Automated testing reactive systems from Event-B model

Dieu Huong Vu, Anh Hoang Truong
VNU University of Engineering and Technology, Vietnam

Yuki Chiba, Toshiaki Aoki
Japan Advanced Institute of Science and Technology, Japan

***Abstract*—We present a model-based testing approach for reactive systems where both test inputs and expected results are generated from 'restricted' Event-B specifications. We show that it is possible to automatically build the restricted Event-B specifications from the original ones base on a knowledge base of the system under tests. The restricted models are to reduce the state space of the original Event-B models while preserving the possible testing paths, so that our model-based generated test suite can archive equivalent path coverage as using the original models. We also present a tool and a testing skeleton that are easy to use so that system developers can effectively test an arbitrary number of scenarios with reactive systems.**

I. Introduction

Reactive systems, e.g., vending machines, elevators, continuously response to stimulus from their environments. For example, vending machines accept customer actions to give them items or elevators accept input from the passengers to bring them to their floors. Internally, the stimulus or events from the environment trigger calls or invocations to system's functions or services. Each event may have additional parameters and may trigger additional events. Each event usually changes the run-time states of the system. Therefore, to guarantee some level of correctness of a reactive system, we need to test it with all many as possible the number of sequences of events (test scenarios) together with the possible values of event parameters. However, the number of test cases grows exponentially by the length of the events and the size of the parameters' domains, so testing reactive systems is challenging [13]. We need an efficient and scalable approach to reduce the number of test cases.

For critical reactive systems, their requirements are usually specified using formal notation, e.g. Event-B specifications [1]. In this paper, we propose a model-based testing approach [16] where models are Event-B specifications. Our approach provides system developers a skeleton that can generate test scenarios which contain both input values and expected results. The skeleton acts as a test environment and a test driver that exercises the system under test (SUT) and reports bugs if the actual results are different from the expected ones. Of course, there are other techniques such as model checking or formal verification to guarantee some correctness properties of reactive systems [11]. However, testing is still a widely applied in practice, because the formal approaches are only applicable for certain classes of programs and correctness properties.

In particular, our problem statements and approach are summarized as follows. Given an Event-B specification and an implementation of a reactive system, also called system under test (SUT), check if all *representative behaviors* of the specification are correctly implemented in the SUT. Here the representative behaviors are the set of different sequences of events with their representative input values. To generate these sequences, we first restrict the original Event-B specification to make a so called *specification for test* (SfT). The technique that we use is based on representative values from equivalence class partitioning, a popular black-boxed testing technique [3], which is effective and suitable in our context as we will explain in more details in the next section. Then we build a labeled transition system (LTS) of the state spaces from the restricted specification. The paths of the LTS are used to build test cases, which contain both test inputs and expected results for assertions.

The main contributions of the paper are: (i) a technique to combine model-based testing with equivalence classes partitioning to reduce redundant test cases; and (ii) a supporting tool and a test driver skeleton for system developers to easily build test driver that automatically tests the reactive systems for correctness with respect to its Event-B specification.

In the followings, we will present a motivating example in Section 2. In section 3, we will formalize our approach. Section 4 introduces our tool and experimental results. Section 5 discusses related works and Section 6 concludes.

II. Motivating example

We will explain our approach via a simplified example. Suppose we have an Event-B specification S for a bank ATM system as shown in Figure 1. The bank ATM should provide services: deposit an amount, withdraw an amount and transfer an amount from an account to another. Figure 2 is a simplified implementation of the system. We need to check that all the behaviors in the specification in Event-B semantics [1] are correctly respected by the implementation.

A. System specification

The behaviors of the specification in Figure 1 are valid sequences of the three events with their corresponding parameters: `Deposit x n`, `Withdraw x n` and `Transfer x, y, n`, where the parameters `x`, `y` and `n` are implicit in the specification. In addition, these sequences should not produce states (the values of variables `Accounts` and `Balance`) that violate the invariants in the `INVARIANT` section. The states here are the values of two variables `Accounts` and `Balance`. The variable `Accounts` holds the set of all existing bank accounts. Its type is abstracted in `ACCOUNTS`, which, for example, be all sequences of 10 digits. The variable `Balance` is a map from `ACCOUNTS` to

the set N of non-negative integer representing the balances of the accounts.

