Model-Based Testing for Embedded Systems

Frédéric Dadeau[1], Fabien Peureux[1], Bruno Legeard[1,2], Régis Tissot[1],
Jacques Julliand[1], Pierre-Alain Masson[1], Fabrice Bouquet[1]

[1] LIFC - University of Franche-Comté – 16 route de Gray – 25030 Besançon cedex, France

[2] Smartesting – 1 rue Alain Savary – 25000 Besançon, France

Chapter 1

Test Generation using Symbolic Animation of Models

In the domain of embedded systems, models are often used either to generate code, possibly after refinement steps, but they also provide a functional view of the system that will be used to produce black box test cases, without considering the actual details of implementation of this system. In the latter process, the tests are generated by appling given test selection criteria on the model. These test cases are then played on the system and the results obtained are compared with the results predicted by the model, in order to ensure the conformance between the concrete system and its abstract representation. Test selection criteria aim at achieving a reasonable coverage of the functionalities or requirements of the system, without involving a heavyweight human intervention.

We present in this chapter works on the B notation as a support for the model design, intermediate verification, and test generation. In B machines, the data model is described using abstract data types (such as sets, functions, relations) and the operations are written in a code-like notation based on generalized substitutions. Using a customized animation tool, it is possible to animate the model, i.e., to simulate its execution, in order to ensure that the operations behave as expected w.r.t. the initial informal requirements. Further-

more, this animation process is also used for the generation of test cases, with more or less automation. More precisely, our work focuses on symbolic animation that improves classical model animation by avoiding the enumeration of operation parameters. Parameter values become abstract variables whose values are handled by dedicated tools (provers or solvers). This process has been tool-supported with the BZ-Testing-Tools framework, that has been industrialized and commercialized by the company Smartesting. We present in this chapter the techniques used to perform the symbolic animation of B models using underlying set-theoretical constraint solvers, and we describe two test generation processes based on this process.

The first one employs animation in a fully-automated manner, as a mean for building test cases that reach specific test targets computed so as to satisfy a structural coverage criterion over the operations of the model, also called static test selection criterion. On the contrary, the second one is a scenario-based testing approach, also said to satisfy a dynamic test selection criterion, in which manually-designed scenarios are described as sequences of operations, possibly targeting specific states. These scenarios are then animated in order to produce the test cases. We illustrate the use and the complementarity of these two techniques on the industrial case of a smart card application, named IAS –Identification Authentication Signature– an electronic platform for loading applications on last-generation smart cards.

1.1 Motivations and Overall Approach

In the domain of embedded systems, a model-based approach for design, verification or validation is often required, mainly because these kind of systems are used to be considered as critical [5]. In that sense, a defect can be relatively costly in terms of money or human lives. The key idea is thus to detect the possible dysfunctions as soon as possible. The use of formal models, on which mathematical reasoning can be performed, is therefore an interesting solution. In the context of software testing, the use of formal models makes it possible to achieve an interesting automation of the process, the model being used as a basis from which the test cases are computed. In addition, the model predicts the expected

results, named the oracle, that describe the response that the System Under Test (SUT) should provide (modulo data abstraction). The conformance of the SUT w.r.t. the initial model is based on this oracle.

We rely on the use of behavioral models, that are models describing an abstraction of the system, using state variables, and operations that may be executed, representing a transition function described using generalized substitutions. The idea for generating tests from these models is to animate them, i.e., simulating their execution. The sequences obtained represent abstract test cases that have to be concretized to be run on the system under test. Our approach considers two complementary test generation techniques that use model animation in order to generate the tests. The first one is based on a structural coverage of the operations of the model, and the second is based on dynamic selection criteria using user-defined scenarios.

Before going further into the details of our approach, let us define the perimeter of the embedded systems we target. We consider embedded systems that do not present concurrency, or strong real time constraints (i.e., time constraints that can not be discretized). Indeed, our approach is suitable for validating the functional behaviors of electronic transaction applications, such as smart cards applets, or discrete automotive systems such as front-wipers or cuise controllers.

1.1.1 Context: the B Abstract Machines Notation

Our work focuses on the use of the B notation [1] for the design of the model to be used for testing an embedded system. Several reasons motivate this choice. B is a very convenient notation for modelling embedded systems, grounded on well-defined semantics. It makes it possible to easily express the operations of the system using a functional approach. Thus, each command of the system under test can be modelled by a B operation, that acts as a function updating the state variables. Moreover the operations' syntax displays conditional structures (IF...THEN...ELSE...END) that are similar to any programming language. One of the advantages of B is that it does not require the user to know the complete topology of

the system (compared to automata-based formal notations) which simplifies its usage in the industry. Notice that we do not consider the whole development process described by the B method, starting from an abstract machine and involving successive refinements, that would be useless for test generation purposes (i.e., if the code is generated from the model, there is no need to test the code). Here, we focus on abstract machines; this does not restrict the expressiveness of the language, since a set of refinement can naturally be flatten into a single abstract machine.

B is based on a set-theoretical data model that makes it possible to abstract complex structures using sets, relations (set of pairs) and a large variety of functions (total/partial functions, injections, surjections, bijections), along with numerous set/relational operators. The dynamics of the model, namely the initialization and the operations, are expressed using *Generalized Substitutions*, that describe the possible evolution of the state variables including simple assignments (x := E), parallel assignments (x, y := E, F also written x := E $\parallel$ y := F), conditional assignments (IF Cond THEN Subst_1 ELSE Subst_2 END), bounded choice substitutions (CHOICE Subst_1 OR .... OR Subst_N END) or unbounded choice substitutions (ANY z WHERE Predicate(z) THEN Subst END) (see. [1, p. 227] for a complete list of generalized substitutions).

An abstract machine is organised in clauses that describe: the constants of the system and their associated properties, the state variables and the invariant (containing the data typing information, and actual invariant properties that one wants to see preserved through the possible execution of the machine), the initial state and the atomic state evolution described by the operations.

Figure 1.1 gives an example of a B abstract machine that will be used to illustrate the various concepts presented in the chapter. This machine models an electronic purse, similar as those embedded on smart cards, managing a given amount of money (variable `balance`). A PIN code is also used to identify the card holder (variable `pin`). The holder may try to authenticate using operation `VERIFY_PIN`. Boolean variable `auth` states whether or not the holder is authenticated. A limited number of tries is given for the holder to authenticate (3 in the model). When the user fails to authenticate the number of tries decreases until

```
MACHINE
    purse
SETS
    BOOLEAN = {true, false}
CONSTANTS
     max_tries
PROPERTIES
    max_tries ∈ ℕ ∧ max_tries = 3
VARIABLES
    balance, pin, tries, auth
INVARIANT
    balance ∈ ℕ ∧ balance ≥ 0 ∧ pin ∈ -1..9999 ∧
    tries ∈ 0..max_tries ∧ auth ∈ BOOLEAN ∧  ...
INITIALIZATION
   balance := 0 ||  pin := -1 || tries := max_tries || auth := false
OPERATIONS
   sw ← SET_PIN(p) ≙ ...
   sw ← VERIFY_PIN(p) ≙ ...
   sw ← CREDIT(a) ≙ ...
   sw ← DEBIT(a) ≙ ...
END
```

Figure 1.1: A B abstract machine modelling a simplified electronic purse

reaching 0, corresponding to a state in which the card is definitely blocked (i.e., no command can be successfully invoked). The model provides a small number of operations that make it possible: to set the value of the PIN code (`SET_PIN` operation), to authenticate the holder (`VERIFY_PIN` operation), and to credit the purse (`CREDIT` operation) or to pay a purchase (`DEBIT` operation).

1.1.2 Model-Based Testing Process

We present in this part the use of B as a formal notation that makes it possible to describe the behavior of the system under test. In order to produce the test cases from the model, the B specification is animated using constraint solving techniques. We propose to develop two test generation techniques based on this principle, as depicted in Figure 1.2.

The first technique is fully automated and aims at applying a structural coverage criterion on the operations of the machine so as to derive test cases that are supposed to exercise all the operations of the system, involving decision coverage and data coverage as a boundary

analysis of the state variables. Unfortunately, this automated process show some limitations, that we will illustrate. This leads us to consider a guided technique based on the design of scenarios. Both techniques rely on the use of animation, either to compute the test sequences by a customized state exploration algorithm, or to animate the user-defined scenarios.

These two processes compute test cases that are said to be abstract, since they are expressed at the model level. These test thus need to be concretized to be run on the system under test. To achieve that, the validation engineer has to write an *adaptation layer* that will be in charge of bridging the gap between the abstract and the concrete level (basically model operations are mapped to SUT commands, and abstract data values are translated into concrete data values).

