Automatic Testing from Formal Specifications[⋆]

Manoranjan Satpathy[1⋆⋆], Michael Butler[2], Michael Leuschel[3], and S. Ramesh[4]

[1]Department of Information Technologies, Abo Akademi University
Joukahaisenkatu 3-5, FIN-20520 Turku, Finland
[2]School of Electronic and Computer Science, University of Southampton
Highfield, Southampton, SO17 1BJ, UK
[3]Institute of Informatik, Heinrich-Heine Universitat Duesseldorf
Universitatsstr. 1, D-40225 Duesseldorf
[4]General Motors India Science Lab
International Tech Park, Whitefield Road, Bangalore – 560066
mannu.satpathy@abo.fi; mjb@ecs.soton.ac.uk
leuschel@cs.uni-duesseldorf.de; s.ramesh@gm.com

Abstract. In this article, we consider model oriented formal specification languages. We generate test cases by performing symbolic execution over a model, and from the test cases obtain a Java program. This Java program acts as a test driver and when it is run in conjunction with the implementation then testing is performed in an automatic manner. Our approach makes the testing cycle fully automatic. The main contribution of our work is that we perform automatic testing even when the models are non-deterministic.

Key Words: *Model Based Testing; B-Method; Non-determinism*

1 Introduction

Software models are usually built to reduce the complexity of the development process and to ensure software quality. A model is usually a specification of the system which is developed from the requirements early in the development cycle [5]. In this paper, we consider model oriented formal specification languages like Z [22], VDM [12] , B [1] and ASM [9]. By *model oriented* we mean that system behaviour is described using an explicit model of the system state along with operations on the state.

A formal model can be subjected to symbolic execution to obtain a coverage graph in which nodes represent instantiated states and edges are labeled with operation applications. One can then select a finite set of finite behaviours from the coverage graph and test if the implementation is consistent with these behaviours. This approach is often termed as model based testing [6]. Model

[⋆] Work done within the EU research project Rodin, IST 511599.

[⋆⋆] Currently at the General Motors India Science Lab (ISL), Bangalore; part of this work was done when the author was visiting GM ISL during Summar'05.

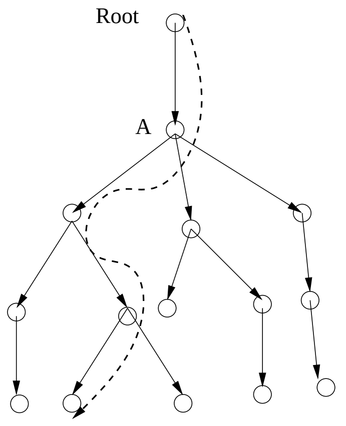

Fig. 1. Non-determinism Scenario

based testing though is an incomplete activity; the selected behaviours could be enriched to capture interesting aspects of the system and hence the success of their testing would give us confidence about the correctness of the system.

In this paper, we discuss automatic testing of an implementation or the system under test (SUT) written in accordance with a formal model in B [1]. Our method first generates a set of test cases from the model, and then from the test cases a test driver. The test driver is essentially a program in the target language. If this driver is plugged into the implementation, then testing can be performed automatically. It invokes all of the test cases generated from the model and reports about their success or failure. Our method does not require the implementation source to be available, and the entire testing cycle is automatic. The main contribution of our work is that our approach can perform automatic testing even if there is non-determinism in the model or in the implementation.

The basic idea behind the handling of non-determinism can be seen from Figure 1. Assume the solid lines in the figure show the full state space of a model in which branchings may be due to non-determinism. A correct implementation of this model must follow one of the paths in the figure, and for testing, we must know which path the SUT has taken. In our strategy, we maintain a generic representation of the possible paths that a model can take. Whenever the SUT makes a choice corresponding to a non-determinism in the model, we require that it makes this choice visible, and then this choice must satisfy a set of constraints which means that the implementation is not deviating from the model behaviour. Furthermore, the test driver uses the implementation choices to align the implementation trace with the appropriate trace of the model. Once this correspondence is established, additional properties could be checked with ease.

The organization of the paper is as follows. In Section 2, we discuss related work. Section 3 introduces the B notation with examples. In Section 4, we consider our approach to handle deterministic models, and in Section 5, we consider non-deterministic models. Section 6 discusses the implementation issues along with the current status of our implementation. In Section 7, we make an analysis of our method, and finally, Section 8 concludes the paper.

2 Related Work

A *testing criterion* is a set of requirements on test data which reflects a notion of adequacy on the testing of a system [19]. A test adequacy criterion determines whether sufficient testing has already been done, and in addition, it provides measurements to obtain the degree of adequacy obtained after testing stops [25]. A *test oracle* is a mechanism to determine correctness of test executions. A *test driver* is a tool which activates a system, provides test inputs and report test results [18]. *Representation mapping* is a mapping which maps the abstract name space of the model with the concrete name space of the SUT [7]. In this context, there are two kinds of mappings: control and data [19]. Control mappings are between control points in the implementation and locations in the specification; these are the points where the specification and the implementation states are to be matched. Data mappings are transformations between data structures in the implementation and those in the specification. A test sequence is called *preset* if the input sequence is fixed prior to the start of testing; it is called *adaptive* if at each step the choice of the next input symbol depends on the previous outputs [23]. The adaptive test cases are in the form of decision trees; the tester supplies an input and depending on the output, a branch is selected.

The work by Dick and Faivre [4] is a major contribution to the use of formal methods in software testing. A VDM [12] specification has state variables and an invariant (Inv) to restrict the variables. An operation, say OP, is specified by a pre-condition (OP_{pre}) and a post-condition (OP_{post}). The expression $\mathrm{OP}_{pre} \wedge \mathrm{OP}_{post} \wedge$ Inv is converted into its Disjunctive Normal Form (DNF); each disjunct, unless a contradiction itself, represents an input sub-domain of OP. Next, as many operation instances are created as the number of valid disjuncts in the DNF. An attempt is then made to create a FSA (Finite State Automaton) in which each node represents a possible machine state and an edge represents an application of an operation instance. A set of test cases is then generated by traversing the FSA, each test case being a sequence of operation instances. The authors discuss only the mechanism of the partitioning algorithm.