When the environment sends an event to the system under test, it triggers the actions in the THEN parts if the corresponding guard conditions in the WHEN parts hold. The benefit of using Event-B as the models for testing is that the correctness of the models specified by the invariants is proved by Event-B tools [12]. In addition, from the formal model and the clear semantics of Event-B, we will explore its behaviors using a labeled transition system whose paths will be test inputs and assertions to be check with the implementation.

However, the state space of the Event-B specification, i.e. the set of all possible valid values of Balance and Accounts, is very large. It is inconvenient to explore the whole state space for test cases generation, as in testing we want to validate the system under test with typical inputs. Hence, we need to build a restricted version of the specification such that it has as small as possible the state space but, for test effectiveness, it should preserve as much as possible the behaviors of the original specification.

```
CONSTANTS ACCOUNTS, AMOUNTS
VARIABLES Accounts, Balance
INVARIANT
    Accounts ⊆ ACCOUNTS,
    Balance ∈ Accounts -> AMOUNTS,
    ∀ x ∈ Accounts: Balance(x) >= 0
INITIALIZATION
    Balance := 0
    Accounts := ACCOUNTS
EVENT
  Deposit = WHEN x ∈ Accounts and n ∈ AMOUNTS
    THEN Balance(x) := Balance(x) + n.

  Withdraw = WHEN x ∈ Accounts AND n ∈ AMOUNTS AND
      n <= Balance(x)
    THEN Balance(x) := Balance(x) - n.

  Transfer = WHEN x, y ∈ Accounts AND n ∈ AMOUNTS
      AND n <= Balance(x)
    THEN Balance(x) := Balance(x) - n
    AND  Balance(y) := Balance(y) + n
```

Fig. 1. Event-B model of a simplified ATM

B. Restricted Specification and Labeled Transition System

For the specification in Figure 1, if we restrict ACCOUNTS to {a, b} and AMOUNTS to {0, 50, 100}, the state space is much smaller and it is feasible to explore all possible behaviors of the restricted specification to check with the system under test. For the example specification in Figure 1, we only need to change the first line to CONSTANTS ACCOUNTS = {'a', 'b'}, AMOUNTS = {0, 50, 100}. After restricting the domains of variables in the specification, we use the restricted specification to generate a labeled transition system (LTS) as in Figure 3 by the following procedure. First, we create the node a0,b0, which is the initial state of the test specification. We mark this node as the entry node. Here we denote xN the account x with balance N. Then, from this initial state, we scan all events that can be triggered by checking their guard conditions. Four more valid states are (a50,b0), (a0,b50), (a100,b0) and (a0, b100) by Deposit event. So we create the nodes with edge labeled by the event names. To simplify the presentation, we use Da to

```
public class Account {
  public int balance = 0;
  public String name;
  public Account(String name){
    this.name = name;
  }
  public void withdraw(int amount) {
    if (amount <= balance && amount > 0) {
      balance = balance - amount;
    } else { throw Ex; }
  }
  public void deposit(int amount) {
    balance = balance + amount;
  }
  public void transfer(Account to, int amount) {
    if (this.balance >= amount && amount > 0) {
      this.balance -= amount;
      to.balance += amount;
    } else { throw Ex; }
} }

public class ATM {
  public List<Account> accounts = new
      List<Account>();
  private Account currentAccount;
  public void add(Account a) {
    accounts.add(a);
  }
  public void authorize(String name) {
    currentAccount = accounts.find(name);
  }
  public void withdraw(int amount) {
    currentAccount.withdraw(amount);
  }
  public void deposit(int amount) {
    currentAccount.deposit(amount);
  }
  public void transfer(Account to, int amount) {
    currentAccount.transfer(to, amount);
} }
```

Fig. 2. A simplified implementation of a bank ATM in Java

denote the event Deposit $50 to account a and similarly for other events and we omit edges with value 100 from a0,b0 to a100,b0 and a0,b100. For each new node, we repeat the above steps to expand the LTS and we can build a LTS which is partly shown in Figure 3. In this figure, we only show edges for the events with 50 in amount.