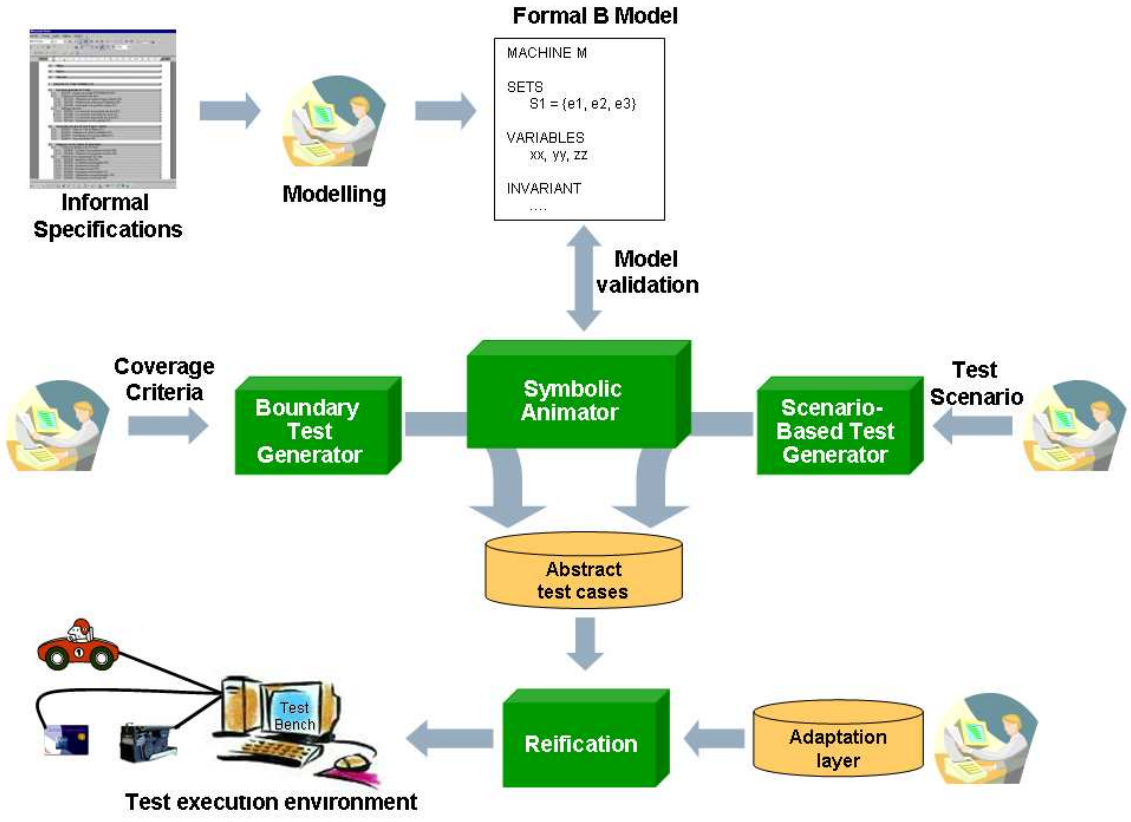

Figure 1.2: Test generation processes based on symbolic animation

Plan of the Chapter

The chapter is organised as follows. Section 1.2 describes the principle of symbolic animation, that will be used in the subsequent sections. The automated boundary test generation technique is presented in section 1.3, whereas the scenario based testing approach is described in Section 1.4. The usefulness and complementarity of these two approaches is illustrated in Section 1.5 on an industrial case studies on smart card applets. Finally, Section 1.6 presents the related work and Section 1.7 concludes and gives an overview of the open issues.

1.2 Principles of Symbolic Animation

Model animation may be used for ensuring that the model behaves as described in the initial requirements. When a B model is animated, the user chooses which operation he wants to invoke. Depending on the current state of the system and the values of the parameters, a dedicated animation tool computes the resulting states that can be obtained. By comparing these states with the informal specification, the user can evaluate its model and correct it if necessary. This process is complementary to the verification that involves properties that have to be formally verified on the model.

The symbolic animation improves the "classical" model animation by giving the possibility to abstract the operation parameters. Once a parameter is abstracted, it is replaced by a symbolic variable that is handled by dedicated constraints solvers. Abstracting all the parameter values turns out to consider each operation as a set of "behaviors", that are the basis from which symbolic animation can be performed [13].

1.2.1 Definition of the Behaviors

A *behavior* is a part of the operation that represents one possible way of executing the operation, in terms of resulting activated effect. Each behavior can be defined as a predicate, representing its activation condition, and a substitution that represents its effect, namely the

evolution of the state variables and the instantiation of the return parameters of the operation. The behaviors are computed as the paths in the control flow graph of the considered B operation, represented as a before-after predicate[1].

Example 1 (Computation of behaviors) *Consider a smart card command, named VERIFY_PIN aiming at checking a PIN code proposed in parameter against the PIN code of the card. As for every smart card commands, this command returns a code, named* `sw` *for status word, that indicates whether the operation succeeded or not, and possibly indicating the cause of the failure. The precondition specifies the typing information on the parameter* `p` *(a four digit number). First, the command can not succeed if there are no remaining tries on the card and if the current PIN code of the card has been previously set. If the PIN codes match, the card holder is authentified, otherwise there are two cases: either there are enough tries on the card, and the returned status word indicates that the PIN is wrong, or the holder has performed his last try, and the status word indicates that the card is now blocked. This operation is given in Figure 1.3, along with its control flow graph representation. This command presents 4 behaviors, that are made of the conjunction of the predicates on the edges of a given path, that is denoted by the sequence of nodes from 1 to 0. For example, behavior [1,2,3,4,0], defined by predicate* $p \in 0..9999 \wedge tries > 0 \wedge pin \neq -1 \wedge p = pin \wedge auth' = true \wedge tries' = max\_tries \wedge sw = ok$ *represents a successful authentication of the card holder. In this predicate,* X' *designates the value of variable* X *after the execution of the operation.*

1.2.2 Use of the Behaviors for the Symbolic Animation

When performing the symbolic animation of a B model, the operation parameters are abstracted and thus, the operations are considered through their behaviors. Each parameter is thus replaced by a symbolic variable whose value is managed by a constraint solver.

Definition 1 (Constraint Satisfaction Problem (CSP)) *A Constraint Satisfaction Problem is a triplet* $\langle X, D, C \rangle$ *in which*

[1] A before-after predicate is a predicate involving state variables before the operation and after, using a primed notation.

```
sw ← VERIFY_PIN(p) ≙
   PRE p ∈ 0..9999 THEN
     IF tries > 0 ∧ pin ≠ -1 THEN
        IF p = pin THEN
           auth := true ||
           tries := max_tries ||
           sw := ok
        ELSE
           tries := tries - 1 ||
           auth := false ||
           IF tries = 1 THEN
              sw := blocked
           ELSE
              sw := wrong_pin
           END
        END
     ELSE
        sw := wrong_mode
     END
   END
```

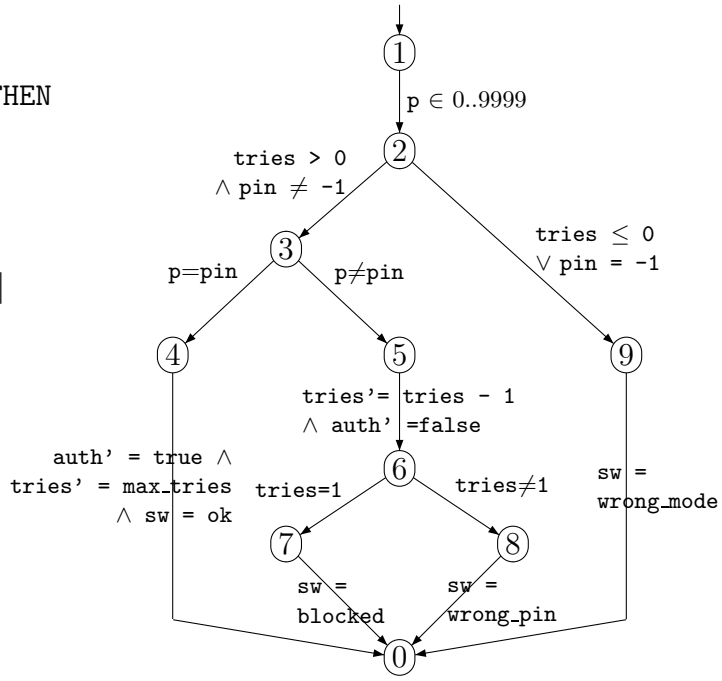

Figure 1.3: B code and control-flow graph of the VERIFY_PIN command

- $X = \{X_1, \ldots, X_N\}$ *is a set of* N *variables*
- $D = \{D_1, \ldots, D_N\}$ *is a set of domains associated to each variable*
- C *is a set of constraints that relate variable values altogether*

A CSP is said to be consistent *if there exists at least one valuation of the variables in* X *that satisfies the constraints of* C*. It is* inconsistent *otherwise.*

Activating a transition from a given state is equivalent to solving a CSP whose variables X are given by the state variables of the current state (i.e., the state from which the transition is activated), the state variables of after state (i.e., the state reached by the activation of the transition) and the parameters of the operation. Accordingly to the B semantics, the domain D of the variables can be found in the invariant of the machine (resp. in the precondition of the operation) for the state variables (resp. for the operation parameters). The contraints C are the predicates composing the behavior that is being activated, enriched with equalities between the before and after variables that are not assigned within the considered behavior.