BZ-Testing Tool (BZ-TT) [2, 3, 14] generates functional test cases from B as well as Z specifications. BZ-TT assumes all sets in the B machine are finite enumerated sets. Each B operation is transformed to its *normal form* [1]. An operation is then partitioned into a set of operation instances; each partition usually corresponds to exactly one control path within the operation. The conjunction of all predicates in a control flow path and the postcondition is termed an *effect predicate* (EP). The free state variables in each EP are assigned to their maximum and minimum values – say, in terms of size – to obtain a set of *boundary goals*. *Boundary input values* are obtained by giving maximum and minimum values to the input variables in the EP. A Constraint Logic Programming (CLP) Solver tries to find a path through symbolic execution from the initial state to a boundary state, a state satisfying a given boundary goal. And then relevant operation instances are applied at the boundary state by giving them boundary input values. The results of the query operations become the oracle information. BZ-TT assumes that the B operations are deterministic. The authors point out that automatic verdict assignment is difficult because of non-determinism, and representation mappings [14, 15].

Satpathy et al. [21] discuss the prototype of a tool called ProTest which performs testing of an implementation in relation to its B model. The tool performs partition analysis using a technique similar to that of Dick and Faivre. A finite coverage graph is created from a symbolic execution of the B model by a model checking tool called ProB [16]. Some paths starting from the initial state are taken as test cases. The ProTest tool can run Java programs. So the B model and its Java implementation are run simultaneously by the tool, and in relation to a test case, similar states are matched to assign a verdict.

Finite state machines have been used to model systems like sequential circuits, communication protocols and some types of programs such as lexical analysis and pattern matching [13]. Though the implementation of such systems is usually deterministic, some of the state parameters may be unspecified during the specification stage. In such cases, non-deterministic finite state machines (NDFSMs) are used for modeling. Sometimes the code for the implementation (SUT) is not available and the problem then is to find if the SUT conforms to its finite state model; i.e. we need to show whether every i/o sequence that is possible in the SUT is also present in the specification. Solutions to this conformance testing problem when the specification is a NDFSM have been addressed by many authors including Hierons [10, 11], Zhang and Cheung [24] and Nachmanson et al [17]. However, the models which we discuss in this paper are in general infinite state machines.

3 The B-Method and Examples

The B-method is a theory and methodology for formal development of computer systems [1]. The basic unit of specification in B is called a *B machine*. Larger specifications can be obtained by composing B machines in a hierarchical manner. An individual B machine consists of a set of variables, an invariant to restrict the variables, and a set of operations to modify the state. An operation has a precondition, and an operation invocation is defined only if the precondition holds. The initialization action and an operation body are written as atomic actions coded in a language called the *generalized substitution language* [1]. The language allows specification of deterministic and non-deterministic assignments and operations. An operation invocation transforms a machine state to a new state. The behaviour of a B machine can be described in terms of a sequence of operation invocations in which the first operation call originates from the initial state of the machine.

We consider two B machines. The B machine TAgency1.mch is deterministic (Table 1). It has two users ($u1$ and $u2$) and two rooms ($r1$ and $r2$). The model

```
Machine TAgency1
SETS USER = $\{u1, u2\}$; SESSION= $\{s1\}$; ROOM= $\{r1, r2\}$
VARIABLES
        sess, booking
INVARIANT
        sess ∈ SESSION +→USER /* +→ means partial function */
        ∧ $booking$ ∈ROOM +→USER
INITIALISATION
        sess := ∅ || booking := ∅
OPERATIONS
login(u) = PRE $u$ ∈ USER ∧ $sess = \emptyset$ THEN sess(s1) := u
        END;
alloc(s)= PRE $s$ ∈SESSION ∧$sess \neq \emptyset \wedge dom(booking) \neq \{r1, r2\}$ THEN
        IF $r_1 \in dom(booking)$ THEN
            booking($r_2$) := sess(s)
        ELSE booking($r1$) := sess(s) END
        END;
logout(s) = PRE $s$ ∈SESSION∧$sess \neq \emptyset$ THEN sess := ∅ END
END
```

Table 1. A Deterministic B machine

can only handle a single session `s1`. `sess` and `booking` are the two variables, and the INVARIANT tells that both are partial functions. Both variables are initialized to empty. There are three operations in all. The login() operation assigns the single session to a user. Then alloc(ss) allocates a room in relation to the session *ss*, but preference is given to $r1$. The logout() operation terminates the session.

Appendix-A shows a skeleton of TAgency2.mch; it is a more complex version of TAgency1 and it involves non-determinism. The system can handle a number of parallel sessions given by the deferred set SESSION. A user can log into the system through the call `login()` to get an available session which is non-deterministically selected. He can then request to book or unbook a room (operations `book()` and `unbook()`), and makes (or receives) payment through a card (`enterCard()`). Next, the user can get a response from `response_book()` (or `response_unbook()`). Room allocation data is stored in the function `booking`. A user can book multiple rooms. The machine has five non-deterministic operations: login(), enterCard(), retryCard(), response_book() and response_unbook(). For the second and the third operations, when the card is entered or retried, a non-deterministic choice out of $\{valid, wrong\}$ is made. For the response_book(), any room out of the set of available rooms may be allocated. And in case of response_unbook(), any room out of the allocated rooms to the current user is cancelled.

4 The Method I: Deterministic Models

We assume flat B machines without any hierarchy and Java is the language of implementation. We now outline our method in the following steps.

Creation of Probe Operations: For each operation in the machine, a set of probe operations are manually created from the domain knowledge and the operation meaning. The probes will be used in matching similar specification and implementation states. For the operation `alloc(s)`, some possible queries to become its probe operations are:

– Which user made the allocation request? (`probe operation alloc_P1()`)
– How many rooms got allocated so far? (`probe operation alloc_P2()`)

These two probe operations can be encoded in B as follows:
uu $\leftarrow$ alloc_P1(s) = PRE s $\in$ SESSION THEN uu := sess(s) END
count $\leftarrow$ alloc_P2 = BEGIN count := card(booking) END

Signature Generation: The SUT in Java must have a similar signature as the specification; i.e., the SUT will have the same operation names as those in the specification but their parameters would be similar in the following sense:

– If a model parameter type is either numeric or boolean it becomes `int` and `boolean` in the implementation respectively.
– For any other model parameter of type PP, we treat it as an object of a class PP in the implementation.

For instance, the login() will have the signature `'void login(USER uu)'`. We also create signatures of the probe operations; for instance, alloc_P1() will be of `'USER alloc_P1(SESSION s)'`. The SUT also implements SESSION, USER and ROOM as Java classes. It is expected that the developer while writing the SUT preserves the signatures of the B operations and their probes.

Generation of Operation Instances: We perform a DNF based analysis over the operation preconditions in order to obtain operation instances. However, operation preconditions in B are relatively simpler; therefore, in order to obtain interesting partitions, we add tautologies through conjunction to the precondition as per the following rules.