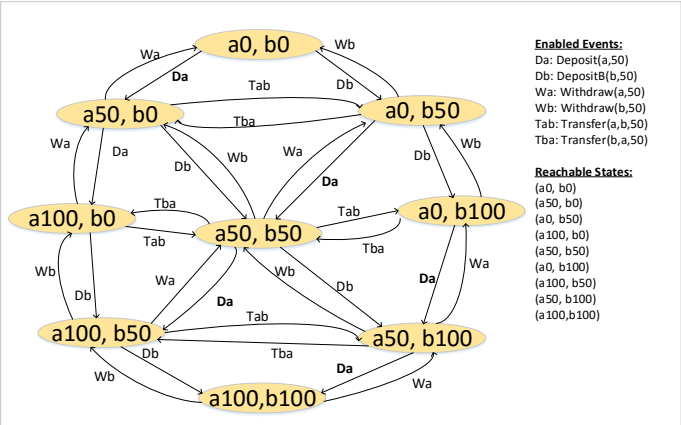

Fig. 3. LTS built from Figure 1

Listing 1. LTS data structure

```
1 //LTS structures
2 public class Node {
3     public Account account1, account2;
4     public List<Edge> edges;
5 }
6 public class Edge {
7     public Node start, end;
8     public String action; //'D', 'W', 'T'
9     public Account param1, param2;
10
11    public Account getAccount() { return param1; }
12    public Account getToAccount() { return param2; }
13 }
14 public class LTS {
15   public Node entry;
16    List<Node> nodes;
17 }
```

From the LTS, now we can generate test cases by searching all possible paths from the entry node with unlimited length. Figure 4 shows paths, which are sequences of events, to be tested with the implementation of the reactive system under test. In these sequences, `Da` stands for depositing $50 to account `a`. Similarly, `Db`, `Wa`, `Wb`, `Tab` and `Db` denote depositing to account `b`, withdrawing from `a`, withdrawing from `b`, transferring from `a` to `b` and transferring from `b` to `a`, respectively, and all with amount of `50`.

```
Da-Da-Db-Db-Wa-Wb-Tab-Wb-Da-Db-Da-Wb-Wa-Tba-...
Db-Da-Tba-Db-Wb-Wa-Wb-Tab-Wb-Tba-Wa-...
Da-Da-Tab-Db-Da-Wb-Tab-Tba-Da-Wb-Tab-Db-Da-...
```

Fig. 4. Sample test sequences generated from the LTS in Figure 3

In addition, we can also generate assertions or test oracles after each event as the node `xn` in a node says that account `a` has balance `n` so we can use this to generate assertions. For example, for node `(a50, b50)` we can verify with `assert(Balance(a)==50 && Balance(b)== 50).`

C. Test Execution

So now we have test sequences with inputs and assertions after each event, we still need to execute the tests to find if the system under test conforms to the provided specification. This module is often called test driver [2]. Its main functions are: read test inputs, execute the SUT, and compare the actual results with the expected ones.

We set the initial state of the SUT and send a series of events to the SUT and observe the new state of the SUT after each event. In our example, as in Listing 2 we will start with the entry node of the generated LTS (line 8), set the SUT to the correct initial state (lines 10-11). Then we randomly select one of its edges (lines 14-15) that can go from the current node and send the corresponding event to the SUT (lines 17-21). After that we check the new state of SUT if it conforms to the expected state of the model represented by the LTS (lines 24-27).

