Model-Based Testing Using Scenarios and Event-B Refinements

Qaisar A. Malik, Johan Lilius, and Linas Laibinis

Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science (TUCS), Finland
{Qaisar.Malik,Johan.Lilius,Linas.Laibinis}@abo.fi

Abstract. In this paper, we present a model-based testing approach based on user provided testing scenarios. In this approach, when a software model is refined to add or modify features, the corresponding testing scenarios are automatically refined to incorporate these changes. The test cases, to be applied on the system under test, are generated from these scenarios. We use the Event-B formalism for software models, while user scenarios are represented as Communicating Sequential Process (CSP) expressions. The presented case study demonstrates how our approach can be used to test different features of a system such as incorporated fault-tolerance mechanisms.

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 [9]. In the model-based development (MBD), this model is then refined until a required abstraction level is reached from which the implementation code can be generated, or written by hand. Model-based testing (MBT) is an approach for deriving tests from the models using automated techniques. The intended cost reductions arise because

1. the tests can be generated by tools, without hand coding,
2. changes in the model do not need extensive re-writing of test-code,
3. test coverability can be improved because we can define coverability on the model and guarantee that important parts of the system are tested.

Test selection is an active research area where both formal and informal approaches exist. In this paper, we propose a testing approach based on user-provided testing scenarios. In the Model-Based development (MBD), we start with the initial model that is created to implement the set of use cases explaining the desired behavior. The use cases are usually described in natural language and

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 177–195, 2009.

are used by requirement engineers to represent a part of the requirements given by customers. Often these use cases also form the basis for the acceptance tests of the final product. However, there is an abstraction gap between the use cases and the final acceptance tests. In order to support automatic generation of tests from given software models, we need to bridge this gap. In this paper, we study this issue in the context of formal MBD by using Event-B [5,4] as our modelling language. Event-B supports stepwise system development by refinement. We will represent formal models of software systems as Event-B specifications. On the other hand, we express the provided use-cases as scenarios in Communicating Sequential Process (CSP) [10]. Representing scenarios as CSP expressions gives us better structure and associated tool support. We show, by making *controlled* refinements of our Event-B models, that we can automatically derive the final tests from the original scenarios. An overview of the approach is given is given in Fig. 1. This work is based on our earlier approach [16] for scenario-based testing from B models.

The program refinement approach has been extensively used to model complex software systems, such as control systems, communication systems etc. It allows us to gradually incorporate implementation details and therefore, helps us to deal with overall complexity. For such systems, it is really important to be dependable i.e., to function even in the presence of faults and failures. The refinement approach allows us to gradually incorporate fault tolerance mechanisms describing how the system reacts on abnormal situations. In [11], a methodology for developing fault-tolerant systems in a stepwise manner is proposed. Our work, presented in this paper, is also based on stepwise development and thus very suitable for testing the desired properties and features of the systems that were developed in such a way.

The organisation of the paper is as follows. Section 2 gives brief introduction to Model-based testing (MBT) and details our scenario-based MBT methodology. In Section 3, we describe the formal framework for our approach and show how it can be instantiated in specific cases. In Section 4, we illustrate our approach by a case-study on the development of a fault-tolerant system. Finally, Section 5 contains related work and some concluding remarks.

2 Model-Based Testing

Model-based testing is a general notion used for testing of software systems using models of the system. In [17], model-based testing is defined as *automation of the design of black-box tests*. The system to be tested is referred as *System-Under-Test* (SUT). The SUT is an executable implementation which is considered as a black-box during the testing process, i.e., only inputs and outputs of the system are visible externally. The SUT is tested by applying *test case(s)*. A *test case* is defined as sequence of steps to test the correct behavior of a particular functionality or feature of the system [2]. In model-based testing, the test cases are generated from the given models of the system.

2.1 Scenario-Based Approach for Model-Based Testing

Our model-based testing approach is based on stepwise system development [6] using behavioral models of the system. By a behavioral model, we mean that the system behavior is modelled as states together with operations (or events) on the states. 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. These models are usually created from the requirements.

In the development process, we start with an abstract model and gradually, given by a number of refinement steps, obtain a sufficiently detailed model. The final system, the system under test (SUT), is an implementation of this detailed model. Ideally, the implementation should be automatically generated, which would make it correct by construction. However, in practice, due to the abstraction gap between formal models and executable implementations, this is not always possible. As a result, an implementation is often hand-coded while consulting with the formal models. The left hand-side of the Fig.1 graphically presents this process.

