Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 253 (2009) 117–132

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

jSynoPSys – A Scenario-Based Testing Tool based on the Symbolic Animation of B Machines

Frédéric Dadeau[1] Régis Tissot[2]

LIFC - INRIA CASSIS Project
16 route de Gray
25030 Besançon, FRANCE

Abstract

This paper presents the jSynoPSys tool that implements the concept of Scenario-Based Testing from B machines. This consists in describing execution scenarios, expressed as regular expressions over the operations of the system, coupled with intermediate system states that have to be reached when the scenario is unfolded. The tool relies on the BZ-Testing-Tools engine, originally designed to perform symbolic animation and boundary test generation from B machines. The main advantage of our Scenario-Based Testing technique is that it performs a full abstraction of the operation parameter values, that are instantiated at the end, when the scenario has been completely unfolded, using constraint solving techniques.

Keywords: Test generation, symbolic animation, scenario-based testing, B machines, constraint solving

1 Context and Motivations

Model-Based Testing, or black-box testing [5], consists in using a formal model, describing the behavior of the system, for both computing the test cases and computing the oracle. This latter that makes it possible to establish the conformance verdict that indicates whether the system conforms, or not, to the considered model. The computation of the test cases can be performed according to different techniques that aim at covering the model, depending on its kind. For example, a model formalized as a finite state machine (FSM) is traditionally covered by state exploration or transition coverage algorithms (such as the Chinese Postman algorithm) [17]. In the case of behavioral specifications of the system, where the operations are described using generalized substitutions, as in B [1], or through pre/postconditions,

[1] Email: frederic.dadeau@lifc.univ-fcomte.fr
[2] Email: regis.tissot@lifc.univ-fcomte.fr

doi:10.1016/j.entcs.2009.09.055

as in UML/OCL [20], techniques aiming at the structural coverage of the model can be employed. It is the case of the Leirios Test Generator [12] tool (LTG) from the Smartesting company[3] that computes test sequences based on a structural coverage of the operations of a B abstract machine.

In the latter case, the test targets consist in activating the operations of the model, accordingly to their preconditions. These targets thus depend on the information contained in the operations, and are restricted to describing sets of states (namely, for LTG, the states satisfying the activation conditions of the considered operations). The main problem comes from the test case computation algorithm of LTG, that systematically builds the shortest operation sequence that leads to the test target. As a consequence, the relevance of the test cases can be decreased, and may lead to avoid possibly interesting cases, related to the experience of the validation engineer, or to a specific property of the system, that one may want to exercise. To overcome this problem, solutions based on the use of scenarios are considered. An example is the combinatorial testing tool TOBIAS [16] which unfolds regular expressions describing combinations of operations with their parameters. Inspired from this work, we have proposed in [10,13] a similar technique based on the symbolic animation of a formal model, written as a B abstract machine. This technique relies on a scenario description language that makes it possible to express the scenarios that have to be played on the model. Contrary to TOBIAS, the enumeration of the possible parameter values can be avoided, since they can be computed automatically based on the operations preconditions.

We propose to use a constraint solver that is in charge of performing the animation of the model, and is thus able to perform a thin enumeration of resulting test cases, using a native backtracking mechanism. This principle is based on the actual constraint solver of BZ-Testing-Tools (BZ-TT) [2], an animation and boundary test generation framework, for B machines, from which LTG originates. These principles have been successfully experimented on several realistic case studies, such as a model of a POSIX-compliant file system [10]. This technique is implemented within a tool, named *jSynoPSys*, a graphical user interface written in Java, coupled with a dedicated Prolog constraints solver. This paper presents the principles and the main features of the tool.

This paper is organized as follows. Section 2 presents a brief introduction to the B abstract machines, the formalism used to model the systems we consider. This section also introduces a running example that will be used in the remainder of the paper to illustrate the discourse. Then, Section 3 presents how B models can be animated using constraint solving techniques. The notion of Scenario-Based Testing, along with the description of the scenario language is given in Sect. 4. Section 5 presents the jSynoPSys tool and its features. Finally, Section 6 presents some related works, before concluding with the forthcoming works in Sect. 7

[3] formely Leirios Technologies

2 B Abstract Machines

The B method [1] is dedicated to the formal development, from high level specification to implementable code. Specifications are based on three formalisms: data are specified using a set theory, properties are first-order predicates and the behavioral part is specified by *Generalized Substitutions*.