Listing 2. Test driver for the ATM example

```
1 //random test code
2 class TestDriver {
3   void runRandomTest (LTS lts) { //main
4     atm = new ATM();
5     atm.add(new Account("a"));
6     atm.add(new Account("b"));
7
8     n = lts.entry;
9     //checking the state with the model
10    assert (atm.accounts["a"].balance ==
          n.account1.balance);
11    assert (atm.accounts["b"].balance ==
          n.account2.balance);
12    while (true) {
13      //random testing strategy
14      r = random(0, n.edges.length()-1);
15      e = n.edges[r];
16      // send event to SUT
17      atm.authorize(e.getAccount().name);
18      switch (e.action){
19        case 'D': atm.deposit(50); break;
20        case 'W': atm.withdraw(50); break;
21        case 'T': atm.transfer(e.getToAccount(),
              50);
22      }
23      //new state
24      n = e.end;
25      //checking the state with the model
26      assert (atm.accounts["a"].balance ==
            n.account1.balance);
27      assert (atm.accounts["b"].balance ==
            n.account2.balance);
28 } } }
```

III. Formalization

A. Event-B models

Event-B models have variables, their domains containing values for variables, invariants containing expressions and boolean expressions, and events containing guards and actions. A guard is a boolean expression. An action is a value assignment, so called substitution.

$\mathcal{V}$ is the set of *variables*. $\mathcal{D}$ is the *domain*, which is the set of values. Exp is the set of expressions in the specifications. An *expression* may contain variables in $\mathcal{V}$, values in $\mathcal{D}$, arithmetic operators, logical operators, and set operators. BExp is the set of boolean expressions (BExp $\subset$ Exp). A *substitution* $a : \mathcal{V} \rightarrow$ Exp is a mapping from $\mathcal{V}$ to Exp. ACT is the set of substitutions. GRD is the set of guards. An *event* is a pair $\langle g, a\rangle$ of a guard g and a substitution a. $\mathcal{E}$ is the set of events. If $e = \langle g, a\rangle$ then we write $grd(e) = g$ and $act(e) = a$. A *state* is a value assignment. $[exp]_\sigma$ denotes the interpretation of the value of an expression exp in a state σ. We say a guard g holds in a state σ iff $[g]_\sigma = tt$ (truth value). Init is the set of special initialization events that have no guard. We denote $\sigma \xrightarrow{e} \sigma'$ for an event $e = \langle g, a\rangle$ and states σ and σ' if $\sigma(g)$ holds and $\sigma' = \{v \mapsto [a(v)]_\sigma \mid v \in V\}$.

We define a specification in Event-B as follows.

Definition 1 (Specification): A *specification in Event-B* is a tuple $S = \langle \mathcal{V}_S, \mathcal{D}_S, \Sigma_S, \text{Init}_S, Inv\rangle$ where $\mathcal{V}_S \subseteq \mathcal{V}$ is the set of variables used in S, $\mathcal{D}_S \subseteq \mathcal{D}$ is the domain of variables in $\mathcal{V}_S$, $\Sigma_S \subseteq \mathcal{E}$ is the set of events, $\text{Init}_S \in$ Init is the initialization of S, and $Inv \in$ BExp is the invariant of S.

B. Restricted models

As mentioned earlier, the state space of the Event-B specification, i.e. the set of all possible valid values of `balance` and `accounts` in the example, is very large. For test effectiveness, we need to build a restricted version of the specification such that it has as small as possible the state space but it should preserve as much as possible the behaviors of the original specification. A simple restriction can be made by changing the domains of the original specification to contain only few representative values of the domains. This task requires expert knowledge about the application domain of the system under test, but it can be done in several steps so that the restricted domains are still large enough for the intermediate computed values to be contained in the domains.