The feasibility of a transition is defined by the consistency of the CSP associated to the activation of the transition from a given state. From a given (symbolic or concrete) state, the iteration between the possible activable behaviors is given by performing a depth-first exploration of the behavior graph.

Example 2 (Behavior activation) *Consider the activation of the VERIFY_PIN operation given in Example 1. Suppose the activation of this operation from the state* s_1 *defined by:* `tries = 2`, `auth = false`, `pin = 1234`. *Two behaviors can be activated. The first one corresponds to an invocation* `ok` $\leftarrow$ `VERIFY_PIN(1234)` *that covers path [1,2,3,4,0], and produces the following consistent CSP (notice that data domains have been reduced so as to give the most human-readable representation of the corresponding states):*

$$\begin{aligned} CSP_1 \;=\; & \langle \{tries, auth, pin, p, tries', auth', pin'\}, \\ & \{\{2\}, \{false\}, \{1234\}, \{1234\}, \{3\}, \{true\}, \{1234\}\}, \\ & \{Inv, Inv', tries > 0, pin \neq -1, p = pin, tries' = 3, \\ & \qquad auth' = true, pin' = pin\}\rangle \end{aligned} \tag{1.1}$$

where Inv (resp. Inv′) designates the constraints from the machine invariant that apply on the variables before (resp. after) the activation of the behavior. The second activable behavior corresponds to an invocation `wrong_pin` $\leftarrow$ `VERIFY_PIN(`p`)`, *that covers path [1,2,3,5,6,8,0] and produces the following consistent CSP:*

$$\begin{aligned} CSP_2 \;=\; & \langle \{tries, auth, pin, p, tries', auth', pin'\}, \\ & \{\{2\}, \{false\}, \{1234\}, 0..1233 \cup 1235..9999, \{1\}, \{false\}, \{1234\}\}, \\ & \{Inv, Inv', tries > 0, pin \neq -1, p \neq pin, tries' = tries - 1, \\ & \qquad auth' = false, tries \neq 1, pin' = pin\}\rangle \end{aligned} \tag{1.2}$$

State variables may also become symbolic variables, if their after value is related to the value of a symbolic parameter. A variable is said to be symbolic if the domain of the variable contains more than one value. A system state that contains at least one symbolic state variable is said to be a *symbolic state* (by opposition to a concrete state).

Example 3 (Computation of Symbolic States) *Consider a the* `SET_PIN` *operation supposed to set the value of the PIN on a smart card:*

```
sw ← SET_PIN(p) ≙
  PRE p ∈ 0..9999 THEN
     IF pin = -1 THEN  pin := p ‖ sw := ok
     ELSE sw := wrong_mode END
  END
```

From the initial state, in which `auth = false`, `tries = 3` *and* `pin = -1`, *the SET_PIN operation can be activated to produce a symbolic state associated to the following CSP:*

$$\begin{aligned} CSP_0 \;=\; & \langle \{tries, auth, pin, p, tries', auth', pin'\}, \\ & \{\{3\}, \{false\}, \{-1\}, 0..9999, \{3\}, \{false\}, 0..9999\}, \\ & \{Inv, Inv', pin = -1, pin' = p\}\rangle \end{aligned} \tag{1.3}$$

The symbolic animation process works by exploring the successive behaviors of the considered operations. When two operations have to be chained, this process acts as an exploration of the possible combinations of successive behaviors for each operation.

In practice, the selection of the behaviors to be activated is done in a transparent manner and the enumeration of the possible combinations of behaviors chaining is explored using backtracking mechanisms. For animating B models, we use CLPS-BZ [12], a set-theoretical constraint solver written in SICStus Prolog [34] that is able to handle a large subset of the data structures existing in the B machines (sets, relations, functions, integers, atoms, etc.).

Once the sequence has been played, the remaining symbolic parameters can be instantiated by a simple labelling procedure, that consists in solving the constraints system and producing an instantiation of the symbolic variables, obtaining an abstract test case.

It is important to notice that constraint solvers work with an internal representation of constraints (involving constraint graphs and/or polyhedra calculi for relating variable values altogether). Nevertheless, consistency algorithms used to acquire and propagate constraints

are not sufficient to ensure the consistency of a set of constraints, and a labelling procedure always has to be employed to guarantee the existence of solutions in a CSP associated to a symbolic state.

The next two sections will now describe the use of symbolic animation for the generation of test cases.

1.3 Automated Boundary Test Generation

We present in this section the use of the symbolic animation for automating the generation of model-based test cases. This technique aims at a structural coverage of the transitions of the system. To make it simple, each behavior of each operation of the B machine is targeted; the test cases thus aim at covering all the behaviors. In addition, a symbolic representation of the system states makes it possible to perform a boundary analysis from which the test targets will result [27, 2]. This technique is recognized as a pertinent heuristics for generating test data [5].

The tests that we propose are made of four parts, as illustrated in Figure 1.4. The first part, called *preamble*, is a sequence of operations that brings the system from the initial state to a state in which the test target, namely a state from which the considered behavior can be activated, is reached. The *body* is the activation of the behavior itself. Then, the *identification* phase is made of user-defined calls to observation operations, that are supposed to retrieve internal values of the system so that they can be compared to model data in order to establish the conformance verdict of the test. Finally, the *postamble* phase is similar to the preamble, but it brings the system back to the initial state or to another state that reaches another test target. The latter part is important to chain the test cases. It is especially used

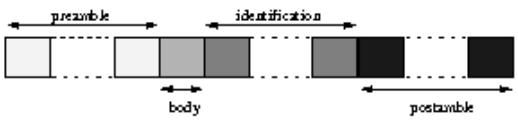

Figure 1.4: Composition of a Test Case

when testing embedded systems, since the execution of the tests on the system is very costly and such systems take usually much time to be reseted by hand.

This automated test generation technique requires some testability hypotheses to be employed. First, the operations of the B machine have to represent the control points of the system to be tested, so as to ease the concretization of the test cases. Second, it is mandatory that the concrete data of the SUT can be compared to the abstract data of the model, so as to be able to compare the results produced by the execution of the test cases with the results predicted by the model. Third, the SUT has to provide observation points that can be modeled in the B machine (either by return values of operations, such as the status words in the smart cards, or by observation operations).

We will now describe how the test cases can be automatically computed, namely how the test targets are extracted from the B machine, and how the test preambles, and postambles, are computed.

1.3.1 Extraction of the Test Targets

The goal of the tests is to verify that the behaviors described in the model exist in the SUT and produce the same result. To achieve that, each test will focus on one specific behavior of an operation. Test targets are defined as the states from which a given behavior can be activated. These test targets are computed so as to satisfy a structural coverage of the machine operations.

Definition 2 (Test Target) *Let* $OP = \langle (Act_1, Eff_1)[] \ldots [](Act_N, Eff_N)\rangle$ *be the set of behaviors extracted from operation* OP*, in which* Act_i *denotes the activation condition of behavior* i*,* Eff_i *denotes its effect, and* $[]$ *is an operator of choice between behaviors. Let* Inv *be the machine invariant. A test target is defined by a predicate that characterizes the states of the invariant from which a behavior* i *can be activated:* $Inv \wedge Act_i$*.*

The use of underlying constraint solving techniques makes it possible to provide interesting possibilities for data coverage criteria. In particular, we are able to perform a boundary

analysis of the behaviors of the model. Concretely, we will consider *boundary goals*, that are states of the model for which at least one of the state variable is at an extremum (minimum or maximum) of its current domain.