2.1 The B Method

The B method starts by the writing of a formal specification, named *abstract machine*, that gives a functional view of the system. The machine is then spiced up with invariant properties that represent properties that have to hold at each state of the system execution. It means that (*i*) the initialization has to establish the invariant, (*ii*) the operations have to preserve the invariant (meaning that if the invariant is satisfied before the operation, then it also has to be satisfied after the execution of the operation). Operations are written in terms of Generalized Substitutions that are built on basic assignments, composed into more generalized and expressive structures, that may for example, represent conditional substitutions (IF...THEN...ELSE...END) or non-deterministic buildings (CHOICE, ANY).

The B method has been applied in many industrial applications, particularly in the railway domain (e.g. line 14 of the Paris called METEOR [4]) and in the context of Java-card application or environment [15].

In this work, we do not take refinements into account, we only focus on abstract machines. Notice that this is not a restriction, since refinements of machines can be flattened into a single B machine.

2.2 A Running Example

The example that we propose is a case study named Demoney [19]. It is a smart card applet designed for research purposes by Trusted Logics. Even if no "real" implementation of Demoney has ever been embedded on a smart card, it represents a realistic case study. The specification of Demoney describes an electronic purse with two PIN codes, one for the card holder and one for the bank. As for every purse, it is possible to credit the card, or to debit it in order to pay a purchase.

Similarily to every smart card application, Demoney presents a notion of *life cycle* that starts with a *personalization* phase, during which the different parameters of the card, namely the maximal balance, the maximal debit and the PIN code values, are set using the `PUT_DATA` command. Then, the personalization is validated by invoking the `STORE_DATA` command, that brings the card to the *use* phase. During this phase, the user may start a credit or debit transaction using the `INIT_TRANSACTION` command and validate this transaction, using `COMMIT_TRANSACTION`. If the user wants to credit the purse, he first has to authenticate with his PIN code, using the `VERIFY_PIN` command. When the user fails to authenticate three times, the card is then in the *blocked* state. To unblock it, the bank has to authentify, also using the `VERIFY_PIN` command and then change the user PIN code, using the

```
out ← PUT_DATA(p, data) ≙
   PRE
      p ∈ -128..127 ∧ data ∈ -32768..32767
   THEN
      IF (card_status = perso) THEN
         IF p = SET_MAX_BALANCE ∧ data ≥ 0 THEN
            max_balance := data || out := sw_Success
         ELSE
            IF p = SET_MAX_DEBIT ∧ data ≥ 0 THEN
               max_debit := data || out := sw_Success
            ELSE
               ... /* remainder of the operation */
            END
         END
      ELSE
         out := sw_Error_life_cycle
      END
   END
```

Figure 1. An excerpt of the PUT_DATA operation

PIN_CHANGE_UNBLOCK command. Similarily, if the bank fails to authenticate four times, the card is definitely *dead*, ie. no other command can be successfully executed.

We have developed a B model of the Demoney applet, that is composed of 500 lines of B code, spread in 6 operations. As for every smart card command, the operations can always be invoked (their precondition is true) and they are written in a defensive style, returning a status word that indicates if the command succeeded or not. As an illustration of the B notation, Figure 1 gives a subset of the PUT_DATA operation. This operation is used to personalize the card. Its first parameter p is a byte that gives the kind of personalization performed among values SET_MAX_BALANCE, SET_MAX_DEBIT, SET_HOLDER_PIN or SET_BANK_PIN supposed to be integer constants whose meanings are obvious. The second parameter is used to specify the value assigned to the considered element to be personalized (in the figure, the maximal balance or the maximal debit of the purse). Different kinds of errors can be returned if the invocation fails, e.g. sw_Error_life_cycle is returned when the command is not invoked at the personalization phase. Status word sw_Success indicates a successful invocation and termination of the command.

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, different resulting states can be obtained. We now describe the principle of symbolic animation.

3 Symbolic Animation of B Models

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 now described.

3.1 Definition of the Behaviors

The Prolog animation engine of BZ-Testing-Tools [8] that we use relies on a decomposition of the B machines operations into *behaviors*. Each behavior is defined as a predicate, representing its activation condition, and a substitution that indicates the evolution of the state variables and the instantiation of the return parameters of the operation. These behaviors are computed as the paths in the control flow graph of the considered B operation. They consist in two parts: an activation condition and a substitution.

