemma.org/.

[16] G. T. Leavens and A. L. Baker, "Enhancing the Pre- and Postcondition Technique for More Expressive Specifications," in *In FM99: World Congress on Formal Methods*. Springer, 1999, pp. 1087–1106.

[17] F. Dadeau and R. Tissot, "jSynoPSys – A Scenario-Based Testing Tool based on the Symbolic Animation of B Machines," *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 2, pp. 117–132, 2009.

[18] S. Nogueira, A. Sampaio, and A. Mota, "Guided Test Generation from CSP Models," in *ICTAC*, 2008, pp. 258–273.

[19] P. D. Stotts, M. Lindsey, and A. Antley, "An Informal Formal Method for Systematic JUnit Test Case Generation," in *XP/Agile Universe*, 2002, pp. 131–143.

[20] M. Satpathy, Q. A. Malik, and J. Lilius, "Synthesis of Scenario Based Test Cases from B Models." in *FATES/RV*, 2006, pp. 133–147.

[21] R.-J. Back and J. von Wright, "Refinement Calculus, Part I: Sequential Nondeterministic Programs," in *REX Workshop*, 1989, pp. 42–66.

[22] A. Iliasov, "On Event-B and Control Flow," *Technical Report in DEPLOY Project*, 2010.

[23] C. Snook and M. Butler, "UML-B: Formal Modeling and Design Aided by UML," *ACM Transaction on Software Engineering Methodologies*, vol. 15, no. 1, pp. 92–122, 2006.