The right hand side of the Fig.1 depicts a parallel process where we start from the requirements and construct an abstract scenario. This abstract scenario is a valid behavior of the abstract model present on the same level of abstraction. In short, we say that the abstract model *conforms* to the abstract scenario. In later stages, we refine our abstract scenario along the refinement chain of the system

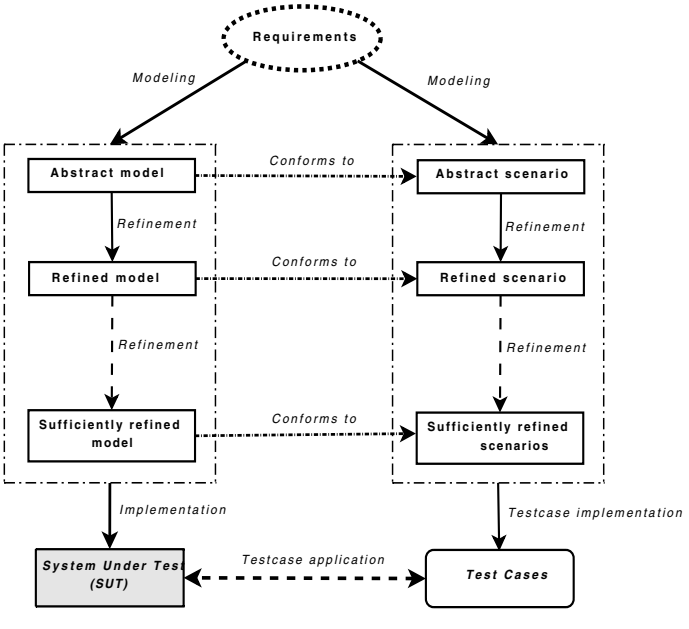


Fig. 1. Overview of our Model-based testing approach

models. Finally, the sufficiently refined scenarios are translated into executable test cases which are then applied to SUT. Later in this section, we provide more detailed information about definitions and representation of the scenarios.

In the literature, one can find several definitions of the term *scenario*. Generally speaking, as described in [1], a scenario is a description of possible actions and events in the future. In the field of software engineering, scenarios have been used to represent various concepts like system requirements, analysis, user-component interactions, test cases etc. [13]. In this work, we use the term *scenario* to represent a *test scenario* for our system under test (SUT). A test scenario is one of the possible valid execution paths that the system must follow. In other words, it is one of the expected functionalities of the system. For example, in a hotel reservation system, booking a room is one functionality, while canceling a pre-booked room is another one. In this article, we use both terms functionality and scenario interchangeably.

Each scenario usually includes more than one system-level event or procedure, which are executed in some particular sequence. Since, in a non-trivial system, there can be many possible execution sequences of the events, identifying all of the valid sequences may not be an easy task. In our approach, we deal with this complexity in a stepwise manner. On the abstract level, an initial scenario is provided by the user. The abstract model of the system *conforms to* or formally *satisfies* this scenario, meaning that the scenario is in fact a the valid behavior of the model. This scenario is a sequence of the abstract event(s) in their order of execution. Once an abstract scenario has been provided, afterwards, for each refinement step scenarios are refined automatically. Fig.2 shows the refinement process where an abstract model M_i is refined by M_{i+1} (denoted by $M_i \sqsubseteq_c M_{i+1}$). This refinement ($\sqsubseteq_c$) is a controlled refinement as will be discussed in detail in section 3.2. Scenario S_i is an abstract scenario, formally satisfiable ($\models$) by specification model M_i, is provided by the user. In the next refinement step, scenario S_{i+1} is constructed automatically from M_i, M_{i+1} and S_i in such a way that S_{i+1} is formally satisfied or conformed by the model M_{i+1}. The automatically generated scenario S_{i+1} represents functionalities, in part or whole, of the model M_{i+1}.

In some cases, the model M_{i+1} may contain some extra functionalities or features, such as incorporated fault-tolerance mechanisms, which were omitted or out of scope of scenario S_i. These *extra features*, denoted by S_{EF}, can be added in the scenario S_{i+1} manually. The modified scenario $S_{i+1} \cup S_{EF}$ must be checked (by means of available tools) to be satisfied/conformed by the model M_{i+1}. We can follow the same refinement process, now starting with $S_{i+1} \cup S_{EF}$, until we get S_{i+n}.

After the final refinement, the system is implemented from the model M_{i+n}. This implementation is called *system under test (SUT)*. The scenario S_{i+n} is unfolded into the executable test cases that are then applied to SUT. In the section 3, we will demonstrate how scenarios are represented and refined. In the next section, we present some mathematical preliminaries for our model-based testing approach.