– If an operation has an `IF` like `'IF C THEN S1 ELSE S2'`, then add the tautology $(C \vee \neg C)$ to the precondition through conjunction. For `'IF C THEN (IF C1 THEN S1 ELSE S2) ELSE S2'`, we add `(C` $\wedge$ `(C1` $\vee \neg C1$`))` $\vee \neg C$
– If the operation has a SELECT with branch conditions $C_1, \ldots, C_k$ then add to the precondition: $C_1 \vee C_2 \vee \ldots C_k \vee (\neg(C_1 \vee C_2 \vee \ldots C_k))$
– If set S occurs in the operation, then add to the precondition: $S = \emptyset \vee S \neq \emptyset$ (similar tautologies can also be added for other data constructs.)

Obtain the DNF of the modified precondition. The non-contradictory disjuncts are used for creating operation instances. Some instances of alloc() are shown below. From now onwards, by operations we will mean operation instances.

alloc$_1$(s): $s \in SESSION \wedge sess \neq \emptyset \wedge booking = \emptyset$
alloc$_2$(s): $s \in SESSION \wedge sess \neq \emptyset \wedge booking \neq \emptyset \wedge dom(booking) \neq \{r1, r2\}$
$\wedge\ booking(r_1) = sess(s)$
alloc$_3$(s): $s \in SESSION \wedge sess \neq \emptyset \wedge booking \neq \emptyset \wedge dom(booking) \neq \{r1, r2\}$
$\wedge\ booking(r_1) \neq sess(s)$

It is easy to see that for each control paths in an operation, we have an operation instance. This means that if are able to generate test cases for each of the operation instances, then the branches within the original operations are also covered.

Creation of a Coverage graph: Our testing criterion is to test each operation at least once; therefore, we try to generate a finite coverage graph so that each operation instance appears at least once. However, we may not be able to cover all operations because: (a) an operation may be infeasible, (b) a certain initialisation may prevent an operation from appearing or (c) an operation may not appear within the finite dimension of the graph. We now outline our construction process in the following steps. Figure 2 shows a coverage graph for TAgency1.mch. The probe calls and their results are shown within the dotted regions.

– **Step 1:** Create an initial node (the root) in which the variables of the B machine get the assignments of the INITIALISATION clause.
– **Step 2:** Take any node in the graph called `source` where all the state variables are already available as ground terms. If the precondition of a non-probe operation holds at source, apply this operation to obtain the target state. Create a new node for target state only if an identical state does not already exist. Label the edge (source,target) with the operation call.
– **Step 3:** For each probe operation `pop()` of OP(), make a call to it at the target state to obtain the result `res`. Attach to the edge just created the pair (`pop(), res`). If enough coverage has not been done then jump to Step 2.

Note that our method can be tuned to many other testing criteria; the graph creation process needs to be changed accordingly.

Generation of Test Sequences: We traverse the coverage graph to generate starting from the initial state a set of paths (or operation sequences) so that each operation is covered. We do not present such an algorithm here; one such algorithm has been given in [21]. It is important that while obtaining a test sequence, we do not go around a loop. And further, the problem being NP-complete [8], we only get a sub-optimal solution. In Figure 2, the path shown by the dashed lines is a test sequence.

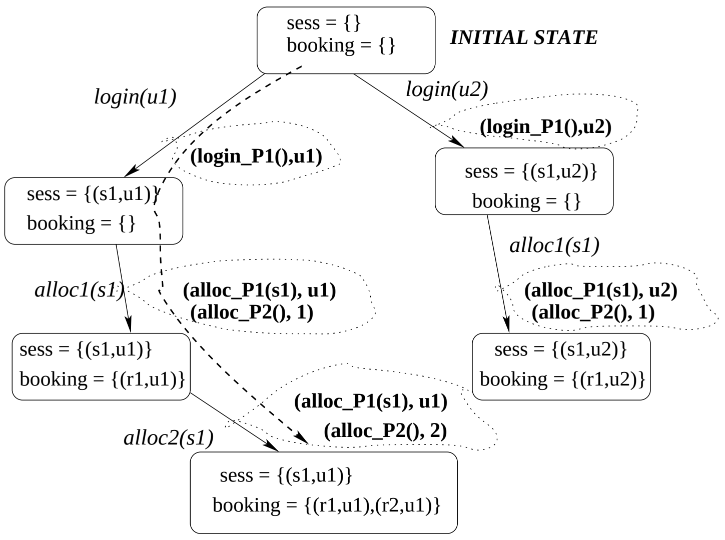

Fig. 2. A coverage graph for TAgency1.mch; dotted path shows one test case

Generation of a test driver: The test driver generator takes a set of test cases and generate a code fragment in Java. This code when executed in a testing context will infer whether the SUT has passed the test cases. At present we consider code for a single test case; multiple test cases can be executed by assuming an initialize() operation to take control back to the initial state. When a new test case is executed after re-initialization, the operation parameters are freshly created; therefore, they are not in conflict with the parameters of the previous runs. By testing context, we mean the following:

– If `MM.mch` is the B machine, then the SUT defines a class with name `MM` and creates an object of the same class, say `mmo`. It is expected that class MM has all operations of the machine as methods with similar signatures.
– In any test case, if an operation has a parameter `pp` as an element of set `PP`, then SUT must have the object `pp` of class `PP`.

Refer to Figure 3(a). (N, N') is an edge in the test sequence with $'res \leftarrow OP(..)'$ as the operation call. This edge has also k probe calls along with their results. The corresponding code in Java has been shown in Figure 3(b). $C[N]$ stand for the code generated at node N. First a call to `mmo.OP(...)` is made. Then we obtain the results of the probe operations from the SUT which are compared with the results of the same operations stored in the test case; this comparison is performed by Java assertions. In (b), `res_1,..., res_k` are the temporary variables to receive the probe results.

Table 2 shows the code fragment in relation to the test case shown in Figure 2. The testing context provides object TA1 of class TAgency1, objects u_1 and u_2 of class USER, objects r_1 and r_2 of class ROOM, and object s_1 of class SESSION.