Given a specification S, we restrict the specification by using a knowledge base of the system under test. The knowledge base contains a set of representative values for each variable in $\mathcal{V}_S$. In particular, the representative values can be selected using equivalence class partitioning, a black-boxed testing techniques. From these representative values, e.g. 0 and 50 in the motivating example, we may need to extend the restricted domain with additional values so that the system under test and the model can be executed other computation operations, e.g. 100 in the motivating example. Formally, the knowledge base is a mapping $G : 2^{D_S} \rightarrow 2^{D_S}$. For the model in Figure 1, $G(ACCOUNTS)$ can be $\{a, b\}$ where a, b are two distinguish sequences of ten digits and $G(BALANCES)$ can be $\{0, 50\}$. We build a restricted version of the specification, called specification for test (SfT), by replacing range of values for each variable in V_S, D_S with the smaller range, 2^{D_S}. For the model in Figure 1, we replace ACCOUNTS with `{a, b}` and replace BALANCES with `{0,50}`. SfT is obtained from S by applying G is a a tuple $SfT = \langle \mathcal{V}_S, \mathcal{D}_{SfT}, \Sigma_{SfT}, \text{Init}_S, Inv \rangle$ where $\mathcal{D}_{SFT} = G(\mathcal{D}_S)$ and Σ_{SFT} is a set of events e in Σ_S that satisfy the restriction defined by G.

C. Building LTS from restricted specification

A labeled transition system (LTS) explored from the restricted specification is a finite LTS. It contains a set of states Q. States of the LTS are computed by assigning all possible values for variables within range of values in the SfT. The set of reachable states is a subset of Q.

Definition 2: (LTS). An LTS M is a quadruple $\langle Q, \Sigma, \delta, I \rangle$ where $Q = \{\sigma \mid \sigma : \mathcal{V}_S \rightarrow \mathcal{D}_{SfT}\}$ is a non-empty set of states, $\Sigma = \Sigma_{SfT}$ is a set of actions, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, we write $(p, a, p') \in \delta$ as $p \xrightarrow{a} p' \in \delta$, and $I \subseteq Q$ is a set of initial states.

We present an algorithm to generate an LTS from SfT in Algorithm 1. This is an extension of an algorithm to generate an LTS from an Event-B model in which every ranges of values for variables is finite presented in [17].

Algorithm 1 Generating LTS $M = \langle Q, \Sigma, \delta, I \rangle$ from specification for test in Event-B $SfT = \langle \mathcal{V}_S, \mathcal{D}_{SfT}, \Sigma_{SfT}, \text{Init}_S, Inv \rangle$. $E(\sigma)$ is used to denote the set of events which are applicable to state σ and in Σ_{SfT}.

1: $QUEUE = empty$
2: $VISITED = empty$
3: $Q = \Sigma = \delta = I = empty$
4: **for** each $\sigma_0 \in \{act(e) \mid e \in \text{Init}_S\}$ **do**
5: **if** $\forall v \in V_S, \sigma_0(v) \in \mathcal{D}_{SfT}$ **then**
6: Push($QUEUE, \langle \sigma_0 \rangle$)
7: $Q = Q \cup \{\sigma_0\}$
8: $I = I \cup \{\sigma_0\}$
9: **end if**
10: **end for**
11: i=0
12: **while** $QUEUE \neq empty$ **do**
13: $\langle \sigma \rangle$ = Pop($QUEUE$)
14: $VISITED = VISITED \cup \{\sigma\}$
15: $\widehat{E} = \{e \mid e \in E(\sigma)\}$
16: **if** $\widehat{E} \neq empty$ **then**
17: WRITEFILE state label(σ)
18: **for** each $v \in V_S$ **do**
19: WRITEFILE $\sigma(v)$
20: **end for**
21: **for** each event $e = (g, a) \in \widehat{E}$ **do**
22: $\sigma' = \{v \mapsto [(act(e))(v)]_\sigma | v \in \mathcal{V}_S\}$
23: WRITEFILE transition label $\{\sigma \xrightarrow{e} \sigma'\}$
24: WRITEFILE from state σ
25: WRITEFILE to state σ'
26: **if** $\sigma' \notin VISITED$ **then**
27: Push($QUEUE$,$\langle \sigma' \rangle$)
28: $Q = Q \cup \{\sigma'\}$
29: **end if**
30: $\Sigma = \Sigma \cup \{e\}$
31: $\delta = \delta \cup \{\sigma \xrightarrow{e} \sigma'\}$
32: **end for**
33: **end if**
34: **end while**
35: return M