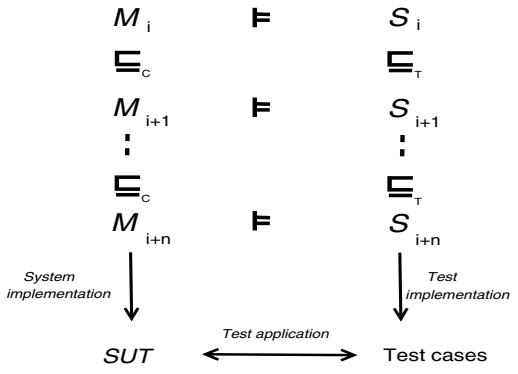


Fig. 2. Refinement of Models and Scenarios

2.2 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 a label transition system with $s, s^{'}$ in S and let $\mu_i \in L$.

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

behavior of 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

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

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 LTS *l*,

$$ts \subseteq beh(l) \qquad \square$$

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.

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

3 Formal Framework

3.1 Modeling in Event-B

The Event-B [5,4] is a recent extension of the classical B method [3] 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*
SETS *TYPES*
VARIABLES *v*
INVARIANT *I*
INITIALISATION *INIT*
EVENTS
$E_1 = \ldots$
$\ldots$
$E_N = \ldots$
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** clause. 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 [8]. 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:

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

An Event-B machine describes a state-machine that represents particular behavior of the machine M. We can describe it as an instantiation of labelled transition system, defined in Section 2.2.

Definition 4
An Event-B machine denotes a labelled transition system $\langle S, L, T, s_0 \rangle$ where

- S is set of Event-B *states*, where the state of an Event-B machine is a particular assignment of values to the Event-B variables;
- L is a set of event names;
- the transition relation T is constructed from single transitions of the form

$$s \xrightarrow{ev} s'$$

  where s and s' are Event-B states and ev is the name of an event.
- s_0 is the state of an Event-B machine after initialization. □

3.2 Refinement of Event-Based Systems

The basic idea underlying the formal stepwise development is to design a system implementation gradually, by a number of correctness preserving steps, called *refinements*. The refinement process starts from creating an abstract, albeit implementable, specification and finishes with generating executable code. In general, the refinement process can be seen as a way to reduce non-determinism of

the abstract specification, to replace abstract mathematical data structures by data structures implementable on a computer, and, hence, gradually introduce implementation decisions.

We are interested in how refinement affects the external behavior of a system under construction. Such external behavior can be represented as a trace of observable events, which then can be used to produce test cases. From this point of view, we can distinguish two different types of refinement called *atomicity* refinement and *superposition* refinement.

In ***Atomicity*** refinement, 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. Let us consider an abstract machine AM_A and a refinement machine AM_AR given below. It can be observed that an abstract event E is split (replaced) by the refined events *E1* and *E2*. Any execution of *E1* and *E2* will correspond to some execution of abstract event *E1*. It is also shown graphically in Fig.3(a).

	REFINEMENT *AM_AR*
MACHINE *AM_A*	**REFINES** *AM_A*
...	...
EVENTS	**EVENTS**
E = **WHEN** g	E_1 **ref** E = **WHEN** $g \wedge g_1$ **THEN** S_1 **END**
THEN S **END**	E_2 **ref** E = **WHEN** $g \wedge g_2$ **THEN** S_2 **END**
END	**END**

In ***Superposition*** refinement, 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 not affect the variables of the abstract specification and only define computations on newly introduced variables. For our purposes, it is convenient to further distinguish two basic kinds of superposition refinement, where

- a non-looping event is introduced,
- a looping but terminating event is introduced.

Let us consider an abstract machine AM_S and a refinement machine AM_SR as shown below

	REFINEMENT *AM_SR*
MACHINE *AM_S*	**REFINES** *AM_S*
...	...
EVENTS	**EVENTS**
E = **WHEN** g	E = **WHEN** g **THEN** S **END**
THEN S **END**	E_1 = **WHEN** g_1 **THEN** S_1 **END**
END	**END**

It can be observed that the refined specification contains both the old and the new events, E and *E1* respectively. To ensure termination of the new event(s), the **VARIANT** clause is added in a refinement machine. This **VARIANT** clause contains an expression over a well-founded type (e.g., natural numbers). The new events should decrease the value of the variant, thus guaranteeing that the new events will eventually return the control as the variant expression can not be decreased indefinitely. These two types of refinements are also shown graphically in Fig.3(b) and (c).