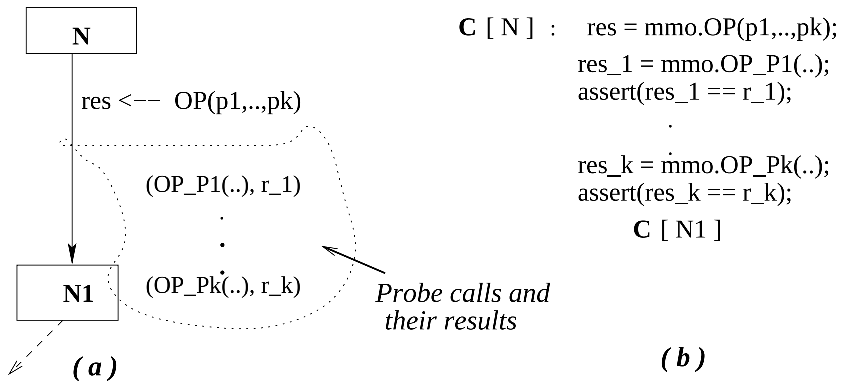

Fig. 3. Code generation from a test case

With these, the code in Table 2 if runs without any assertion violation it would mean that the SUT has passed the test case. Note that the generation of the code in Table 2 can easily be automated. It is to be further noted that the testing context must be provided by the implementor because it involves some design decisions like defining the constructors of various classes.

```
USER login_t1, alloc_t1; int alloc_t2;
TA1.login(u1); login_t1= TA1.login_P1();
     assert(login_t1==u1);
TA1.alloc(s1); alloc_t1= TA1.alloc_P1(s1);
     assert(alloc_t1==u1);
alloc_t2 = TA1.alloc_P2(s1);
     assert(alloc_t2== 1);
TA1.alloc(s1); alloc_t1= TA1.alloc_P1(s1);
     assert(alloc_t1==u1);
alloc_t2 = TA1.alloc_P2(s1);
     assert(alloc_t2== 2);
```

Table 2. Code for the test case in Figure 2

5 The Method II: Non-deterministic Models

The first three steps – generation of probe operations, signature file and the operation instances – of the method outlined in the last section, remain identical for non-deterministic B models. In addition, the process of attaching probe operation calls and their results to an edge in the coverage graph also remains the same. We will discuss the remaining steps here.

OP(..)= ...	br ← OP(..)= ...
SELECT C1 THEN S_1	SELECT C1 THEN S_1 || br := 1
...	...
WHEN C_k THEN S_k END	WHEN C_k THEN S_k || br := k END

Table 3. Making internal choices observable: SELECT statement

There are two primary categories of non-determinism in B [1]: unbounded choice through the ANY statement and bounded choice through the SELECT statement, both having the following syntax respectively.

```
ANY $x_1, \ldots, x_k$ WHERE              SELECT $C_1$ THEN $S_1$
    $P(x_1, \ldots, x_k)$                           ...
THEN $S$ END                              WHEN $C_k$ THEN $S_k$ END
```

The ANY statement makes k non-deterministic choices satisfying the predicate $P(x_1, \ldots, x_k)$ which are used to perform the substitution S. For a non-deterministic SELECT, the branching conditions $C_1, \ldots C_k$ do overlap; and then one valid branch is selected in a non-deterministic way. In addition to SELECT and ANY, B supports non-deterministic assignments in initializations with the syntax $x\ :\in\ S$ meaning that x is given any element of S. However, this statement can always be converted to: `ANY` y `WHERE` $y \in S$ `THEN` $x := y$ `END`.

5.1 Pre-processing of the B model

We make the internal choice – within an ANY statement or the branch selection in SELECT – visible by making the associated B operations more observable. This we do by introducing additional result parameters. Refer to the enterCard() in Appendix-A. We have added a result parameter to observe the internal choice made by the ANY statement. Similarly, we have also added a result parameter to login() to make its non-deterministic choice visible. We also make the branch choice that a non-deterministic SELECT makes observable by introducing an additional result parameter (refer to Table 3). We term the constraint under which a choice is made as *choice predicate*. For ANY, it is the constraint within the WHERE clause. For SELECT, we define it to be $br \in \{1, .., k\}$, where br is the output variable introduced to capture which branch the SUT would select (refer to Table 3), and k is the number of branches.

5.2 Coverage graph for a non-deterministic model

Our convention is that whenever we make a call to a non-deterministic operation then we select a fresh variable in place of the choice and restrict it by the choice predicate. We will refer to this fresh variable as a `choice variable`. A choice variable once created can be used as a parameter in subsequent invocations as long as it satisfies the typing rules.

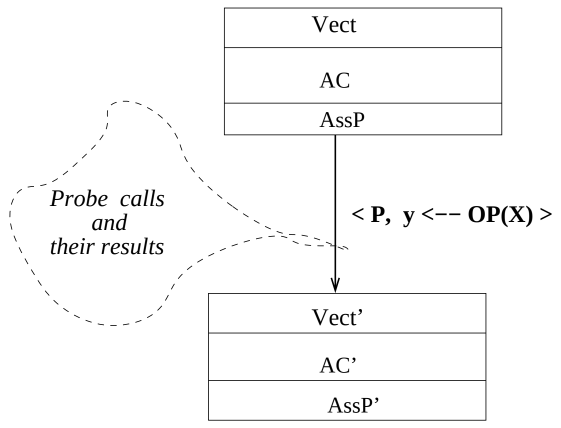

Fig. 4. Application of a non-deterministic operation

A node in the coverage graph is a tuple $< Vect, AC, AssP >$, where $Vect$ is the state vector to store the bindings of expressions to state variables; these expressions may contain choice variables as sub-terms. AC is the set of accumulated constraints which essentially restricts the choice variables occurring in the expressions in $Vect$. $AssP$ is an assertion which results from an application of a non-deterministic operation. For a node N, we refer to its fields by the dot notation such as $N.Vect, N.AC$ etc.

Figure 4 shows an edge in the coverage graph, where $< Vect, AC, AssP >$ constitute the source node. Let application of call `OP(X)` at the source would give us the target node $< Vect', AC', AssP' >$. The edge between the source and the target is labeled with $< P, y \leftarrow OP(X) >$. The derivation process is as follows:

- `P` is a predicate to check that `OP(X)` is applicable at $< Vect, AC, AssP >$. If $pre(OP(X))$ is the precondition of OP(X) then `P` is an expression over the choice variables occurring in $Vect$ and is equivalent to $AC \wedge pre(OP(X))$ or its boolean simplification. We call it the Precondition Satisfaction Predicate (or PSP) which being `false` would mean that `OP(X)` is not applicable.
- If `OP()` is a non-deterministic operation, y stands for the choice variable selected in place of the internal non-deterministic choice. If `cc` is the internal choice in OP(), and *cpred* is the choice predicate, then $AssP'$ is the reduced form of the constraint $cpred[y/cc]$; i.e., the substitution of y in place of the free occurrences of cc in *cpred*.
- $Vect'$ is the reduced form of $Vect[body(OP(X))]$ where $body(OP(X))$ is the substitution in relation to $OP(X)$.
- AC' is the accumulated constraint of the target node and is equivalent to the reduced form of $(AC \wedge P \wedge AssP')$. Note that AC' always includes $AssP'$. We maintain $AssP'$ separately to be referred to by the test case generator.