Example 3.1 *(Computation of behaviors)* Consider the excerpt of the PUT_DATA operation given in Fig. 1. It presents at least 4 behaviors, that are the following (for each behavior, the $\Longrightarrow$ symbol is used to separate the activation condition and the substitution):

b_1 : $\texttt{p} \in \text{-128..127} \wedge \texttt{data} \in \text{-32768..32767} \wedge \texttt{card\_status} = \texttt{perso} \wedge \texttt{p} = \texttt{SET\_MAX\_BALANCE} \wedge \texttt{data} \geq 0 \Longrightarrow \texttt{max\_balance := data} \parallel \texttt{out := sw\_Success}$

b_2 : $\texttt{p} \in \text{-128..127} \wedge \texttt{data} \in \text{-32768..32767} \wedge \texttt{card\_status} = \texttt{perso} \wedge (\texttt{p} \neq \texttt{SET\_MAX\_BALANCE} \vee \texttt{data} < 0) \wedge \texttt{p} = \texttt{SET\_MAX\_DEBIT} \wedge \texttt{data} \geq 0 \Longrightarrow \texttt{max\_debit := data} \parallel \texttt{out := sw\_Success}$

b_3 : $\texttt{p} \in \text{-128..127} \wedge \texttt{data} \in \text{-32768..32767} \wedge \texttt{card\_status} = \texttt{perso} \wedge (\texttt{p} \neq \texttt{SET\_MAX\_BALANCE} \vee \texttt{data} < 0) \wedge (\texttt{p} \neq \texttt{SET\_MAX\_DEBIT} \vee \texttt{data} < 0) \wedge \ldots \Longrightarrow \ldots$

b_4 : $\texttt{p} \in \text{-128..127} \wedge \texttt{data} \in \text{-32768..32767} \wedge \texttt{card\_status} \neq \texttt{perso} \Longrightarrow \text{out} := \texttt{sw\_Error\_life\_cycle}$

The decomposition of operations into behaviors gives equivalence classes in terms of resulting states after the execution of the operation.

3.2 Use of the Behaviors for the Symbolic Animation

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

All state variables that are related to the abstracted parameter (e.g. assigned in a substitution), also become symbolic variables, linked to the corresponding parameter. A system state that contains at least one symbolic state variable is said to be a *symbolic state*.

Example 3.2 *(Activation of a single behavior)* Consider behavior b_1 from the previous example. Suppose it is invoked from the initial state of the system, namely:

$$init \leadsto \texttt{max\_balance} = -1, \texttt{max\_debit} = -1, \texttt{card\_status} = \texttt{perso}, ...$$

The subsequent activation of b_1 results in the following symbolic state:

$$init; PUT\_DATA(X_1, X_2) \leadsto \texttt{max\_balance} = X_3, \texttt{max\_debit} = -1, ...$$

with the following set of constraints:

$$X_1 = \texttt{SET\_MAX\_BALANCE},\ X_2 \in 0..32767,\ X_2 = X_3$$

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 [8], a set-theoretical constraint solver written in SICStus Prolog that is able to handle a large subset of the data structures existing in the B machines (sets, relations, functions, integers, atoms, etc.).

Example 3.3 *(Combination of behaviors)* Suppose we restart the animation from the symbolic state reached at the end of the previous example. Activating behavior b_2 will cause the following state to be reached.

$$init; PUT\_DATA(X_1, X_2); PUT\_DATA(X_4, X_5) \leadsto \texttt{max\_balance} = X_3, \texttt{max\_debit} = X_6, ...$$

with the following set of constraints:

$$\begin{aligned} &X_1 = \texttt{SET\_MAX\_BAL.},\ X_2 \in 0..32767,\ X_2 = X_3, \\ &\quad X_4 = \texttt{SET\_MAX\_DEBIT},\ X_5 \in 0..32767,\ X_5 = X_6 \end{aligned}$$

Once the sequence has been symbolically played, the remaining symbolic parameters can be instantiated by a simple labeling procedure, that consists in solving the constraints system and produce an instantiation of the symbolic variables. This makes it possible to produce an instantiated test case.