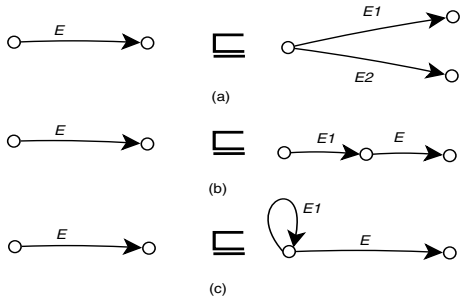


Fig. 3. Basic refinement transformations

Let us note that the presented set of refined types is by no means complete. However, it is sufficient for our approach based on user defined scenarios.

The event-based development gradually (in a controlled way) reveals more observable states of the system. Even if the idea of controlled refinement is generic, the inspiration for it came from [11], where a fault-tolerant agent system was developed, gradually incorporating fault-tolerance mechanisms. In this way, both normal and abnormal functionalities (faults/failures) can be modelled. To model abnormal behavior, the system reactions in the form of specific fault-tolerance mechanisms are added to the refined models. These mechanisms include modelling of special degraded states, recovery actions, failure modes and so on. In the development of safety/security-critical or communication systems, handling of such events often contains the most of the system's complexity.

3.3 Scenario Refinement and Representation

In section 2.1, we introduced the notion of scenarios. Each such scenario can be represented as a Communicating Sequential Process (CSP) [10] expression. Since we develop our system in a controlled way, i.e. using basic refinement transformations described in Section 3.2, 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 ProB [12] model checker. ProB supports execution (animation) of Event-B specifications, guided by CSP expressions. The satisfiability check is performed at each refinement level as shown in the Fig.2. The refinement of scenario S_i is the CSP trace-refinement [14] denoted by $\sqsubseteq_T$.

As we have described before, the scenarios are represented as CSP expressions. We refine our models in a controlled way targeting at individual events. We assume that the events are only executed when their guards are enabled. For simplicity, we omit the guard information from CSP expressions. Here we will discuss how individual refinement steps affect the scenarios. Let us assume we are given an abstract specification M_0 with three events, namely, `A`, `B` and `C`, and

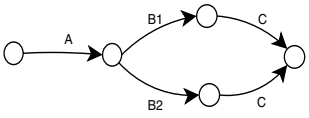


Fig. 4. Atomicity Refinement

a scenario S_0 representing the execution order of these events: first the event A, then the event B, and finally the event C. The CSP expression for scenario S_0 is given by

$$\mathtt{S_0} = \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{C} \rightarrow \mathtt{SKIP}$$

In the next refinement step, the model M_0 is refined by M_1. This refinement step may involve any of three types of the supported refinements, as discussed in Section 3.2. In order to reflect the changes from the refined model into scenarios, we need to update/refine our CSP expressions accordingly. We will discuss the scenario refinement step one by one in the following.

Atomicity Refinement. Let us suppose an event B is refined using atomicity refinement. As a result, it is split into two events namely $\mathtt{B_1}$ and $\mathtt{B_2}$. It means that the older event B will be replaced by two new events $\mathtt{B_1}$ and $\mathtt{B_2}$ modelling a branching in the control flow. As a CSP expression we can represent the new refined scenario S_1 as

$$\begin{gathered}\mathtt{S_1} = \mathtt{A} \rightarrow ((\mathtt{B_1} \rightarrow \mathtt{CONT}) \sqcap (\mathtt{B_2} \rightarrow \mathtt{CONT})) \\ \mathtt{CONT} = \mathtt{C} \rightarrow \mathtt{SKIP}\end{gathered}$$

where $\sqcap$ is an internal choice operator in CSP. We use internal choice operator instead of external one because all of the events in our system have guards which are strengthened in a way that at a time only one event is enabled for execution. This can also be represented graphically as shown in Fig. 4

Superposition refinement. Let us suppose we use superposition refinement to refine an event C. As a result, a new non-looping event D is introduced in the system.

The new scenario S_1 is represented as a CSP expression in the following:

$$\mathtt{S_1} = \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{D} \rightarrow \mathtt{C} \rightarrow \mathtt{SKIP}$$

In the second case, let us suppose we again use superposition refinement to refine event C. However, this time a new looping event D is introduced into the system. The corresponding CSP expression is given as

$$\mathtt{S_1} = \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{D} \rightarrow \mathtt{C} \rightarrow \mathtt{SKIP}$$

where D is defined as

$$\mathtt{D} = \mathtt{D} \sqcap \mathtt{SKIP}$$

The new scenario can also be represented graphically as Fig. 5.

In the next section, we outline how scenarios are unfolded into test cases.