For the initial node, AC is initialized to the constraints made out of the set declarations and the constraints. Figure 5 shows a part of the coverage graph for the B machine TAgency2. The node marked '1' is the initial node. Its AC field is initialized to $AC0$ as given in the figure. The $Vect$ field here corresponds to the INITIALIZATION clause of the machine. $AssP$ is given the trivial value of $true$. In node 2, ZS_1 represents the non-deterministically selected session identifier. Now consider the call of responseBook$_1$(ZS1) at node 3. Observe how the choice variable $ZS1$ has been used as a parameter. Now consider the application of the following operation instance at node 3:

```
rstatus ← responseBook$_1$(sid) =
PRE $sid \in SESSION \wedge sid \in dom(session) \wedge$
        $s\_req(sid) = book \ \wedge \ s\_state(sid) = s4 \wedge$
        $s\_card(sid) = valid \wedge dom(booking) \subset (ROOM - null_R)$
THEN
        ANY rr WHERE rr$\in (ROOM - null\_R) - dom(booking)$
        THEN booking(rr) := sess(sid) || rstatus := rr END || ...
END
```

The predicate $AC3 \wedge pre(responseBook_1(ZS1))$ reduces to $ZC1 = valid$ to become the PSP of the current call. If $ZR1$ is the choice variable due to ANY, then $(rr \in (ROOM - null_R) - dom(booking))\,[ZR1/rr]$ reduces to $ZR1 \in (ROOM - null_R)$ to becomes the AssP of the target node. Substitution of the operation body over the Vector of the source node, gives us the new Vector. Finally, $AC3$ augmented with the PSP and the AssP becomes the AC of the target node.

5.3 Test cases from non-deterministic models

As in the case of deterministic models, the coverage graph could be traversed to generate a set of linear test cases. We will term those as basic test cases; they will be combined to form adaptive test cases. If we treat a basic test case as a test case proper, then consider the case when SUT control encounters an edge with a non-trivial PSP and it does not hold. For example, in Figure 5, if edge $(ZC1 = valid,\ ZC1 \leftarrow responseBook(ZS1))$ occurs in a basic test case, then $ZC1 = valid$ could be false, and then there is no point in following this edge any further. In the worst case scenario, we may not be able to test any of the basic test cases into completion. Adaptive test cases are introduced precisely for this purpose. An adaptive test case in the coverage graph is a subgraph in the form of a tree with the following properties:

- Its root is same as the root of the coverage graph.
- The paths from the root to the leaves are mutually exclusive in that at any non-leaf node, the PSPs of its outgoing edges are mutually contradictory. In this way we would be able to test all the paths of the test case by a single threaded test driver.

From this definition, it should be clear that given a set of basic test cases as paths in the coverage graph, we can carve out a set of adaptive test cases. One such

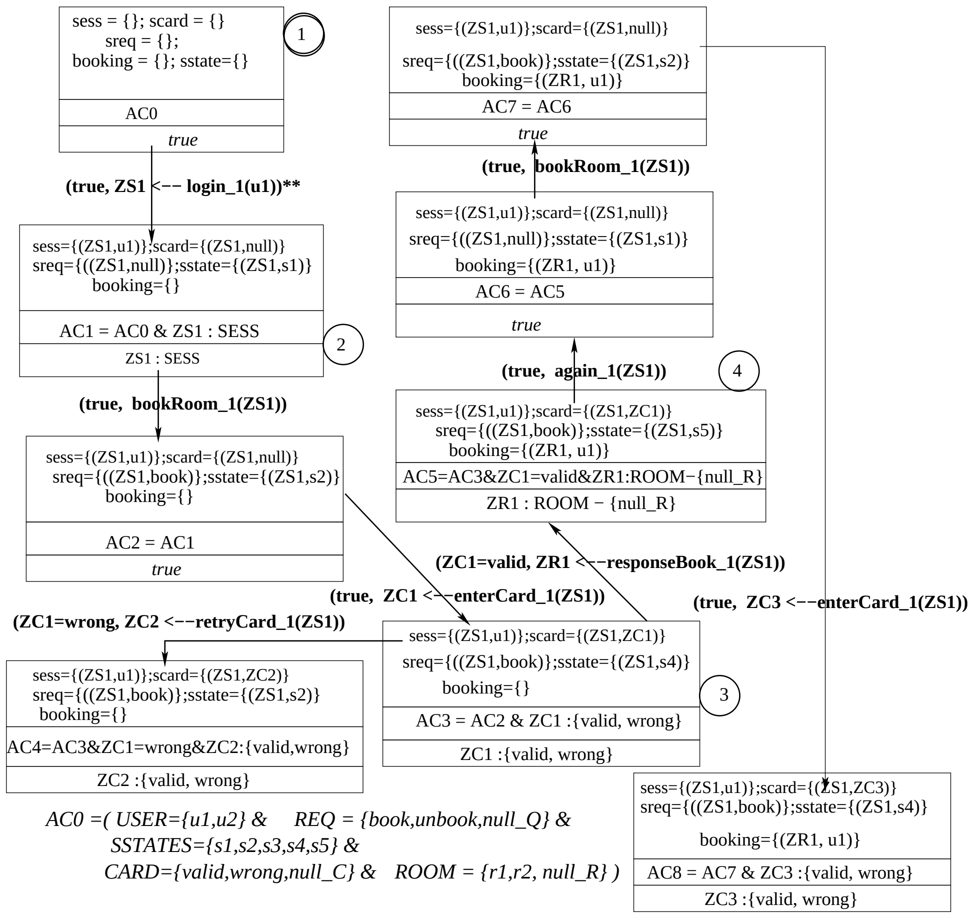
sess = {}; scard = {}
sreq = {};
booking = {}; sstate={}
AC0
true
1
(true, ZS1 <−− login_1(u1))**
sess={(ZS1,u1)};scard={(ZS1,null)}
sreq={((ZS1,null)};sstate={(ZS1,s1)}
booking={}
AC1 = AC0 & ZS1 : SESS
ZS1 : SESS
2
(true, bookRoom_1(ZS1))
sess={(ZS1,u1)};scard={(ZS1,null)}
sreq={((ZS1,book)};sstate={(ZS1,s2)}
booking={}
AC2 = AC1
true
(true, ZC1 <−−enterCard_1(ZS1))
(ZC1=wrong, ZC2 <−−retryCard_1(ZS1))
sess={(ZS1,u1)};scard={(ZS1,ZC2)}
sreq={((ZS1,book)};sstate={(ZS1,s2)}
booking={}
AC4=AC3&ZC1=wrong&ZC2:{valid,wrong}
ZC2 :{valid, wrong}
sess={(ZS1,u1)};scard={(ZS1,ZC1)}
sreq={((ZS1,book)};sstate={(ZS1,s4)}
booking={}
AC3 = AC2 & ZC1 :{valid, wrong}
ZC1 :{valid, wrong}
3
(ZC1=valid, ZR1 <−−responseBook_1(ZS1))
sess={(ZS1,u1)};scard={(ZS1,ZC1)}
sreq={((ZS1,book)};sstate={(ZS1,s5)}
booking={(ZR1, u1)}
AC5=AC3&ZC1=valid&ZR1:ROOM−{null_R}
ZR1 : ROOM − {null_R}
4
(true, again_1(ZS1))
sess={(ZS1,u1)};scard={(ZS1,null)}
sreq={((ZS1,null)};sstate={(ZS1,s1)}
booking={(ZR1, u1)}
AC6 = AC5
true
(true, bookRoom_1(ZS1))
sess={(ZS1,u1)};scard={(ZS1,null)}
sreq={((ZS1,book)};sstate={(ZS1,s2)}
booking={(ZR1, u1)}
AC7 = AC6
true
(true, ZC3 <−−enterCard_1(ZS1))
sess={(ZS1,u1)};scard={(ZS1,ZC3)}
sreq={((ZS1,book)};sstate={(ZS1,s4)}
booking={(ZR1, u1)}
AC8 = AC7 & ZC3 :{valid, wrong}
ZC3 :{valid, wrong}
AC0 =(USER={u1,u2} & REQ = {book,unbook,null_Q} &
SSTATES={s1,s2,s3,s4,s5} &
CARD={valid,wrong,null_C} & ROOM = {r1,r2, null_R})