Example 3.4 *(From an abstract operation sequences to a test case)* Suppose we restart the animation from the symbolic state reached at the end of the previous example. The first possible instantiation of the symbolic variables that satisfies the constraints is the following:

$$X_1 = \texttt{SET\_MAX\_BALANCE},\ X_2 = 0,\ X_3 = X_0,\ X_4 = \texttt{SET\_MAX\_DEBIT},\ X_5 = 0,\ X_6 = 0$$

which produces the following test case:

$$init; PUT\_DATA(\texttt{SET\_MAX\_BALANCE}, 0); PUT\_DATA(\texttt{SET\_MAX\_DEBIT}, 0)$$

The consistency of the resulting constraints system defines the feasibility of the operation sequence; this latter is said to be *feasible* if and only if there exists at least one solution that assigns a correct value to the variables, according to their related constraints.

We now describe the principles of our approach of Scenario-Based Testing that reuses the symbolic animation of the model.

4 Principles of Scenario-Based Testing

Scenario-Based Testing (SBT) is a concept according to which the validation engineer describes by himself scenarios of use cases of the system, thus defining the test cases. In the context of testing software systems, it consists in describing sequences of actions that exercise a particular functionality of the system.

The TOBIAS tool [16] makes it possible to describe scenarios using schemas expressed through regular expressions describing sequences of operation calls and the combination of their parameters. Improving this principle, we have proposed to rely on symbolic animation of formal models of the system in order to unburden the user from providing the parameters of the operations. This makes it possible to focus only on the description of the successive operations, possibly punctuated by intermediate states that guide the steps of the scenario. Scenarios are described using a dedicated language, introduced in [13] that is now presented.

4.1 Scenario Description Language

We present here the language that we use for defining the scenarios. It is based on regular expressions that are then unfolded and played using a symbolic animation engine. This language is composed of 3 layers:

(i) the *sequence* layer, that 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;

(ii) the *model* layer, that describes the operation calls at the model level and constitutes the interface between the model and the scenarios;

(iii) the *directive* layer, that makes it possible to drive the tests that are generated by specifying behavior coverage criteria that will be interpreted by the test generation engine.

4.1.1 Syntax of the Sequence and Model Layers

The syntax of these layers are given in Fig. 2 and Fig. 3. Rule SP describes a state predicate, whereas OP is used to describe the operation calls that can be (i) an

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

REPEAT  ::=  "?" | "{" n "}" | "{," n "}"
          |  "{" n "," m "}"
```

Figure 2. Syntax of the sequence layer

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

OPLIST  ::=  operation_name
          |  operation_name "," OPLIST

SP      ::=  state_predicate
```

Figure 3. Syntax of the model layer

operation name, (*ii*) the $OP keyword, meaning "any operation", or (*iii*) $OP\{OPLIST} meaning "any operation except those of OPLIST".

Rule SEQ describes a sequence of operation calls as a regular expression. A step in the sequence can be a simple operation call, denoted by OP1. Sequences may also 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. Finally a sequence may leads to a state satisfying a state predicate, denoted by SEQ ⇝(SP). This latter represents an improvement w.r.t. usual scenarios description languages, since it makes it possible to define the target of an operation sequence, without necessarily having to enumerate all the operations that compose the sequence.

4.1.2 Syntax of the 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 research for the instantiation of a test scenario. This part of the language is given in Fig. 4.

Rule CHOICE introduces two operators denoted | and ⊗, for covering the branches of a choice. For example, if S_1 and S_2 are two sequences, $\mathtt{S_1} \mid \mathtt{S_2}$ specifies that the test generator has to produce tests that will cover S_1 and other tests that will cover schema S_2, whereas $\mathtt{S_1} \otimes \mathtt{S_2}$ 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 – default option) of 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

```
CHOICE      ::=  "|"                 OP1      ::=  OP | "["OP"]"
              |  "⊗"                           |  "[" OP "/w" CPTLIST "]"
                                               |  "[" OP "/e" CPTLIST "]"
ALL_or_ONE  ::=  "_one"
              |  ϵ                   CPTLIST  ::=  "{" cpt_label ("," cpt_label)* "}"
```

Figure 4. Syntax of the test generation directive layer

behaviors are authorized (/w) or refused (/e). These behaviors are expressed using labels, that have to be linked with actual behaviors of the machine operations. We now give an illustration of this mechanism through an example.