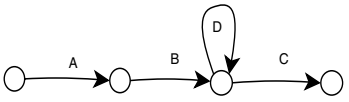


Fig. 5. Superposition refinement type II

3.4 Instantiation of Scenarios as Test Cases

Now we need to translate a scenario into a test case. The distinction between the two is the following. For a scenario, we are *mathematically* guaranteed that the model will conform to the scenario and it can be checked, e.g., by model-checking. For the SUT, we clearly can not give any such guarantee. Thus for each event in the scenario we need to check that SUT has executed the corresponding action correctly. For our approach, we use the ProB model checker, which has the functionality to animate B specifications guided by the provided CSP expression. After the execution of each event, present in the scenario, information about the changed system state is stored.

In other words, the execution trace is represented by a sequence of pairs $< e, s' >$, where e is an event and s' is a post-state (the state after execution of event e). From now on we will refer to a single pair $< e, s' >$ as an *ESPair*.

For a finite number of events $e_1, e_2.....e_n$, present both in the model M and the System Under Test (SUT), a test case t of length n, defined in terms of test a sequence in section 2.2, consists of an initial state *INIT* and a sequence of *ESPairs*

$$t = INIT, \{< e_1, s'_1 >, < e_2, s'_2 >, ... < e_n, s'_n >\}$$

Similarly, a scenario, as formally defined in section 2.2 as finite set of related test cases, i.e., a scenario ts is given as

$$ts = \{t_1, t_2, .., t_n\}$$

As mentioned earlier, *ESPair* relates an event with its post-state. This information is stored during test-case generation. For SUT these stored post-states become expected outputs of the system and act as a *verdict* for the testing. After execution of each event, the expected output is compared with the output of the SUT. This comparison is done with the help of probing functions. The probing functions are such functions of SUT that at a given point of their invocation, return state of the SUT. For a test-case to pass the test, each output should match the expected output of the respective event. Otherwise, we conclude that a test case has failed. In the same way, test cases from any refinement step can be used to test implementation as long as both the implementation and the respective test cases share the same events and signatures.

4 Testing Development of a Fault-Tolerant System

In this section we will demonstrate our approach on a case study development of a fault tolerant agent system. The case study was first presented in [11]. Agent

systems are examples of complex distributed systems. Though agents operate in unreliable communication environment, often such systems have high reliability requirements imposed on them. Thus, ability to operate in a volatile error prone environment and have regularly to cope with abnormal situations that are typical for agent systems is the essential requirement for designing such systems. The development of such systems should also facilitate systematic integration of the fault tolerance mechanisms into agent applications.

The most typical faults that these applications encounter are temporal connectivity losses, which can cause failures of communication between cooperating agents or between an agent and the server. In [11], the agent and server software are developed from the corresponding B specifications, where the fault tolerance features are gradually integrated into these specifications. Hence, the development of the fault-tolerance mechanisms becomes a part of the system development.

For example, while modelling collaboration of agents, we have to define the agent behavior in the presence of message losses, hardware failures, etc. Generally speaking, fault tolerance in agent systems is supported by a set of abstractions used by the application developers and a specialised middleware. The abstractions are developed to systematically separate the normal system behavior from the abnormal one. The middleware detects disconnections and, when necessary, involves agents into error recovery.

The formal development, presented in [11], is used in this section to demonstrate our model based testing approach. The main refinement steps introducing the abnormal system behavior (disconnections, hardware failures) and the corresponding system reactions (recovery actions, aborting) are reflected in the corresponding test scenarios.

We start our development with a very simple specification of a mobile agent system, where an agent performs three basic tasks when connected to the server. These basic tasks are named as *Engage*, *NormalActivity* and *Disengage*. To incorporate the fault-tolerant behavior, the system is repeatedly refined using the basic refinement types described in Section 3.3. The introduction of fault-tolerance increases the complexity of the system. Our testing methodology can be applied to test the new scenarios that result from this complexity. The initial *Event-B* machine named *AgentSystem* specifies the three basic events, mentioned above.