Fig. 5. Part of the coverage graph for TAgency2.mch

algorithm is given in [20]. Refer to Figure 5. In this tree all paths from the root to leaves can be seen as basic test cases. Only the node marked 3 has outgoing edges with non-trivial PSPs. The PSPs of the two outgoing edges of this node are $ZC1 = valid$ and $ZC1 = wrong$, and hence mutually contradictory; so, the whole tree in the figure produces the single adaptive test case.

5.4 Test Driver Generation

Since the elements of an enumerated set is available in the model, the test driver can have control over its range. If there is a need to check the range of ROOM, it can be done explicitly. But the range of a deferred set like SESSION, cannot be checked. When the operation login() is called from SUT the system depending on availability may or may not be able to allocate a session for the user to

term	Reduced Terms	condition
NULL	`null`	null object reference
$dom(R)$	$\{s_1,\ldots,s_k\}$	$R=\{(s_1,t_1),\ldots,(s_k,t_k)\}$
$ran(R)$	$\{t_1,\ldots,t_k\}$	$R=\{(s_1,t_1),\ldots,(s_k,t_k)\}$
R^{-1}	$\{(t_1,s_1)\ldots,(t_k,s_k)\}$	$R=\{(s_1,t_1),\ldots,(s_k,t_k)\}$
$F(s_i)$	t_i	$F=\{(s_1,t_1),\ldots,(s_k,t_k)\}$; F is a function

Table 4. Schema Rules for reducing some B terms

work with. When it allocates a session there is no need to check the model predicate $ZS1 \in SESSION$ because it would be trivially satisfied by the type checking rule of Java. But if a null object reference is returned then from the view point of testing there is no need to check the subsequent operation calls. In summary, when an element of a deferred set is obtained it must be checked for non-nullness. To signify that in the coverage graph of Figure 5, we have marked the edge joining nodes 1 and 2 with '**'.

B predicates to Java Assertions While generating a coverage graph, we obtain predicates which involves choice variables. Let us call these *graph predicates.* They are different from model predicates, the predicates occurring in a B model. We require a Set Constraint Solver (SCS) to translate any graph predicate into Java assertions. The development of such a SCS in general is a challenging task. In this paper we consider a simple SCS and so we put restrictions on model predicates which in turn restrict the graph predicates. If x is an internal choice — like cc in enterCard() or rr in response_book() — the syntactic constraint on model predicates is that, it can be of the form: $x \in S \wedge P$, where S is either a deferred set or an enumerated set or a basic set (Bool or Int), and P includes finite number of (state) variables and constants. For instance, in login(), the model predicate $sid \in SESSION \wedge sid \notin dom(sess)$ is of this form.

SCS performs two main tasks: (a) to evaluate a graph predicate — involving choice variables, sets, relation, function etc. — to obtain `PSPs`; this can be done by extending Constraint Logic Programming (CLP) to sets, relations and functions; and (b) to reduce the PSPs and the AssPs into Java assertions.

Table 4 shows the reduction rules for some terms in graph predicates. Table 5 shows the rules to reduce some graph predicates to Java assertions by a translation function γ. Note that each s_i or t_i stands for a term occurring in graph predicates. We assume that the reduction of terms can be performed by syntactic checking only. For instance, the reduction of $\{(ZS1,u1),(ZS2,u2)\}(ZS2)$ can be done by syntax checking, whereas that of $\{(ZS1,u1),(ZS2,u2)\}(ZS8)$ given that $ZS8 \in \{ZS1,ZS2\}$ can not be performed by syntactic checking alone. Though the latter terms can be reduced to Java by defining more and more rules, we do not consider them here.

Test Driver Generation Algorithm The test driver generation algorithm for adaptive test cases is trivial. An adaptive test case is in the form of a tree.

predicates	to Java	condition
$\gamma[TRUE]$	`true`	boolean constant in Java
$\gamma[FALSE]$	`false`	boolean constant in Java
$\gamma[(s_1,t_1) = (s_2,t_2)]$	$(s_1 == t_1)$ && $(s_2 == t_2)$	
$\gamma[X \in S]$	$(X == s_1) \,\|\|\, \ldots \,\|\|\, (X == s_k)$	$S = \{s_1,\ldots,s_k\}$
$\gamma[X \in S]$	!(X == `null`)	S is a deferred set
$\gamma[X \in T_1 \cup T_2]$	$\gamma[(X \in T_1)] \,\|\|\, \gamma[(X \in T_2)]$	
$\gamma[X \in T_1 \cap T_2]$	$\gamma[X \in T_1]$ && $\gamma[X \in T_2]$	
$\gamma[A \subseteq B]$	$\gamma[s_1 \in B]$ && ... &&$\gamma[s_k \in B]$	$A = \{s_1,\ldots,s_k\}$
$\gamma[\forall x \in S.P(x)]$	$\gamma[P(s_1)]$ && ... && $\gamma[P(s_k)]$	$S = \{s_1,\ldots,s_k\}$
$\gamma[\exists x \in S.P(x)]$	$\gamma[P(s_1)] \,\|\|\, \ldots \,\|\|\, \gamma[P(s_k)]$	$S = \{s_1,\ldots,s_k\}$