Definition 3 (Boundary Goal) *Let $minimize(V, C)$ (resp. $maximize(V, C)$) be a function that instantiates a symbolic variable V to its minimal value (resp. its maximal value), under the constraints given in C. Let Act_i be the activation condition of behavior i, let $\vec{P}$ be the parameters of the corresponding operation, and let $\vec{V}$ be the set of state variables that occur in behavior i, the boundary goals for the variables $\vec{V}$ are computed by:*

$$BG^{min} = minimize(f(\vec{V}), Inv \wedge \exists \vec{P}.Act_i)$$
$$BG^{max} = maximize(f(\vec{V}), Inv \wedge \exists \vec{P}.Act_i)$$

in which f is an optimization function that depends on the type of the variable:

if $\vec{X}$ is a set of integers, $f(\vec{X}) = \sum_{x \in \vec{X}} x$
if $\vec{X}$ is a set of sets, $f(\vec{X}) = \sum_{x \in \vec{X}} card(x)$
otherwise, $f(\vec{X}) = 1$

Example 4 (Boundary test targets) *Consider behavior [1,2,3,4,5,0] from operation VERIFY_PIN presented in Figure 1.3. The machine invariant gives the following typing informations:*

$$Inv \hat{=} tries \in 0..3 \wedge pin \in -1..9999 \wedge auth \in \{true, false\}$$

The boundary test targets are computed using the minimization/maximization formulas:

$$\begin{aligned} BG^{min} &= minimize(tries + pin, Inv \wedge \exists p \in 0..9999.(tries > 0 \wedge pin \neq -1 \wedge pin = p)) \\ &\rightsquigarrow tries = 1,\ pin = 0 \\ BG^{max} &= maximize(tries + pin, Inv \wedge \exists p \in 0..9999.(tries > 0 \wedge pin \neq -1 \wedge pin = p)) \\ &\rightsquigarrow tries = 3,\ pin = 9999 \end{aligned}$$

In order to improve the coverage of the operations, a predicate coverage criterion [30] can be applied by the validation engineer. This criterion acts as a rewriting of the disjunctions in the decisions of the B machine. Four rewritings are possible, that make it possible to

N	Rewriting of $P_1 \vee P_2$	Coverage criterion
1	$P_1 \vee P_2$	Decision Coverage (DC)
2	P_1 [] P_2	Condition/Decision Coverage (C/DC)
3	$P_1 \wedge \neg P_2$ [] $\neg P_1 \wedge P_2$	Full Predicate Coverage (FPC)
4	$P_1 \wedge P_2$ [] $P_1 \wedge \neg P_2$ [] $\neg P_1 \wedge P_2$	Multiple Condition Coverage (MCC)

Table 1.1: Decision coverage criteria depending on rewritings

satisfy different specification coverage criteria [30], as given in Table 1.1.

Rewriting 1 leaves the disjunction unmodified. Thus, the Decision Coverage criterion will be satisfied if a test target satisfies either P_1 or P_2 indifferently (also satisfying the Condition Coverage criterion (CC). Rewriting 2 produces two test targets, one considering the satisfaction of P_1, the other the satisfaction of P_2. Rewriting 3 will also produce two test targets, considering an exclusive satisfaction of P_1 without P_2 and vice-versa. Finally, Rewriting 4 produces three test targets that will cover all the possibilities to satisfy the disjunctions.

Notice that the consistency of the resulting test targets is checked so as to eliminate inconsistent test targets.

Example 5 (Decision coverage) *Consider behavior [1,2,9,0] from operation VERIFY_PIN presented in Figure 1.3. The selection of the Multiple Condition Coverage criterion will produce the following test targets:*

1. $Inv \wedge \exists p \in 0..9999 \;.\; (tries \leq 0 \wedge pin = -1)$
2. $Inv \wedge \exists p \in 0..9999 \;.\; (tries > 0 \wedge pin = -1)$
3. $Inv \wedge \exists p \in 0..9999 \;.\; (tries \leq 0 \wedge pin \neq -1)$

providing contexts from which boundary goals will then be computed.

We now describe how these targets are reached by symbolic animation by computation of the test preamble.

1.3.2 Computation of the Test Cases

Once the test targets and boundary goals are defined, the idea is to employ symbolic animation in an automated manner that will aim at reaching each target. To achieve that, a state exploration algorithm, variant of the A* path-finding algorithm and based on a Best-First exploration of the system states, has been developed.

This algorithm aims at finding automatically a path, from the initial state, that will reach a given set of states characterized by a predicate. The informal principle of the algorithm is given in Figure 1.5. From a given state, the symbolic successors, through each behavior, are computed using symbolic animation (procedure `compute_successors`). Each of these successors is then evaluated to compute the distance to the target. This latter is based on a heuristics that considers the "distance" between the current state and the targeted states (procedure `compute_distance`). To do that, the sum of the distances between each state variable is considered; if the domains of the two variables intersect, then the distance for these variables is 0, otherwise a customized formula, involving the type of the variable and the size of the domains, computes the distance (see [17] for more details). The computation of the sequence restarts from the most relevant state, i.e., the one presenting the smallest distance to the target (procedure `pop_minimal_distance` returning the most interesting triplet (state,sequence,distance) and removing it from the list of visited states). The algorithm starts with the initial state (denoted by s_init and obtained by initializing the variables according to the INITIALIZATION clause of the machine denoted by the `initialize` function). It ends if a zero-distance state is reached by the current sequence, or if all sequences have been explored for a given depth.

Since reachability of the test targets can not be decided, this algorithm is bounded in depth. Its worst case complexity is $O(n^d)$ where n is the number of behaviors in all the operations of the machine and d is the depth of the exploration (maximal length of test sequence). Nevertheless, the heuristics consisting in computing the distance between the states explored and the targeted states to select the most relevant states improves the practical results of the algorithm.

```
SeqOp ← compute_preamble(Depth, Target)
 begin
   s_init  ←  initialize ;
   Seq_curr ←  [init] ;
   dist_init ← compute_distance(Target,s_init) ;
   visited ←  [(s_init, Seq_curr, dist_init)] ;
    while visited ≠ []  do
      (s_curr, Seq_curr, MinDist) ← pop_mininal_distance(visited) ;
       if length(Seq_curr) < Depth  then
          [(s_1, Seq_1), ..., (s_N, Seq_N)] ← compute_successors((s_curr, Seq_curr)) ;
           for each (s_i, Seq_i) ∈ [(s_1, Seq_1), ..., (s_N, Seq_N)]  do
              dist_i ← compute_distance(Target,s_i) ;
               if dist_i = 0  then
                   return Seq_i;
               else
                  visited ← visited ∪ (s_i, Seq_i, dist_i) ;
               end if
           done
       end if
    done
    return [];
 end
```

Figure 1.5: State exploration algorithm

The computation of the preambule ends for three possible reasons. It may have found the target, and thus, the path is returned as a sequence of operations/behaviors. Notice that, in practice, this path is often the shortest from the initial state, but it is not always the case because of the heuristics used in during the search. The algorithm may also end by stating that the target has not been reached. This can be due to the fact that the exploration depth was too small, but it may also be due to the unreachability of the target.

Example 6 (Reachability of the test targets) *Consider the three targets given in Example 5. The last two can easily be reached. Target 2 can be reached by setting the value of the PIN, and Target 3 can be reached by setting the value of the PIN, followed by three successive authentication failures.*

Nevertheless, the first target will never be reached since the decrementation of the tries can only be done if pin ≠ -1. In order to avoid considering unreachable targets, the machine invariant has to be complete enough to catch at best the reachable states of the system, or, at least, to exclude unreachable states. In the example, completing the invariant by:

$pin = -1 \Rightarrow tries = 3$ makes Target 1 inconsistent, and thus, removes it from the test generation process.

The sequence returned by the algorithm represents the preamble, to which is concatenated the invocation of the considered behavior (representing the test body). If operation parameters are still constrained, they are also minimized or maximized, for their instantiation. The observation operations are specified by hand, and the (optional) postamble is computed on the same principle as the preamble.

1.3.3 Leirios Test Generator for B

This technique has been industrialized by the company Smartesting[2], a startup created from the research work done at the university of Franche-Comté in 2003, in a tool-set named Leirios[3] Test Generator for B machines [21] (LTG-B for short). This tool presents features of animation, test generation and publication of the tests. In a perspective of industrial use, the tool brings out the possibility of requirements traceability. Requirements can be tagged in the model by simple markers that will make it possible to relate them to the corresponding tests that have been generated (see [9] for more details). The tool also presents test generation reports that shows the coverage of the test targets and/or the coverage of the requirements, as illustrated in the screenshot shown in Figure 1.6.

1.3.4 Limitations of the automated approach

Despite this automated approach has been used successfully in various industrial case studies on embedded systems (as will be described in Section 1.5), the feedback from the field experience has shown some limitations.