MACHINE *AgentSystem*
SETS *Agents*
VARIABLES *agents*
INVARIANT $agents \subseteq Agents$
INITIALISATION $agents := \emptyset$
EVENTS
Engage = **ANY** aa **WHERE** $aa \in Agents \wedge aa \notin agents$
THEN $agents := agents \cup \{aa\}$ **END**;
NormalActivity = **ANY** aa **WHERE** $aa \in Agents \wedge aa \in agents$
THEN skip END ;

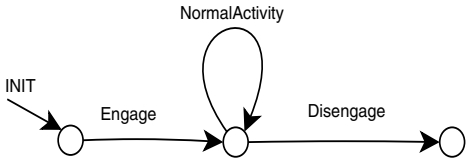


Fig. 6. Execution graph of machine *AgentSystem*

Disengage = **ANY** *aa* **WHERE** $aa \in Agents \wedge aa \in agents$
THEN $agents := agents - \{aa\}$ **END**
END

In the specification *AgentSystem*, let us note that the event *NormalActivity* may happen zero or more times. The sequence of events, as determined by the specification, is shown in Fig.6. The INIT is an initialisation event.
The given scenario can be expressed as the following Communicating Sequential Process (CSP) expression

```
AgentSystem  = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = Disengage -> SKIP
```

In the next refinement machine *AgentSystem1*, the event *Disengage* is refined into two new events in order to differentiate between leaving normally or because of a failure. This refinement step is *atomicity* refinement as discussed in Section 3.3. The other events of the specification remain the same. The execution graph for this refinement is shown in Fig.7.

REFINEMENT *AgentSystem1* **REFINES** *AgentSystem*
...
EVENTS
...
NormalLeaving ref Disengage = **ANY** *aa* **WHERE** $aa \in Agents \wedge aa \in agents$
THEN $agents := agents - \{aa\}$ **END**
Failure ref Disengage = **ANY** *aa* **WHERE** $aa \in Agents \wedge aa \in agents$
THEN $agents := agents - \{aa\}$ **END**
END

The testing scenario for *AgentSystem1* is expressed as the following CSP expression

```
AgentSystem1 = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = NormalLeaving -> SKIP
Node1 = Failure -> SKIP
```

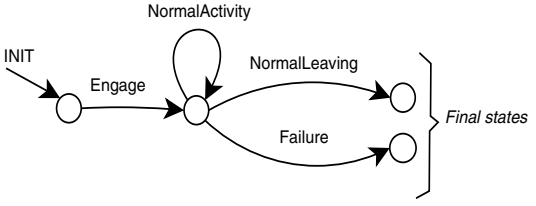


Fig. 7. Execution graph of machine *AgentSystem*1

In the next refinement machine *AgentSystem2*, we introduce temporary loss of connection for our agents. This new event is called *TempFailure*. This refinement step introduces a looping event (see superposition refinement in Section 3.3). To guarantee termination of the new event, we introduce a new variable *disconn_limit*, which is used as a variant.

REFINEMENT *AgentSystem2* **REFINES** *AgentSystem1*
...
VARIABLES *agents*, *disconn_limit*
INVARIANT *disconn_limit* ∈ **NAT**
VARIANT *disconn_limit*
EVENTS
...
NormalActivity = **ANY** *aa* **WHERE** *aa* ∈ *agents*
THEN *disconn_limit* := *Disconn_limit* **END**;
TempFailure = **ANY** *aa* **WHERE** (*aa* ∈ *agents)*
THEN *disconn_limit* := *disconn_limit* - 1 **END**;
END

The execution flow for *AgentSystem2* is given in Fig.8. The CSP expression for the refined scenario is given as

```
AgentSystem2  = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = TempFailure -> Node1
Node1 = NormalLeaving -> SKIP
Node1 = Failure -> SKIP
```

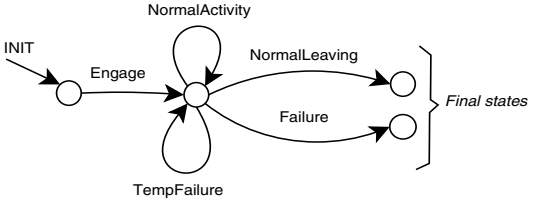


Fig. 8. Execution graph of machine *AgentSystem*2

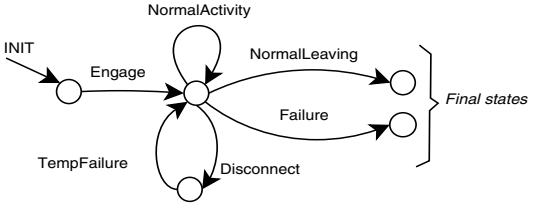


Fig. 9. Execution graph of machine *AgentSystem*3

In next refinement machine *AgentSystem3*, a new event *Disconnect* is introduced. It is the event that precedes (causes) *TempFailure* event. This refinement is a superposition refinement introducing a non-looping event. A new variable *timers* is used to ensure order of execution.

REFINEMENT *AgentSystem3* **REFINES** *AgentSystem2*

. . .

EVENTS

. . .