D. Test sequences and test oracle explored from LTS

Test sequences are unlimited chains of invocations which call services of SUT. Test sequences are explored from the LTS by traversing edges from the initial states. Supposing LTS $M = \langle Q, \Sigma, \delta, I \rangle$ be generated from SfT and $E(\sigma)$ be used to denote a set of applicable invocations in state σ, we define test sequences as follows: $TS = [\sigma_0]e_1; [\sigma_1]e_2; ...; [\sigma_{k-1}]e_k; ...$, where $\sigma_0 \in I$, and $e_k \in E(\sigma_{k-1})$.

Test oracles are assertions used to check whether actual results of methods in SUT are the same as results of events in SfT. The results of events in SfT are represented in reachable states following edges in the LTS. As we mention earlier, states are value assignments. For each variable $v \in V_S$ and for each corresponding variable u in SUT, we expect that value of u is the same as value of v in states before and after each method of SUT is invoked. We use $AcV(u)$ to denote the actual value for variable u observed in a state of SUT. Supposing LTS $M = \langle Q, \Sigma, \delta, I \rangle$ be generated from SfT, assertions are defined as follows:

- In the initial states: For each variable $v \in V_S$ and for each corresponding variable u in SUT, $AcV(u) = \sigma_0(v)$, where $\sigma_0 \in I$ (see line 10-11 of Listing 2 for an example of these assertions)
- After each invocation e in TS is applied where $\sigma \xrightarrow{e}$

Listing 3. LTS of the ATM example in JSON format

```
1  "states": [ {
2              "id": "a0b0",
3            "a": "0",
4            "b": "0"
5          },...]
6  "transitions": [ {
7            "name": "Da",
8              "from": "a0b0",
9              "to": "a50b0",
10             "params": "50"
11         },...]
```

$\sigma' \in \delta$, we check the value of variables in reachable states of SUT: For each variable $v \in V_S$, and for each corresponding variable u in SUT, $AcV(u) = \sigma'(v)$ (see line 26-27 of Listing 2 for an example of these assertions)

Actual values of variables in SUT are tested in states following chains of invocations by assertions. A test cases fails when an assertion is violated.

IV. Tool and Experiment

A. Test Driver Skeleton

We developed a tool to generate LTS from the SfT described in the previous section. The result LTS generated by the module in Algorithm 1 is stored in JSON format. A simplified JSON of the ATM example is in Listing 3. To test the implementation, we provide a test driver skeleton that can be easily used by system developers to build a test driver to check their implementation with the LTS as one of the input (see Listing 2 for an example). The skeleton is mainly the lines 8-15 and 23-28 in Listing 2. It travels the LTS randomly and checks the system under test before and after each event. The events are generated randomly in lines 14. With this skeleton, system developers can easily modify it for other models and systems. We used this tool and skeleton in our case studies. Target systems used in our case studies are vending machines and ATMs. All of these systems are implemented in Java.

B. Case studies

We build two models, for the ATM example and for a vending machine. A vending machine is a machine which dispenses items such as snacks, beverages, cigarettes, lottery tickets, etc. to customers automatically, after the customer inserts currency into the machine. The specification of vending machines describes their external behaviors including inserting credit into the machine, returning credit, restocking an item, and dispensing an item. Each of them is a so-called service. Figure 5 demonstrates a specification of the vending machine in Event-B. Variable `stock` defines a set of items that are currently available to be dispensed. It has an abstract data type namely `PRODUCT`. Variable `cred` defines the total of money deposited so far and available to make a purchase. Variable `state` defines the state of the vending machine. Variable `card` defines the size of `stock`. The knowledge base for the vending machine is also straightforward.