Table 5. Schema rules for transforming some B predicates

A junction nodes gets transformed to an `if-elseif-else` statement, and the PSPs of the branches become the `if (or elseif)` conditions. In addition, there has to be an `else` clause because the set of the PSPs may not be exhaustive. If during testing, SUT control enters this `else branch`, this would mean that we cannot carry out testing any further; appropriate message can be given to the tester in this case. A detailed discussion on this situation has been given in [20]. Leaving aside the PSPs, in relation to a branch in the test case, we encounter a sequence of operation applications which may have assertions (AssPs) and they become Java assertions in the code. We show this by generating code for the single adaptive test case of Figure 5; the code has been shown below. Observe how the branching in the code corresponds to the branching in the adaptive test case. We do not show the testing context here since it remains the same as earlier.

```
SESSION ZS1; ROOM ZR1; CARD ZC1,ZC2;
USER temp_u; int temp_c;
ZS1 = TA2.login(u1); assert(ZS1 != null);
TA2.bookRoom(ZS1); ZC1 = TA2.enterCard(ZS1);
assert(ZC1 == valid || ZC1 == wrong);
if (ZC1 == wrong) { // PSP of 1st branch holds
     ZC2=TA2.retryCard(ZS1);
     assert(ZC5 == valid || ZC5 == wrong);
} else if (ZC1 == valid) { // PSP of next branch holds
     ZR1=TA2.responseRoom(ZS1);
     assert(ZR1 == u1 || ZR1 == u2 || ZR1 == u3);
     temp_u = TA2.whichUser(ZS1); assert(temp_u == u1);
     temp_c = TA2.numOfRoomsBooked(); assert(temp_c == 1);
} else {
     Sys.out.println("SUT control deviated; testing stops");
}
```

6 Implementation

ProB is a model checking and animation tool for B machines [16]. ProB includes a fully automatic animator written in SicStus Prolog. An extension of ProB will be our implementation platform. As of now, we have implemented testing of a SUT written in accordance with a deterministic B model. We have made the following steps automatic so far.

– Given a B operation with its precondition enriched with tautologies (refer to Section 4) we generate a set of operation instances.
– Given a B model, we generate a Java signature template for all B operations.
– After restricting the sets to be finite, the current tool automatically creates a coverage graph (now no deferred sets).
– We traverse the coverage graph to generate a set of preset test cases; this is because we consider deterministic models only.
– Given a set of test cases, we generate a test driver which when augmented with the testing context performs automatic testing of a Java implementation.

Now the development of a SCS to handle a subset of B predicates is under progress. This will enable us in handling non-deterministic models.

7 Discussion

– Making the whole testing cycle automatic in presence of non-determinism is an important contribution of our work. It is often the case that non-determinism in B is gradually refined out in the B refinement process, but our strategy does not assume the implementation to be deterministic. Were the implementation non-deterministic, our method would work without any change. Existing testing tools like `BZ-TT` [2] avoid the issues related to non-determinism.
– The tester (or the specifier) has to write a set of probe operations for each B operation. In this paper, we have kept this step outside of the scope of our testing cycle. However, we believe writing a set of probe operation from the domain knowledge and the intention of the operation is too ad-hoc an approach. Probe operations should be made finer by generating them from the model in a systematic manner. One possibility is to define abstract functions mapping the concrete states of the Java program to the abstraction level in the B machine. This issue requires further research.
– Our method can create a coverage graph in presence of deferred sets.
– The problem of obtaining a PSP out of a set of B constraints requires to solve a set of set constraints; this being a variant of the satisfiability problem is NP-complete [8]. Good specification practices recommend to write smaller and simpler operations. In this case, we expect the problem size would remain small and then a CLP solver should be able to do the job. This issue needs further investigation.

– For the development of a SCS, we took a simple subset of the B predicates. However, this is a useful subset since we have examined a number of examples and seen that this subset is sufficient. The examples include B models for a larger version of the travel agency example, the router component of a Network-on-Chip system and a component of a TV teletext system. The problem of implementing a robust SCS will require further research.

8 Conclusion

We have discussed how a test driver in the form of a Java program can be mechanically generated from a B model, possibly non-deterministic, to perform automatic testing. The constraints arising out of non-deterministic choices and oracle information matching become Java assertions in the test driver which if runs without any assertion violation would mean that the implementation has passed the test cases. Our approach can generate the test driver much before the implementation; however, it assumed that the implementation should adhere to the Java signature template obtained from the model. We have made comparisons of our research with existing work, the important contributions being the handling of non-determinism.

Acknowledgement: We would like to thank Linas Laibinis for going through an earlier version of this work and offering many a useful suggestion. The comments of the anonymous reviewers also helped us in improving the quality of this paper.

References

1. Abrial J.-R. (1996). *The B–Book: Assigning Programs to Meanings*, Cambridge University Press.
2. Bernard E., Legeard B., Luck X., Peureux F. (2004). Generation of test sequences from formal specifications: GSM 11-11 standard case study, *Software Practice and Experience*, Volume 34 (10) , pp. 915 - 948.
3. Colin, S., Legeard, B., Peureux, F. (2004). Preamble computation in automated test case generation using constraint logic programming, Software Testing Verification and Reliability, John Wiley, Vol. 14: 213–235.
4. Dick, J.; Faivre, A. (1993). Automating the Generation and Sequencing of Test Cases from Model-based Specifications, Proc. of the FME'03, LNCS 670. pp. 268–284.
5. Dalal, S.R., Jain A., Karunanithi, N., Leaton J.M., Lott C.M., Patton G.C., Horowitz B.M. (1999). Model Based Testing in Practice, Proc. of ICSE '99.
6. El-Far, I.K., Whittaker, J.A. (2001). Model Based Software Testing, *Encyclopedia on Software Engineering (Ed. J.J. Marciniak)*, John Wiley.
7. Gannon, J.D., Hamlet R.G., Mills, H.D. (1987). Theory of modules, IEEE Transactions on Software Engineering, 13(7):820–829.
8. Garey, M.R., Johnson, D.S. (1979). Computers and Intractability, W. H. Freeman and Company.