Example 4.1 *(Example of scenario)* Consider the `PUT_DATA` operation from the example in Fig. 1. A piece of scenario that expresses the personalization of the card can be expressed with the following expression:

$$\texttt{PUT\_DATA}\{4\}\ .\ \texttt{STORE\_DATA} \leadsto (\texttt{card\_status = use})$$

that may also be expressed by:

$$\texttt{PUT\_DATA}\{4\}\ .\ [\texttt{STORE\_DATA /w }\{\texttt{ok}\}]$$

if "`ok`" labels the behavior of STORE_DATA that successfully validates the personalization of the card.

4.2 Unfolding and Instantiation of the Scenarios

The scenarios are unfolded so as to obtain the integrality of the sequences described. Each scenario is translated into a Prolog file, directly interpreted by the symbolic animation engine of BZ-Testing-Tools. Each solution provides an instantiated test case. The internal backtracking mechanism of Prolog is used to iterate on the different solutions. Once all the solutions have been explored, the final test suite is obtained. As explained before, the labeling mechanism at the end of the process aims at computing the values of the parameters of the operations composing the test case, so that the sequence is *feasible*.

Example 4.2 *(Scenario unfolding)* Consider the scenario given in the previous example. It requires 4 iterations of the `PUT_DATA` command before a successful invocation of the `STORE_DATA` command, that terminates and validates the personalization. When unfolded, it produces a single symbolic test case (before parameter instantiation), that is:

$$\begin{gathered}\texttt{PUT\_DATA(SET\_MAX\_BALANCE,}X_1\texttt{)}\ .\ \texttt{PUT\_DATA(SET\_MAX\_DEBIT,}X_2\texttt{)}\ .\\ \texttt{PUT\_DATA(SET\_HOLDER\_PIN,}X_3\texttt{)}\ .\ \texttt{PUT\_DATA(SET\_BANK\_PIN,}X_4\texttt{)}\ .\\ \texttt{STORE\_DATA()}\end{gathered}$$

with associated constraints:

$$X_1 \in 0..32767, X_2 \in 0..32767, X_3 \in 0..9999, X_4 \in 0..9999, X_1 > X_2, X_3 \neq X_4$$

due to the fact that to complete the personalization, the maximal balance has to be greater than the maximal debit, and the two PIN codes have to be different.

Notice that changing the scenario to surround `PUT_DATA` with square brackets, such as:

$$[\texttt{PUT\_DATA}]\{4\}\ .\ [\texttt{STORE\_DATA /w }\{\texttt{ok}\}]$$

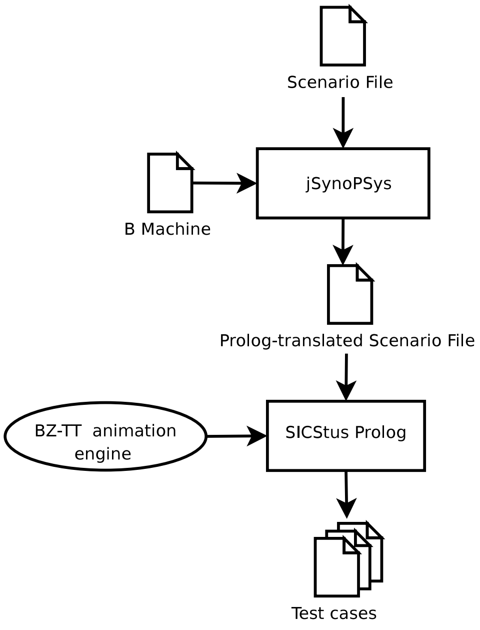
leads to the unfolding of 24 symbolic test cases (representing all the possible combinations of the 4 behaviors that have to be invoked for `PUT_DATA`).

One of the advantages of the Scenario-Based Testing approach that we propose 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, by considering that each scenario addresses a specific requirement. We now present a tool, named jSynoPSys that implements these principles.

5 The jSynoPSys Tool

jSynoPSys is graphical user interface, coupled with the BZ-Testing-Tools animation engine, that makes it possible to perform the Scenario-Based Testing approach presented in the previous sections.