The first issue is the problem of reachability of the test targets. Even if the set of system states is well-defined by the machine invariant, the experience shows that some test targets

[2] `www.smartesting.com`
[3] former name of the company Smartesting

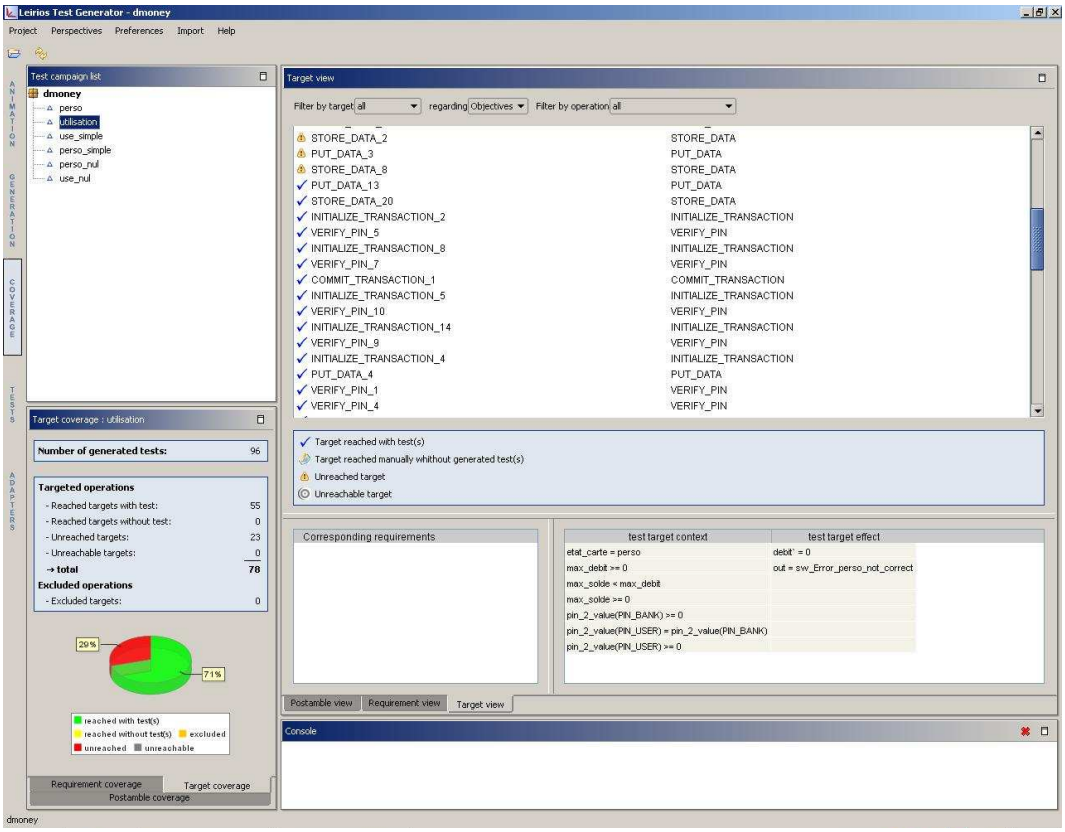

Figure 1.6: A screenshot of the LTG-B user interface

require an important exploration depth to be reached automatically, which may strongly increase the test generation time. Second, the lack of observations on the SUT may weaken the conformance relationship. As explained before, it is mandatory to dispose of a large number of observations points on the SUT to improve the accuracy of the conformance verdict. Nevertheless, if a limited number of observation is provided by the test bench (e.g. in smart cards only status words can be observed) it is mandatory to be able to check that the system has actually and correctly evolved. Finally, an important issue is the coverage of the dynamics of the system (e.g. ensure that a given sequence of commands can not be executed successfully if the sequence is broken). Knowing the test generation driving possibilites of the LTG-B tool, it is possible to encode the dynamics of the system by additional (ghost) variables on which a specific coverage criterion will be applied. This solution is not recommanded because it requires a good knowledge of how the tool works to be employed, which is not necessarily the case of any validation engineer. Again, if limited observation points are provided, this task is all the more complicated. This weakness is

amplified by the fact that the preambles are restricted to a single path from the initial state, and do not cover possibly interesting situations that would have required different sequences of operation to be computed (longer, involving loops, etc.).

These reasons led us to consider a complementary approach, also based on model animation, that would overcome the limitations described previously. This solution is based on user-defined scenarios, that will capture the know-how of the validation engineer and assist him in the design of his test campaigns.

1.4 Scenario-Based Test Generation

Scenario-Based Testing (SBT) is a concept according to which the validation engineer describes scenarios of use cases of the system, thus defining the test cases. In the context of software testing, it consists in describing sequences of actions, more or less formal, that exercise the functionalities of the system. We have chosen to express scenarios as regular expressions representing sequences of operations, possibly presenting intermediate states that have to be reached.

Such an approach is related to combinatorial testing, that use combinations of operations and parameter values, as done in the Tobias tool [25]. Nevertheless, combinatorial approaches can be seen as input-only, meaning that they do not produce the oracle of the test, and only provide a syntactical mean for generating tests, without checking the adequacy of the selected combinations w.r.t. a given specification. Thus, the various combinations of operations calls that can be produced may turn out to be not executable in practice. In order to improving this principle, we have proposed to rely on symbolic animation of formal models of the system in order to free the validation engineer from providing the parameters of the operations [18]. This makes it possible to only focus on the description of the successive operations, possibly punctuated with checkpoints, as intermediate states, that guide the steps of the scenario. The animation engine is then in charge of computing the feasibility of the sequence at unfolding-time and to instantiate the operation parameters values. One of the advantages of our SBT approach is that it helps the production of test cases by considering symbolic values for

the parameters of the operations. Thus, the user may force the animation to reach specific states, defined by predicates, that add constraints to the state variables' values. Another advantage is that it provides a direct requirement traceability of the tests, considering that each scenario addresses a specific requirement.

1.4.1 Scenario Description Language

We present here the language that we use for designing the scenarios, first introduced in [23]. Its core are regular expressions that are then unfolded and played by the symbolic animation engine. The language is structured in three layers: the *sequence* layer, the *model* layer, and the *directive* layer, that are now described.

Sequence and Model Layers. The *sequence* layer (Fig. 1.7) is based on regular expressions that make it possible to define test scenarios as operation sequences (repeated or alternated) that may possibly lead to specific states. The *model* layer (Fig. 1.8) describes the operation calls and the state predicates at the model level and constitutes the interface between the model and the scenario. A set of rules specifies the language.

Rule SEQ describes a sequence of operation calls as a regular expression. A step in the sequence is either a simple operation call, denoted by OP1, or a sequence of operation calls that leads to a state satisfying a state predicate, denoted by SEQ $\rightsquigarrow$(SP). This latter represents an improvement w.r.t. usual scenarios description languages, since it makes it possible

```
SEQ    ::=  OP1 | "(" SEQ ")"
          |  SEQ "." SEQ
          |  SEQ REPEAT ALL_or_ONE
          |  SEQ CHOICE SEQ
          |  SEQ "⇝(" SP ")"
REPEAT ::=  "?" | n | n..m
```

Figure 1.7: Syntax of the sequence layer

```
OP     ::=  operation_name
          |  "$OP"
          |  "$OP \ {" OPLIST "}"
OPLIST ::=  operation_name
          |  operation_name "," OPLIST
SP     ::=  state_predicate
```

Figure 1.8: Syntax of the model layer

```
CHOICE      ::=  "|"                    OP1   ::=  OP | "["OP"]"
               |  "⊗"                          |  "[" OP "/w" BHRLIST "]"
                                               |  "[" OP "/e" BHRLIST "]"
ALL_or_ONE  ::=  "_one"
               |  ϵ                     BHRLIST ::=  bhr_label (","  bhr_label)*
```

Figure 1.9: Syntax of the test generation directive layer

to define the target of an operation sequence, without necessarily having to enumerate all the operations that compose the sequence. Scenarios can be composed of the concatenation of two sequences, the repetition of a sequence, or the choice between two or more sequences. In practice, we use bounded repetition operators: 0 or 1, exactly n times, at most m times, between n and m times. Rule SP describes a state predicate, whereas OP is used to describe the operation calls that can be (*i*) an operation name, (*ii*) the $OP keyword, meaning "any operation", or (*iii*) $OP\{OPLIST} meaning "any operation except those of OPLIST".

Test Generation Directive Layer. This layer makes it possible to drive the step of test generation, when the tests are unfolded. We propose three kinds of directives that aim at reducing the search for the instantiation of a test scenario. This part of the language is given in Fig. 1.9.

Rule CHOICE introduces two operators denoted | and $\otimes$, for covering the branches of a choice. For example, if S_1 and S_2 are two sequences, $\mathsf{S_1} \mid \mathsf{S_2}$ specifies that the test generator has to produce tests that will cover S_1 and other tests that will cover schema S_2, whereas $\mathsf{S_1} \otimes \mathsf{S_2}$ specifies that the test generator has to produce test cases covering either S_1 or S_2. Rule ALL_or_ONE makes it possible to specify if all the solutions of the iteration will be returned (ϵ – by default) or if only one will be selected (_one).

Rule OP1 indicates to the test generator that it has to cover one of the behaviors of the OP operation (default option). The test engineer may also require all the behaviors to be covered by surrounding the operation with brackets. Two variants make it possible to select the behaviors that will be applied, by specifying which behaviors are authorized (/w) or refused (/e) using labels that have to tag the operations of the model.

Example 7 (An example of scenario) *Consider again the* `VERIFY_PIN` *operation from the previous example. A piece of scenario that expresses the invocation of this operation until the card is blocked, whatever the number of remaining tries might be, is expressed by* $(\texttt{VERIFY\_PIN}^{0..3}$ *_one)* $\rightsquigarrow$ *(*`tries=0`*).*

1.4.2 Unfolding and Instantiation of Scenarios

The scenarios are unfolded and animated on the model at the same time, in order to produce the test cases. To do that, each scenario is translated into a Prolog file, directly interpreted by the symbolic animation engine of BZ-Testing-Tools framework. Each solution provides an instantiated test case. The internal backtracking mechanism of Prolog is used to iterate on the different solutions. The instantiation mechanism involved in this part of the process aims at computing the values of the parameters of the operations composing the test case, so that the sequence is *feasible* [1, p. 290]. If a given scenario step can not be activated (e.g. due to an unsatisfiable activation condition) the subpart of the execution tree related to the subsequence steps of the sequence is pruned and will not be explored.