9. Gurevich, Y. (2000). Sequential Abstract-State Machines Capture Sequential Programs, ACM Transaction on Computational Logic, Vol 1(1): 77–111.
10. Hierons R.M. (2004). Testing from a Non-deterministic Finite State Machine using Adaptive State Counting, IEEE Transactions on Computers, Vol 53(10), pp. 1330-1342.
11. Hierons R.M. (2006). Applying Adaptive Test Cases to Non-deterministic Implementations, *Information Processing Letters*, 98(2006): 56–60.
12. Jones, C.B. (1990). Systematic Software Development using VDM (2nd Edn), Prentice Hall.
13. Lee, D.; Yannakakis, M. (1996). Principles and Methods of Testing Finite State Machines: A survey, Proc. of the IEEE, 80(8): 1090–1123.
14. Legeard, B., Peureux, F., Utting, M. (2002). Automatic Boundary Testing from Z and B, Formal Methods Europe '02, LNCS Volume 2391, Springer, pp. 21–40.
15. Legeard, B., Peureux, F., Utting, M. (2004). Controlling test case explosion in test generation from B formal models, Software Testing, Verification and Reliability, John Wiley, pp.81–103.
16. Leuschel, M., Butler M. (2005). ProB: A Model Checker for B, Proc. FME'03, LNCS Volume 2805, Springer, pp. 855–874.
17. Nachmanson L., Veanes M., Schulte W., Tillmann N. and Grieskamp W.(2004). Optimal Strategies for Testing Nondeterministic Systems, ACM ISSTA'04, Boston, July 2004.
18. Panzl, D.J. (1978). Automatic Software Test Drivers, IEEE Computer, 11(4).
19. Richardson D.J., Leif Aha A., O'Malley T.O. (1992). Specification-based Test Oracles for Reactive Systems, Proc. of the 14th ICSE, Melbourne, pp. 105–118.
20. Satpathy, M., Butler,M., Ramesh, S.,Leuschel, M. (2006). Automatic Testing of Formal Specifications, Technical Report 792, Abo Akademi University, Turku, Finland. (available at: `http://www.tucs.fi/publications`)
21. Satpathy, M., Leuschel, M., Butler,M. (2005). ProTest: An Automatic Test Environment for B Specifications, *Electronic Notes on TCS (ENTCS)*, 111, pp: 113–136.
22. Spivey, J.M. (1988). *Understanding Z*, Cambridge University Press.
23. Yannakakis M., Lee D.(1995). Testing Finite State Machines: Fault Detection, *Journal of Computer and System Sciences*, Vol. 50, pp.209–277.
24. Zhang F. and Cheung T.(2003). Optimal Transfer Trees and Distinguishing Trees for Testing Observable Nondeterministic Finite State Machines, IEEE Transactions on Software Engineering, Vol 29(1): 1–14.
25. Zhu, H., Hall P.A.V., May J.H.R. (1997). Software Unit Test Coverage and Adequacy, *ACM Computing Surveys*, 29(4):366–427.

Appendix – A

```
MACHINE TAgency2
SETS SESSION; /* A deferred set */
    USER = $\{u1, u2\}$; REQ=$\{book, unbook, null\_r\}$;
    SSTATES= $\{s1, s2, s3, s4, s5\}$;
    CARD = $\{valid, wrong, null\_C\}$; ROOM = $\{r1, r2, null\_R\}$
VARIABLES $sess, scard, sstate, sreq, booking$ /* all partial functions */
INVARIANT $sess \in SESSION \mathrel{+\!\!\rightarrow} USER \wedge scard \in SESSION \mathrel{+\!\!\rightarrow} CARD \wedge$
        $sstate \in SESSION \mathrel{+\!\!\rightarrow} SSTATES \wedge sreq \in SESSION \mathrel{+\!\!\rightarrow} REQ \wedge$
        $booking \in (ROOMS - \{null\_R\}) \mathrel{+\!\!\rightarrow} USER \wedge \ldots$
INITIALISATION $sess, scard, sstate, sreq, booking$ := $\emptyset, \emptyset, \emptyset, \emptyset, \emptyset$
```

```
OPERATIONS
    $id \longleftarrow login(uu)$ = PRE $uu \in USER$ THEN
        ANY $sid$ WHERE $sid \in SESSION \wedge sid \notin dom(sess)$ THEN
            $sess(sid) := uu \mid\mid s\_card(sid) := null\_C \mid\mid$
            $s\_state(sid) := s1 \mid\mid s\_req(sid) := null\_r$
            $\mid\mid id := sid$
        END END;
    $bookRoom(sid)$ = PRE $sid \in SESSION \wedge sid \in dom(sess) \wedge$
            $s\_state(sid) = s1 \wedge s\_req(sid) = null_r$
        THEN $s\_state(sid) := s2 \mid\mid s\_req(sid) := book$
        END;
    $unbookRoom(sid) = \ldots$
    $cstatus \longleftarrow enterCard(sid)$ = PRE $sid \in SESSION \wedge sid \in dom(sess) \wedge$
        $s\_state(sid) \in \{s2, s3\}$
    THEN $s\_state(sid) := s4 \mid\mid$
        ANY $cc$ WHERE $cc \in \{valid, wrong\}$
        THEN $s\_card(sid) := cc \mid\mid cstatus := cc$
    END END;
    $cstatus \longleftarrow retryCard(sid)$ = PRE $sid \in SESSION \wedge sid \in dom(sess) \wedge$
        $s\_state(sid) = s4 \wedge s\_card(sid) = wrong$
    THEN ANY $cc$ WHERE $cc \in \{valid, wrong\}$
            THEN $s\_card(sid) := cc \mid\mid cstatus := cc$
        END END;
    $rstatus \longleftarrow response\_book(sid)$ =
        PRE $sid \in SESSION \wedge sid \in dom(sess) \wedge s\_req(sid) = book \wedge$
            $s\_state(sid) = s4 \wedge s\_card(sid) = valid$
        THEN $s\_state(sid) := s5 \mid\mid$
        IF $dom(booking) \subset (ROOM - \{null\_R\})$ THEN
            ANY $rr$ WHERE $rr \in (ROOM - \{null\_R\}) - dom(booking)$
            THEN $booking(rr) := sess(sid) \mid\mid rstatus := rr$ END
        ELSE $rstatus := null\_R$
        END
    END;
    $rstatus \longleftarrow response\_unbook(sid) = \ldots$
    $again(sid) = \ldots$
    $logout(sid) = \ldots$
END
```