The general architecture of the tool is depicted hereby. The tool takes as input a B abstract machine, on which scenarios will be played. Each scenario is described in a specific file that contains a regular expression, accordingly to the syntax given in Figures 2, 3 and 4. The tool parses the scenario and typechecks the potential B predicates in it. It also checks the existence of behavior labels contained in the scenario. If no error occurs, a Prolog file is produced the contains a translation of the scenario that is now compatible with the Prolog API of the BZ-Testing-Tools animation engine. This scenario is then interpreted by the Prolog Virtual Machine, calling the animation primitives of BZ-TT, in order to build the abstract test cases.



The main frame of the tool is given in Fig. 5. From there, the user has an overview of all the scenarios he has already written for the considered B machine. He may choose:

- to edit the scenario, using the editor frame shown in Fig. 6;
- to parameterize the operations, as shown in Fig. 7, in order to associate the labels used in the scenarios to the behaviors of the operations;
- to run the generation of the test cases. In this context, two options make it possible to limit the number of resulting test cases and/or the test generation time, possibly reducing the combinatorial explosion due to such an approach;

- to visualize the generated test cases, as shown in Fig. 8;
- to export the tests cases in a generic XML format, that contains the tests themselves and the oracle, namely the successive model states encountered when playing the test cases. This file may later on be translated to any test publisher using the adequate translator.

jSynoPSys, like BZ-Testing-Tools, is based on the CLPS-BZ constraint solver that is able of handling data types such as integers (using the classical clp(fd) for finite domain integers), sets, relations, or functions used in the B notation. The AC-3 arc consistency algorithm used in the solver does not make it possible to necessarily detect inconsistencies as soon as they appear in the constraint system. Thus, a labeling phase has to be performed to ensure the consistency of the constraints, i.e., the constraint system is consistent if and only if the labeling succeeds. In our approach, the labeling of a variable may happen at two steps: either when asserting that a given state has been reached, or, eventually, when the test case is instantiated, at the end of the unfolding, in order to assign values to the parameters.

In practice, this approach has shown evidences of scalability, having been exercized on several case studies of various sizes and complexity of manipulated data structures (the Demoney electronic purse: 500 LOC, the POSIX file system standard: 500 LOC, the security model of the IAS plateform: 1000 LOC). These provided results in acceptable times (e.g. 1500 tests –produced with explosive scenarios– in about 300 seconds).

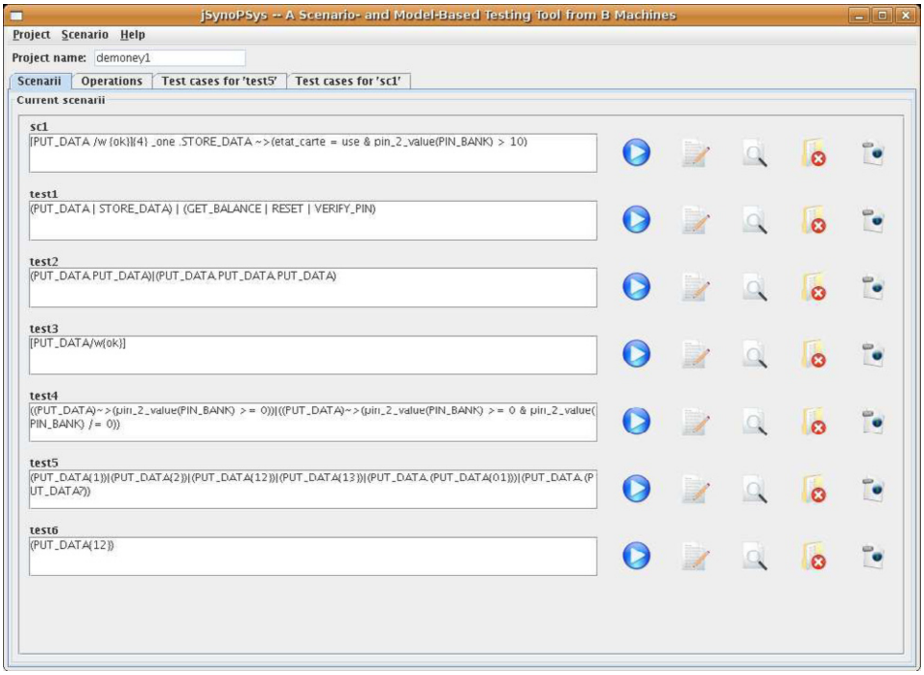
6 Related Work

In the litterature, a lot of scenario based testing works focus on extracting scenarios from UML diagrams, such as the SCENTOR approach [25] or SCENT [21] using statecharts. The SOOFT approach [24] proposes an object oriented framework for performing scenario-based testing. In [6], 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 [3], 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 [9] 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 [14].