Supposing that a domain knowledge database of vending machines shows 3 kinds of items available for customers to

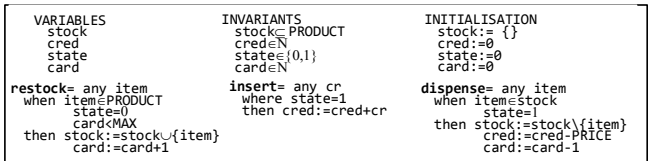

Fig. 5. Specification of Vending Machines

be bought such as water, tea, coffee, and 20 for each. Such kind of vending machines should accept quarter, dollar as input before selecting any item. Based on this domain knowledge database, we restrict `PRODUCT` to `{w(water), t(tea), c(coffee)}`, `PRICE` in `{$1}`, `COIN` could be `Q(QUARTER)` and `D(DOLLAR)`, `cred` in `{$0.25, $0.5, $1}`, and `card` in `[0..60]`. These ranges of values are appropriate in practical applications of vending machines.

All experiments are conducted on Intel (R) Core (TM) i5 Processor at 2.67GHz running Windows 10. Experiment results are shown in Table I. In all cases of our experiment, memory usage is minimal (M). Traces of the test show that about 800 invocations are tested per second and every state and invocation are covered in the test. The test is executed infinitely if no error is detected. This is due to unlimited nature of reactive system, which is caused by unbounded length of the test sequences. The test is interrupted immediately after the first error is detected and the test returns a violation assertion.

TABLE I. Experiment Results

#	Systems	Mem (Mb)	State Cover.	Arc Cover.	Result
1	ATM-Ver1	M	100%	100 %	no error
2	ATM-Ver2	M	100%	100%	no error
3	VM-Ver1	M	N/A	N/A	1 error
4	VM-Ver2	M	100%	100%	no error

The experiment #1, #2, and #4 are executed infinitely. The experiment #3 returns a violation assertion. Following the trace shown with the error, we could easily detect bugs in implementation of service `select item` for Vending Machines. Memory usage and performance are very good to decrease cost of the test.

We asked the engineers to intentionally add some bugs into versions of ATM and Vending Machine. Then, we test these versions using the same test code. All bugs are detected in a short time. Types of bugs could be detected by our test code as below:

- Conditions enabling services and result of individual services of SUT is not corrected with respect to its specification. For example, one version of ATM allows the customers to withdraw money when the balance of their accounts is smaller than the amount.
- Combination of services does not operate correctly with respect to its specification. For example, the results of `insert money` and `select item to be bought` are not appropriate inputs for `dispense item`.

V. Discussion

In approach, we can test not only individual services of SUT but also a series of service calls. This is a strong point

because in order to check essential properties of SUT, we should exercise sequences of different services of SUT. Our strategy to test the SUT with random sequence of events is natural in practice, but we can replace random strategy by more systematic approaches, using different path exploration for the LTS. We can also easily track the coverage of the test with respect to the model by marking edges and nodes that test sequences have traversed. So our approach is extensible for more advanced test strategies that developers may need. With our test driver skeleton, memory and performance are not a problem as the traversal of the LTS does not incur additional cost for memory after each loop.

VI. Related Works

There exists many approaches to model -based testing. In [5] and [8], the authors focus on negative test cases rather than positive ones. Full combination of services is not considered. In our approach, we focus on positive test cases, test oracles and sequences of them. Approaches presented in [15], [9], [14] aim at generating test cases from UML diagrams. UML is semi-formal language, so test oracles are not easy to make them right. In [4], the authors present how test sequences and test oracles are generated from a design in Promela which describes internal behaviors of the system. In contrast, we focus on external behavior of reactive systems. In [13], an approach to generate *test case chains* is focused. In this context, test case covers a property if it triggers the transition the property relates to. A test case that covers a sequence of properties is regarded as a test case chain. This work focuses on safety properties. Our work present an approach to generate test cases and test oracle that cover all possible behavior with unlimited length of invocation sequences coming from reactive system's environment. An MBT plug-in for Rodin which generates test suites from Event-B models is presented in [7]. A test case is defined as a sequence of actions (or events, or triggers) together with corresponding test data that can be executed against a SUT. In order to generate test cases, the users of this plug-in need to set a constant value for the maximum sequence length of test cases. In our work, we do not limit the length of test cases. This is appropriate for infinite behavior of reactive systems. Event-B is adopted in [10] to construct a model of SUT from abstract level to implementation level and uses CSP to formalize test scenarios. These test scenarios are refined step by step and conform to the model of SUT in Event-B. Test cases are generated from the refined scenarios. In our work, we generate test cases from requirement specification of SUT described in Event-B. Our test cases could exercise all possible behaviors of SUT. A test generation approach from user-defined scenarios, for behavioral models expressed as B machines, is presented in [6]. In our work, we focus on reactive systems; therefore, we consider the infinite behavior of reactive systems and the nondeterminism to select an invocation applicable in each state of SUT. In our approach, the length of test cases is infinite and test cases cover full combination of services.