Disconnect = **ANY** *aa* **WHERE** $aa \in agents$

THEN $timers := timers \cup \{aa\}$ **END**

TempFailure = **ANY** *aa* **WHERE** $(aa \in agents) \wedge (aa \in timers)$

THEN $disconn\_limit := disconn\_limit - 1 \parallel timers := timers - \{aa\}$ **END**;

END

The execution flow for *AgentSystem*3 is shown in Fig.9. The refined scenario is represented as following CSP expression

```
AgentSystem3  = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = Disconnect -> TempFailure -> Node1
Node1 = NormalLeaving -> SKIP
Node1 = Failure -> SKIP
```

In the final refinement step, we elaborate on error recovery and time expiration by splitting the events *TempFailure* and *Failure* by atomicity refinement.

REFINEMENT *AgentSystem4* **REFINES** *AgentSystem3*

. . .

EVENTS

. . .

TimerExpiration ref Failure = **ANY** *aa* **WHERE**

$(aa \in agents) \wedge (aa \in ex\_agents)$

THEN $agents := agents - \{aa\} \parallel ex\_agents := ex\_agents - \{aa\}$ **END**;

AgentFailure ref Failure = **ANY** *aa* **WHERE**

$(aa \in agents) \wedge (aa \notin timers) \wedge (aa \notin ex\_agents)$

THEN $agents := agents - \{aa\}$ **END**;

Connect ref TempFailure = **ANY** *aa* **WHERE** $(aa \in agents) \wedge (aa \in timers)$

THEN $disconn\_limit := disconn\_limit - 1 \parallel timers := timers - \{aa\}$ **END**;

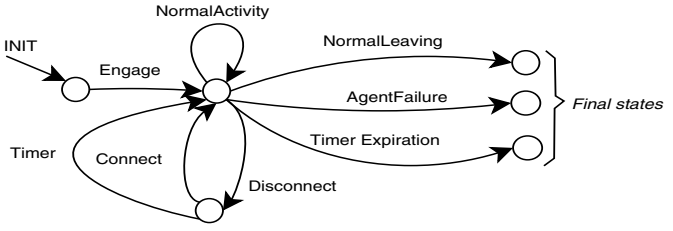
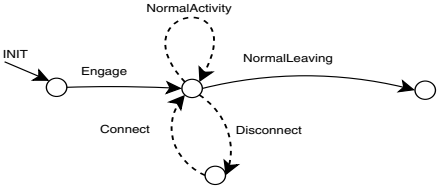
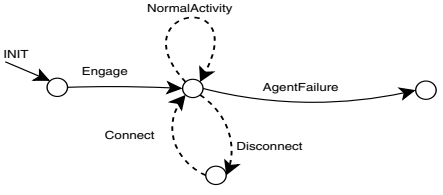


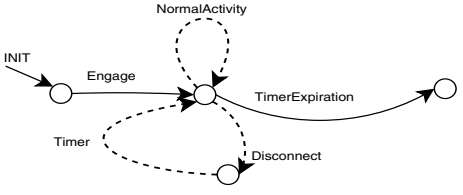
Fig. 10. Execution graph of machine *AgentSystem4*



(a) Event execution possibility 1



(b) Event execution possibility 2



(c) Event execution possibility 3

Fig. 11. All possible Event execution scenarios

Timer ref TempFailure = **ANY** *aa* **WHERE** (*aa* ∈ *agents*) ∧ (*aa* ∈ *timers*)
THEN *disconn_limit* := *disconn_limit* - 1 || *ex_agents* := *ex_agents* ∪ {*aa*} ||
timers := *timers* - {*aa*} **END**
END

The execution graph for *AgentSystem4* is shown in Fig.10. This graph shows all the possible events with their respective states but the order of execution is controlled by their guards. In addition, the Fig.11 shows all the possible scenarios based on the information derived from events' guards and bodies. The dashed arrows represent possible loops of the event(s) during the execution. In order to generate concrete test cases from such models, the number of executions of an event in the loop can be restricted to some finite bound. The value for this bound depends on user's coverage criteria.

When testing event-based systems, the system can have demonic choice and can execute the loop-event forever. One of the possible ways to restrict such an execution is to introduce variants both in the specification and the implementation. The variant will ensure finite number of executions of that event. In the case where implementation does not has such variant and the system may

continue execution forever then applying test case of finite length will detect a live-lock in the system. Here it is assumed that no valid implementation has such infinite execution.