Example 8 (Unfolding and instantiation) *When unfolded, scenario* $(\texttt{VERIFY\_PIN}^{0..3}$ *_one)* $\rightsquigarrow$ *(*`tries=0`*) will produce the following sequences:*

(i) $\rightsquigarrow$ *(*`tries=0`*)*

(ii) `VERIFY_PIN`$(P_1) \rightsquigarrow$ *(*`tries=0`*)*

(iii) `VERIFY_PIN`(P_1) . `VERIFY_PIN`$(P_2) \rightsquigarrow$ *(*`tries=0`*)*

(iv) `VERIFY_PIN`(P_1) . `VERIFY_PIN`(P_2) . `VERIFY_PIN`$(P_3) \rightsquigarrow$ *(*`tries=0`*)*

where P_1, P_2, P_3 *are variables that will have to be instantiated afterwards. Suppose that the current system state gives* `tries=2` *(remaining tries) and* `pin=1234`. *Sequence* (i) *can not be satisfied,* (ii) *does not make it possible to block the card after a single authentication failure, sequence* (iii) *and* (iv) *are feasible, leading to a state in which the card is blocked. According to the selected directive (_one), only one sequence will be kept (here,* (iii) *since it represents the smallest number of iterations).*

The solver then instantiates parameters P_1 *and* P_2 *for sequence (iii). This sequence activates behavior* $[1, 2, 3, 5, 6, 8, 0]$ *of* `VERIFY_PIN` *followed by behavior* $[1, 2, 3, 5, 6, 9, 0]$ *that blocks the card (cf. Figure 1.3). The constraints associated to the variables representing the parameters are thus* $P_1 \neq 1234$ *and* $P_2 \neq 1234$*. A basic instantiation will then return* $P_1 = P_2 = 0$*, resulting in sequence:* `VERIFY_PIN(0); VERIFY_PIN(0).`

These principles have been implemented into a tool named *jSynoPSys* [18], a Scenario-Based Testing tool working on B Machines. A screenshot of the tool is displayed in Fig. 1.10. The tool makes it possible to design and play the scenarios. Resulting tests can be displayed in the interface or exported to be concretized. Notice that this latter makes it possible to reuse existing concretization layers that would have been developed for LTG-B.

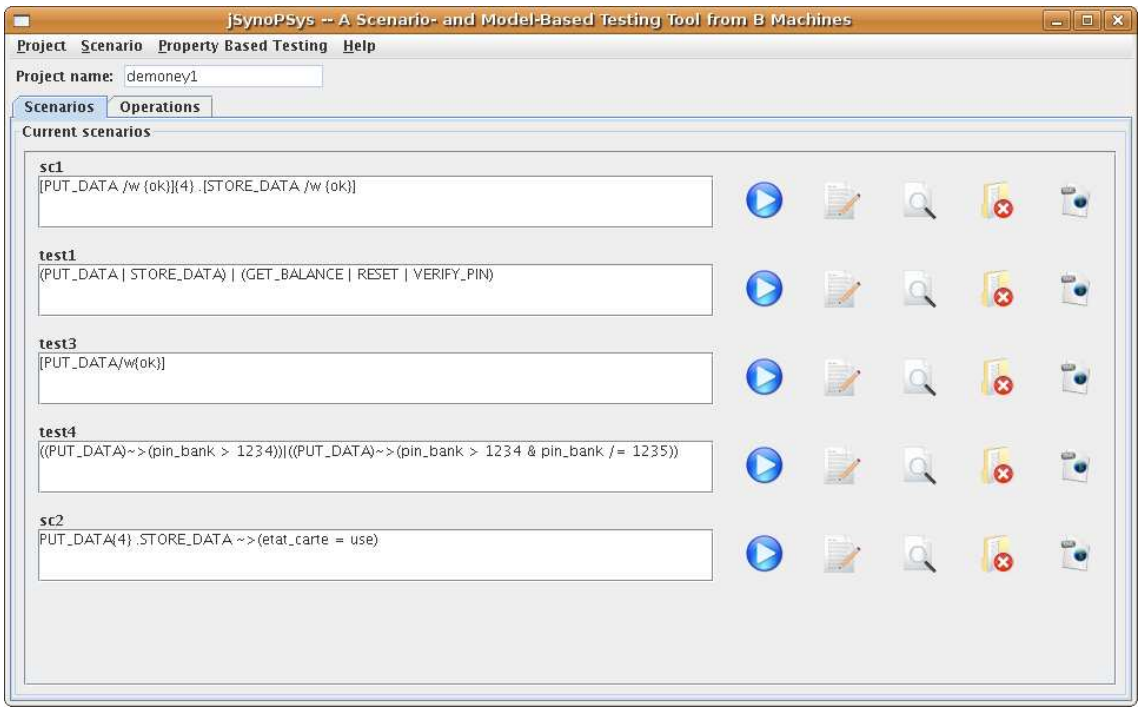

Figure 1.10: The jSynoPSys Scenario-Based Testing tool

1.5 Experimental Results

This section relates the experimental results obtained during various industrial collaboration in the domain of embedded systems: smart card applets [6] or operating systems [10], ticketing applications, automotive controllers [11], or space on-board software [15]. We first illustrate the relevance of the automated test generation approach compared to manual test design. Then, we show the complementary of the two test generation techniques presented in this chapter.

1.5.1 Automated vs. Manual Testing - the GSM 11.11 Case Study

In the context of an industrial partnership with the smart card division [4] of the Schlumberger company, a comparison has been done between a manual and an automated approach for the generation of test cases. The selected case study was the GSM 11.11 standard [20], that defines, on mobile phones, the interface between the Subscriber Identification Module (SIM) and the Mobile Equipment (ME).

The part of the standard that has been modeled consisted in the structure of the SIM, namely its organization in directories (called Dedicated Files – DF) or files (called Elementary Files – EF), and the security aspects of the SIM, namely the access control policies applied to the files. Files are accessible for reading, with 4 different access levels: ALWays (access can always be performed), CHV (access depends on a Card Holder Verification performed previously), ADM (for administration purposes) or NEVer (the file can not be directly access through the interface). The commands modeled were: SELECT_FILE (used to explore the file system), READ_BINARY (used to read in the files if permitted), VERIFY_CHV (used to authenticate the holder), UNBLOCK_CHV (used to unblock the CHV when too many unsuccessful authentication attempts with VERIFY_CHV happened). In addition, a command named STATUS makes it possible to retrieve the internal state of the card (current EF, current DF and current values of tries counters). Notice that no command was modeled to create/delete files or set access control permission: the file system structure and

[4] now Gemalto — `www.gemalto.com`

permission have been modeled as constants and manually created on the test bench.

The B model was about 500 lines of code. 42 boundary goals have been computed, leading to the automated computation of 1008 test cases. These tests have been compared to the existing test suite, that had been hand-written by the validation team, and covering the same subset of the GSM 11.11 standard.

This comparison showed the automated test suite included 80% of the manual tests. More precisely, since automated test cases cover behaviors atomically, a single manual test may usually exercise the SUT in the same way that several automated tests would do. On the opposite, 50% of the automated tests were not absent from the manual test suite. Among the 20% of tests that were not produced automatically, three reasons appear. Some of the missing tests (5%) consider boundary goals that have not been generated. Other tests (35%) consider the activation of several operations from the boundary state, that is not considered by the automated approach. Whereas these two issues are not crucial, and do not put the process into question, it appeared that the rest of the tests (60%) cover parts of the informal requirements that were not expressed in the B model. To overcome this limitation, a first attempt of scenario-based testing has been proposed, asking the validation engineer to provide tests designed independently, with the help of the animation tool.

The study also compared the efforts for designing the test cases. As shown in Table 1.2 the automated process reduces test implementation time, but adds time for the design of the B model. On the example, the overall effort is reduced by 30%.

Manual design		Automated Process	
Design of the test plan	6 m/d	Modelling in B	12 m/d
		Test generation	automated
Implementation and test execution	24 m/d	Test execution	6 m/d
Total	30 m/d	Total	18 m/d

Table 1.2: Comparison in terms of time spent on the testing phase in men/day

1.5.2 Completing functional tests with scenarios – the IAS Case Study

The scenario-based testing process has been designed during the French National project POSE[5] that involved the leader of smart cards manufacturers, Gemalto, and aimed at the validation of security policies for the IAS platform.