The closest work to ours is the TOBIAS tool [16] that works with scenarios expressed using regular expressions representing the combinations of operations

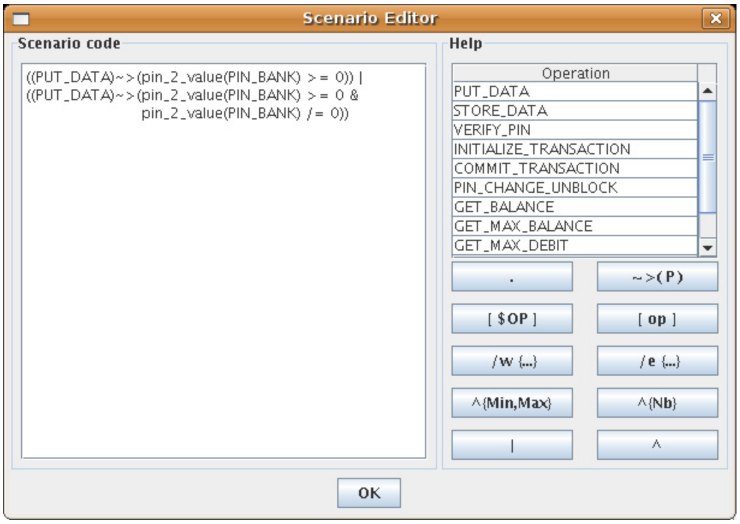


Project name: demoney1

Current scenarii

sc1
```
[PUT_DATA /w {ok}]{4} _one .STORE_DATA ~>(etat_carte = use & pin_2_value(PIN_BANK) > 10)
```
test1
```
(PUT_DATA | STORE_DATA) | (GET_BALANCE | RESET | VERIFY_PIN)
```
test2
```
(PUT_DATA.PUT_DATA)|(PUT_DATA.PUT_DATA.PUT_DATA)
```
test3
```
[PUT_DATA/w{ok}]
```
test4
```
((PUT_DATA)~>(pin_2_value(PIN_BANK) >= 0))|((PUT_DATA)~>(pin_2_value(PIN_BANK) >= 0 & pin_2_value(
PIN_BANK) /= 0))
```
test5
```
(PUT_DATA{1})|(PUT_DATA{2})|(PUT_DATA{12})|(PUT_DATA{13})|(PUT_DATA.(PUT_DATA{01}))|(PUT_DATA.(P
UT_DATA?))
```
test6
```
(PUT_DATA{12})
```

Figure 5. Main frame of the tool

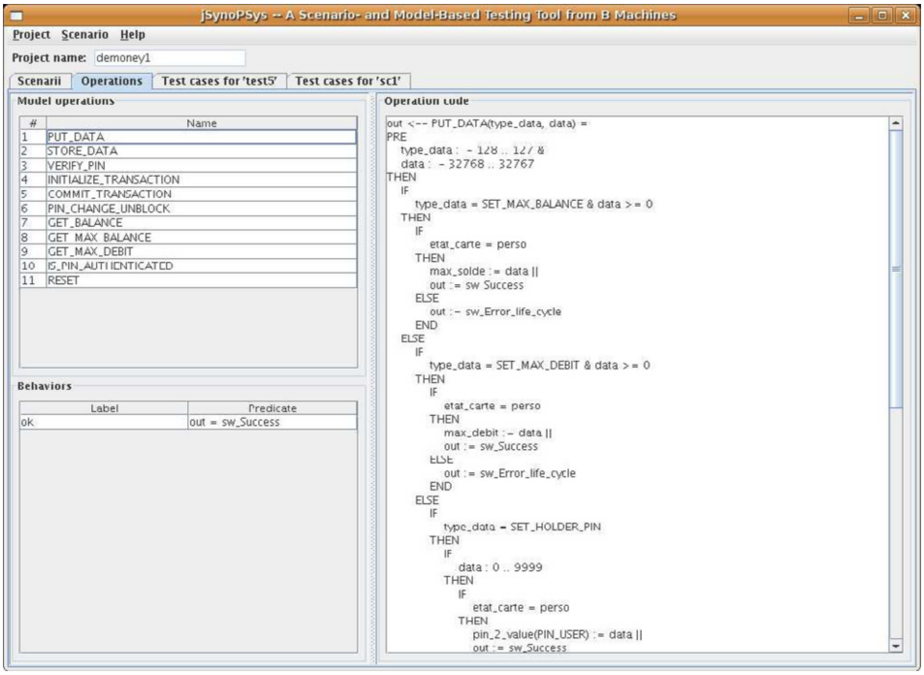


```
((PUT_DATA)~>(pin_2_value(PIN_BANK) >= 0)) |
((PUT_DATA)~>(pin_2_value(PIN_BANK) >= 0 &
                pin_2_value(PIN_BANK) /= 0))
```