The CSP representations of the *AgentSystem4* machine is given in the following.

```
AgentSystem4 = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = Disconnect -> Node2
Node1 = Failure -> SKIP
Node1 = NormalLeaving -> SKIP
Node1 = TimerExpiration -> SKIP
Node2 = TempFailure -> Node1
Node2 = Timer -> Node1
```

Since in Event-B, every event is guarded, here, it is assumed that each event is enabled only when its corresponding *guard* is enabled. The guard information can also be expressed within CSP expressions as

```
(BooleanGuard & EventName)
```

These CSP expressions are finally unfolded into test cases by the methodology described in Section 3.4. These test cases are applied on the implementation to test the fault-tolerance scenarios.

In this case study, we showed how our scenario-based testing approach can be used in developing a fault-tolerant software application. We described a step-wise development approach showing how testing scenarios are refined alongside the refinements in the corresponding models. In this case study example, we used a chain of Event-B machines where the test cases are finalized from the single sufficiently refined machine. However, in practice, it is possible to decompose functionalities of the system across multiple components (machines). Our scenario-based testing approach would also work in that case provided that these components are developed from an abstract component in a consistent fashion also obeying the basic refinement types described earlier in the section 3.2 of this paper.

5 Related Work and Conclusions

The existing tools or techniques for model-based testing using model-oriented languages (e.g., see [7,15]) are based on the notion of the coverage graph, which is obtained from symbolic execution of the model. In these approaches, as a first step, the input space of an operation is partitioned into equivalent classes to create the corresponding operation instances and then the coverage graph is constructed. This coverage graph contains the sequence of operations which are then tested in the implementation. However, in these approaches, the user scenarios are not represented and tested.

In our earlier work [16], we presented the scenario-based testing approach for B models where we designed an algorithm for constructing test sequences across different refinement [6] models. However, this algorithm is exponential in nature thus limiting its practical applicability.

In this paper, we presented a model-based testing approach based on automatic refinement of test scenarios. In this work, we also described basic refinement rules, allowing us to do the development in a controlled way, and transform our testing scenarios according to those rules. This approach does not involve any exponential algorithm thus making it more applicable in practice. This methodology is well suited for development of complex systems in which the fault-tolerance mechanisms are incorporated alongside the main system functionality. However, this automatic test case generation method can also be used in formal software development process in general. The presented methodology also allows us to have multiple instances of testing scenarios to test different functionalities or features of the system.

Currently, our approach supports several basic refinement types. However, in the future, we are going to extend our method by including more refinement types. We also plan to work on building an execution environment where the test cases can be executed on the system-under-test in a controlled manner. This would enable us to interpret the test results for each test case execution. In the cases, where an error is discovered, the environment should help to trace this error to a particular place in the corresponding software model.

Acknowledgments

This work is supported by IST FP6 RODIN Project.

References

1. Cambridge Dictionary for English, `http://dictionary.cambridge.org/`
2. Software Testing Online Blog. `http://testingsoftware.blogspot.com/2006/02/test-case.html`
3. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
4. Abrial, J.-R.: Event Driven Sequential Program Construction (2000), `http://www.matisse.qinetiq.com`
5. Abrial, J.-R., Mussat, L.: Introducing Dynamic Constraints in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, p. 83. Springer, Heidelberg (1998)
6. Back, R.-J., von Wright, J.: Refinement calculus, part i: Sequential nondeterministic programs. In: REX Workshop, pp. 42–66 (1989)
7. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: Gsm 11-11 standard case study. Softw. Pract. Exper. 34(10), 915–948 (2004)
8. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
9. Dalal, S.R., et al.: Model Based Testing in Practice. In: Proc. of the ICSE 1999, Los Angeles, pp. 285–294 (1999)

10. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Englewood Cliffs (1985)
11. Laibinis, L., Troubitsyna, E., Iliasov, A., Romanovsky, A.: Rigorous development of fault-tolerant agent systems. In: RODIN Book, pp. 241–260 (2006)
12. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
13. Naslavsky, L., Alspaugh, T.A., Richardson, D.J., Ziv, H.: Using Scenarios to support traceability. In: Proc. of 3rd int. workshop on Traceability in emerging forms of software engineering (2005)
14. Roscoe, A.W.: The theory and practice of concurrency. Prentice-Hall, Englewood Cliffs (1998)
15. Satpathy, M., Leuschel, M., Butler, M.J.: ProTest: An automatic test environment for B specifications. Electr. Notes Theor. Comput. Sci. 111, 113–136 (2005)
16. Satpathy, M., Malik, Q.A., Lilius, J.: Synthesis of scenario based test cases from *b* models. In: FATES/RV, pp. 133–147 (2006)
17. Utting, M., Legeard, B.: Practical Model-Based Testing. Morgan Kaufmann Publishers, San Francisco (2006)