IAS stands for Identification, Authentication and electronic Signature. It is a standard for Smart Cards developed as a common platform for e-Administration in France, and specified by GIXEL. IAS provides identification, authentication and signature services to the other applications running on the card. Smart cards such as the french identity card, or the "Sesame Vitale 2" health card are expected to conform to IAS. Being based on the GSM 11.11 interface, the models present similarities. This platform presents a file system containing DFs and EFs. In addition, DFs host Security Data Objects (SDO) that are objects of an application containing highly sensitive data such as PIN codes or cryptographic keys. The access to an object by an operation in IAS is protected by security rules based on the security attributes of the object. The access rules can possibly be expressed as a conjunction of elementary access conditions, such as Never (which is the rule by default, stating that the command can never access the object), Always (the command can always access the object), or User (user authentication: the user must be authenticated by means of a PIN code). The application of a given command to an object can then depend on the state of some others SDOs, which complicates the access control rules.

The B model for IAS is 15500 lines long. The complete IAS commands have been modelled as a set of 60 B operations manipulating 150 state variables. A first automated test generation campaign had been experimented, producing about 7000 tests. A close examination of the tests concluded to the same weakness as for the GSM 11.11 case study, namely, interesting security properties were not covered at best, and manual testing would be necessary to overcome this weakness.

The idea of the experiment was to relate to the Common Criteria (C.C.) norm [14], a standard for the security of Information Technology products that provides a set of assurances w.r.t. the evaluation of the security implemented by the product. When a product is

[5] http://www.rntl-pose.info

delivered, it can be evaluated w.r.t. the C.C. that ensure the conformance of the product w.r.t. security guidelines related to the software design, verification and validation of the standard. In order to pass the current threshold of acceptance, the C.C. require the use of a formal model and evidences of the validation of the given security properties of the system. Nevertheless, tracing the properties in the model in order to identify dedicated tests was not possible, since some of the properties were not directly expressed in the original B model.

For the experimentation, we started by designing a simplified model, focusing on the access control features, and called Security Policy Model (SPM). This model is 1100 lines long with 12 operations manipulating 20 state variables, and represents the files management with authentications on their associated SDOs.

In order to complete the tests generated automatically from the complete model, three scenarios have been designed for exercising specific security properties that could not be covered previously. The scenarios and their associated tests provide direct evidences of the validation of given properties. Each scenario is associated to a *test needs* that informally expresses the intention of the scenario w.r.t. the property, and provides documentation on the test campaign.

- The first scenario exercises a security property stating that the access to an object protected by a PIN code requires to gain an authentication over the PIN code. The tests produced automatically exercise this property in a case where the authentication is gained, and in a case where it is not. The scenario completes these tests by considering the case in which the authentication has first been obtained, but lost afterwards. The unfolding of this scenario provided 35 instantiated sequences, illustrating the possible ways of losing an authentication.
- The second scenario exercises the case of homonym PIN files located in differents DFs, and their involvment in the access control conditions. In particular, it aimed at ensuring that an authenticated PIN in a specific DF is not mistakenly considered in an access control condition that involves another PIN displaying the same name but located in another DF. The unfolding of this scenario resulted in 66 tests.
- The third and last scenario exercises a property specifying that the authentication

gained over a PIN not only depends on the location of the PIN, but also on the life cycle state of the DF where a command protected by the PIN is applied. This scenario aimed at testing situations where the life cycle state of the directory is not always activated (which was not covered by the first campaign). The unfolding of this scenario produced 82 tests.

In the end, the three scenarios produced 183 tests that were run on the SUT. Even if this approach did not reveal any errors, the execution of these tests help increasing the confidence in the system w.r.t. the considered security properties. In addition, the scenarios could provide direct evidence of the validation of these properties, which were useful for the Common Criteria evaluation of the IAS.

Notice that, when replaying the scenarios on the complete IAS model, the scenario-based testing approach detected a non-conformance between the SPM and the complete model, due to a different interpretation of the informal requirements in the two models.

1.5.3 Complementarity of the Two Approaches

These two case studies illustrate the complementarity of the approaches. The automated boundary test generation approach is efficient at replacing most of the manual design of the functional tests, saving efforts in the design of the test campaigns. Nevertheless, it is mandatory to complete the test suite at least to exercise properties related to the dynamics of the system to be tested. To this end, the scenario-based testing approach provides an interesting way to assist the validation engineer in the design of complementary tests. In both cases, the use of symbolic techniques ensure the scalability of the approach.

1.6 Related Work

This section is divided in two. The first subsection is dedicated to automated test generation using model coverage criteria. The second compares with other scenario based testing

approaches.

1.6.1 Model-Based Testing Approaches using Coverage Criteria

Many model-based testing approaches rely on the use of a Labeled Transition System or a Finite State Machine from which the tests are generated using dedicated graph exploration algorithms [26]. Tools like TorX [37] or TGV [22] use a formal representation of the system written as Input-Output Labeled Transition Systems (IOLTS), on which test purposes are applied to select the relevant test cases to be produced. In addition, TorX proposes the use of test heuristics that help filtering the resulting tests according to various criteria (test length, cycle coverage, etc.). The conformance is established using the ioco [38] relationship. The major differences with our automated approach is that, first, we do not know the topology of the system. Second, these processes are based on the online (or on-the-fly) testing paradigm in which the model program and the implementation are considered altogether. On the contrary, our approach is assimilated to offline testing, that requires a concretization step for the tests to be run on the SUT and the conformance to be established.

The STG tool [16] improves the TGV approach by considering Input-Output Symbolic Transitions Systems, on which deductive reasoning applies, involving constraint solvers or theorem provers. Nevertheless, the kind of data manipulated are often restricted to integers and booleans, whereas our approach manipulates additional data types, assimilated to collections (sets, relation, functions, etc.) that may be useful for the modeling step. Similarly, AGATHA [7] is a test generation tool based on constraint solving techniques that works by building a symbolic execution graph of systems modelled by communicating automata. Tests are then generated using dedicated algorithms in charge of covering all the transitions of the symbolic execution graph.

The CASTING [40] testing method is also based on the use of operations written in DNF for extracting the test cases [19]. In addition, CASTING considers decomposition rules that have to be selected by the validation engineer so as to refine the test targets. CASTING has been declined for B machines. Test targets are computed as constraints applying on

the before and after states of the system. These contraints define states that have to be reached by the test generation process. To achieve that, the concrete state graph is built and explored. Our approach improves this technique by considering symbolic techniques, that make it possible to perform a boundary analysis for the test data, potentially improving the test targets. Moreover, the on-the-fly exploration of the states graph avoids the complete enumeration of all the states of the model.

Also based on B specifications, ProTest [33] is an automated test generator coupled with the ProB model-checker [28]. ProTest works by first, building the concrete system state graph by model animation, that is then explored for covering states and transitions, using classical algorithms. One point in favor of ProTest/ProB is that it covers a large subset of the B notation as our approach, notably dealing with sequences. Nevertheless, the major drawback is the exhaustive exploration of all the concrete states, that complicates the industrial use of the tool on large models. In particular, the IAS model used in the experiment reported in Section 1.5.2 can not be handled by the tool.

1.6.2 Scenario Based Testing approaches

In the litterature, a lot of scenario based testing works focus on extracting scenarios from UML diagrams, such as the SCENTOR approach [41] or SCENT [32] using statecharts. The SOOFT approach [39] proposes an object oriented framework for performing scenario-based testing. In [8], Binder proposes the notion of round-trip scenario test that cover all event-response path of an UML sequence diagram. Nevertheless, the scenarios have to be completely described, contrary to our approach that abstracts the difficult task of finding well-suited parameter values.

In [4], the authors propose an approach for the automated scenario generation from environment models for testing of real-time reactive systems. The behavior of the system is defined as a set of events. The process relies on an attributed event grammar (AEG) that specifies possible event traces. Even if the targeted applications are different, the AEG can be seen as a generalization of regular expressions that we consider.

Indirectly, the test purposes of the STG [16] tool, described as IOSTS (Input/Output Symbolic Transition Systems) can be seen as scenarios. Indeed, the test purposes are combined with an IOSTS of the system under test, by an automata product, that restricts the possible executions of the system to those illustrating the test purpose. Such an approach has also been adapted to the B machines, in [24].

A similar approach is the test by model-checking, where test purposes can be expressed in the shape of temporal logic properties, as is the case in [3] or [35]. The model checker computes witness traces of the properties by a synchronized product of the automata of the property and of a state/transition model of the sytem under test. These traces are then used as test cases. A input/output temporal logic has also been described in [31] to express temporal properties w.r.t. IOSTS. The authors use an extension of the AGATHA tool to process such properties.

As explained in the article, we were inspired by the TOBIAS tool [25] that works with scenarios expressed using regular expressions representing the combinations of operations and parameters. Our approach improves this principle by avoiding to enumerate the combinations of input parameters. In addition, our tool provides test driving possibilities that may be used to easily master the combinatorial explosion, inherent to such an approach. Nevertheless, the TOBIAS input language is more expressive than ours and a combination of these two approaches, that would employ the TOBIAS tool for describing the test cases, is currently under study. Notice that an experiment has been done in [29] for coupling TOBIAS with UCASTING, the UML version of the CASTING tool [40]. This work made it possible to use UCASTING for (i) filtering the large tests sequences combinatorially produced by TOBIAS, by removing traces that were not feasible on the model, or (ii) to instantiate operation parameters. Even if the outcome is similar, our approach differs since the inconsistency of the test cases is detected without having to completely unfold the test sequences. Moreover, the coupling of these tools did not include as much test driving options as we propose.