VII. Conclusion

We have presented a model-based approach for testing reactive systems, in which the key challenge is the state space explosion problem of the model. We proposed an approach to modify the original models by restricting the domain of variables in the model to only some representative values base on the domain knowledge of the application. The domain knowledge should be easily built by experts or the customers of the system under test and they can be reused, enriched if needed to make the automated testing suitable to the test goals. From the restricted model, it is feasible to explore its state space to test with the implementation for conformance between the model and the implementation. We have built a test framework that takes a model, a knowledge base and a system under test as input and system developers can easily build a test driver to check the conformance between the model and the implementation. For future work, we plan to build a plug-in for Rodin platform so that our approach can work with more generic Event-B models.

Acknowledgments

This work is supported by the project No. 102.03–2015.25 granted by Vietnam National Foundation for Science and Technology Development (Nafosted).

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[2] Fairouz Tchier Ali Mili. *Software Testing: Concepts and Operations*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2015.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[4] Jiang Chen and Toshiaki Aoki. Conformance testing for osek/vdx operating system using model checking. APSEC '11, pages 274–281.

[5] Yunja Choi. *Constraint Specification and Test Generation for OSEK/VDX-Based Operating Systems*, pages 305–319. Springer, 2013.

[6] Frederic Dadeau, Kalou Cabrera Castillos, and Regis Tissot. Scenario based testing using symbolic animation of b models. *Software Testing, Verification and Reliability*, 22(6):407–434, 2012.

[7] Ionut Dinca, Florentin Ipate, Laurentiu Mierla, and Alin Stefanescu. *Learn and Test for Event-B – A Rodin Plugin*, pages 361–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[8] L. Fang, T. Kitamura, T. B. N. Do, and H. Ohsaki. Formal model-based test for autosar multicore rtos. In *2012 IEEE 15th International Conf. on Software Testing, Verification and Validation*, pages 251–259.

[9] Peter Fröhlich and Johannes Link. Automated test case generation from dynamic models. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 472–492, 2000.

[10] Qaisar A. Malik, Johan Lilius, and Linas Laibinis. Fast abstract: Generating test cases from scenario-based formal development.

[11] Martin Ouimet and Kristina Lundqvist. Formal software verification: Model checking and theorem proving. Technical report, March 2007.

[12] RODIN and DEPLOY group. Event-B and the RODIN platform, http://www.event-b.org/.

[13] Peter Schrammel, Tom Melham, and Daniel Kroening. Generating test case chains for reactive systems. *International Journal on Software Tools for Technology Transfer*, 18(3):319–334, 2016.

[14] Dirk Seifert. Conformance testing based on uml state machines. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, pages 45–65. Springer-Verlag, 2008.

[15] Dirk Seifert, Steffen Helke, and Thomas Santen. Test case generation for UML statecharts. In *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conf.*, pages 462–468, 2003.

[16] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[17] Dieu-Huong Vu, Yuki Chiba, Kenro Yatake, and Toshiaki Aoki. A framework for verifying the conformance of design to its formal specifications. *IEICE Transactions*, 98-D(6):1137–1149, 2015.