The technique for specifying scenarios can be related to Microsoft Parameterized Unit Tests (PUT for short) [36], in which the user writes skeletons of test cases involving parameterized data, that will be instantiated automatically using constraint solving techniques.

Moreover, the test cases may contain basic structures such as conditions and iterations, that will be unfolded during the process, so as to produce test cases. Our approach is very similar in its essence, but some differences exist. First, our scenarios do not contain data parameters. Second, we express them on the model, whereas the PUT approach aims at producing test cases that will be directly executed on the code, leaving the question of the oracle not addressed. Nevertheless, the question of refining the scenario description language so as to propagate some symbolic parameterized data along the scenario is under study.

1.7 Conclusion and Open Issues

This chapter has presented two test generation techniques using the symbolic animation of formal models, written in B, used for automating test design for in the context of embedded systems such as smart cards. The first technique relies on the computation of boundary goals that define tests targets. These are then automatically reached by a customized state exploration algorithm. This technique has been commercialized by the Smartesting company, and applied on various case studies in the domain of embedded systems, in particular in the domain of electronic transactions. The second technique uses user-defined scenarios, expressed as regular expressions on the operations of the model and intermediate states, that are unfolded and animated on the model so as to filter the inconsistent test cases. This technique has been designed and experimented during an industrial partnership. This scenario based testing approach has shown to be very convenient, firstly with the use of a dedicated scenario description language that is easy to put into practice. Moreover, the connection between the tests, the scenarios and the properties from which they originate can be directly established, providing a mean for ensuring the traceability of the tests, which is useful in the context of high-level evaluation of Common Criteria, that requires evidences of the validation of specific properties of the considered software.

The work presented here has been applied to B models, but it is not restricted to this formalism, and the adaptation to UML, in partnership with Smartesting, is currently being

studied.

Even if the scenario based testing technique overcomes the limitations of the automated approach, in terms of relevance of the preambles, reachability of the test targets, and observations, the design of the scenario is still a manual step that requires the validation engineer to intervene. One interesting lead would be to automate the generation of the scenarios, in particular using high-level formal properties that they would exercise. Another approach is to use model abstraction for generating the tests cases, based on dynamic test selection criteria, expressed by the scenarios.

Finally, we have noticed that the data coverage for the operation parameters is relatively limited. We are thus investingating to complete the scenario description language by re-introducing the operation parameters and giving the possibility to define symbolic values inside the scenario, that would propagated throughout the whole test case, unlike in the Parametrized Unit Tests [36].

References

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

[2] F. Ambert, F. Bouquet, B. Legeard, and F. Peureux. Automated boundary-value test generation from specifications - method and tools. In *4th Int. Conf. on Software Testing, ICSTEST 2003*, pages 52–68, Cologne, Allemagne, April 2003.

[3] P. Amman, W. Ding, and D. Xu. Using a model checker to test safety properties. In *ICECCS'01, 7-th Int. Conf. on Engineering of Complex Computer Systems*, page 212, Washington, DC, USA, 2001. IEEE Computer Society.

[4] M. Auguston, J.B. Michael, and M.-T. Shing. Environment behavior models for scenario generation and testing automation. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, pages 1–6, New York, NY, USA, 2005. ACM.

[5] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.

[6] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *International Journal of Software Practice and Experience*, 34(10):915–948, 2004.

[7] C. Bigot, A. Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin. Automatic test generation with agatha. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003*, volume 2619 of *Lecture Notes in Computer Science*, pages 591–596. Springer, 2003.

[8] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting. Requirement traceability in automated test generation - Application to smart card software validation. In *Procs. of the ICSE Int. Workshop on Advances in Model-Based Software Testing (A-MOST'05)*, St. Louis, USA, May 2005. ACM Press.

[10] F. Bouquet, J. Julliand, B. Legeard, and F. Peureux. Automatic reconstruction and generation of functional test patterns - Application to the Java Card Transaction Mechanism (confidential). Technical Report TR-01/02, LIFC - University of Franche-Comté and Schlumberger Montrouge Product Center, 2002.

[11] F. Bouquet, F. Lebeau, and B. Legeard. Test case and test driver generation for automotive embedded systems. In *5th Int. Conf. on Software Testing, ICS-Test 2004*, pages 37–53, Düsseldorf, Germany, April 2004.

[12] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B: A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2):143–157, August 2004.

[13] F. Bouquet, B. Legeard, M. Utting, and N. Vacelet. Faster analysis of formal specifications. In J. Davies, W. Schulte, and M. Barnett, editors, *6th Int. Conf. on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 239–258, Seattle, WA, USA, November 2004. Springer-Verlag.

[14] Common Criteria for Information Technology Security Evaluation, version 3.1. Technical Report CCMB-2006-09-001, sept 2006.

[15] Ph. Chevalley, B. Legeard, and J. Orsat. Automated Test Case Generation for Space

On-Board Software. In Eurospace, editor, *DASIA 2005, Data Systems In Aerospace Int. Conf.*, pages 153–159, Edinburgh, UK, May 2005.

[16] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: a tool for generating symbolic test programs and oracles from operational specifications. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 301–302, New York, NY, USA, 2001. ACM.

[17] S. Colin, B. Legeard, and F. Peureux. Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3):213–235, 2004.

[18] F. Dadeau and R. Tissot. jSynoPSys – a scenario-based testing tool based on the symbolic animation of B machines. In *MBT'09, 5rd Int. Workshop on Model-Based Testing, co-located with ETAPS'2009)*, York, United Kingdom, March 2009. To appear in ENTCS.

[19] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock and P. Gorm Larsen, editors, *FME '93: First International Symposium of Formal Methods Europe*, volume 670 of *LNCS*, pages 268–284, Odense, Denmark, April 1993. Springer.

[20] European Telecommunications Standards Institute, F-06921 Sophia Antipolis cedex - France. *GSM 11-11 V7.2.0 Technical Specifications*, 1999.

[21] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated test generation from B models. In *B'2007, the 7th Int. B Conference - Industrial Tool Session*, volume 4355 of *LNCS*, pages 277–280, Besancon, France, January 2007. Springer.

[22] C. Jard and T. Jron. Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, October 2004.

[23] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *AST'08, 3rd Int. workshop on Automation of Software Test*, pages 41–44, Leipzig, Germany, May 2008. ACM Press.

[24] J. Julliand, P.-A. Masson, and R. Tissot. Generating tests from B specifications and test purposes. In *ABZ'2008, Int. Conf. on ASM, B and Z*, volume 5238 of *LNCS*, pages

139–152, London, UK, September 2008. Springer.

[25] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS Combinatorial Test Suites. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering, 7th Int. Conf., FASE 2004*, volume 2984 of *LNCS*, pages 281–294, Barcelona, Spain, 2004. Springer.

[26] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, pages 1090–1123, 1996.

[27] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proc. of the Int. Conf. on Formal Methods Europe, FME'02*, volume 2391 of *LNCS*, pages 21–40, Copenhaguen, Denmark, July 2002. Springer.

[28] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[29] O. Maury, Y. Ledru, and L. du Bousquet. Intgration de TOBIAS et UCASTING pour la gnration de tests. In *16th International Conference Software and Systems and their applications-ICSSEA*, Paris, 2003.

[30] A.J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las Vegas, USA, October 1999. IEEE Computer Society Press.

[31] N. Rapin. Symbolic execution based model checking of open systems with unbounded variables. In *TAP'09, Tests and Proofs*, pages 137–152, 2009.

[32] J. Ryser and M. Glinz. A practical approach to validating and testing software systems using scenarios, 1999.

[33] M. Satpathy, M. Leuschel, and M. Butler. ProTest: An Automatic Test Environment for B Specifications. *Electronic Notes in Theroretical Computer Science*, 111:113–136, January 2005.

[34] Swedish Institute of Computer Sciences. *SICStus Prolog 3.11.2 manual documents*, June 2004. `http://www.sics.se/sicstus.html`.

[35] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IRI'2004, IEEE Int. Conf. on Information Reuse and Integration*, pages 413–498,

November 2004.

[36] N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005.

[37] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[38] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[39] W. T. Tsai, A. Saimi, L. Yu, and R. Paul. Scenario-based object-oriented testing framework. *qsic*, 00:410, 2003.

[40] L. van Aertryck, M. Benveniste, and D. Le Mtayer. Casting: A formally based software test generation method. *Formal Engineering Methods, International Conference on*, 0:101, 1997.

[41] J. Wittevrongel and F. Maurer. Scentor: Scenario-based testing of e-business applications. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 41–48, Washington, DC, USA, 2001. IEEE Computer Society.