Operation
PUT_DATA
STORE_DATA
VERIFY_PIN
INITIALIZE_TRANSACTION
COMMIT_TRANSACTION
PIN_CHANGE_UNBLOCK
GET_BALANCE
GET_MAX_BALANCE
GET_MAX_DEBIT

Figure 6. Scenario edition frame

and parameters. Our approach improves this principle by avoiding to enumerate the combinations of input parameters. In addition, our tool provides test driving

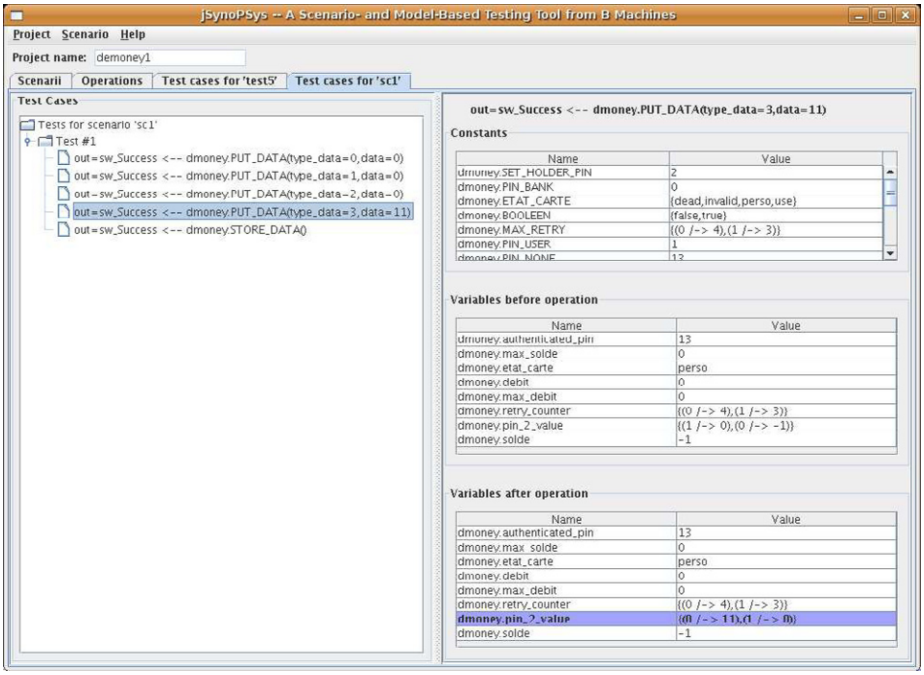


#	Name
1	PUT_DATA
2	STORE_DATA
3	VERIFY_PIN
4	INITIALIZE_TRANSACTION
5	COMMIT_TRANSACTION
6	PIN_CHANGE_UNBLOCK
7	GET_BALANCE
8	GET_MAX_BALANCE
9	GET_MAX_DEBIT
10	IS_PIN_AUTHENTICATED
11	RESET

Label	Predicate
ok	out = sw_Success

```
out <-- PUT_DATA(type_data, data) =
PRE
  type_data : -128 .. 127 &
  data : -32768 .. 32767
THEN
  IF
    type_data = SET_MAX_BALANCE & data >= 0
  THEN
    IF
      etat_carte = perso
    THEN
      max_solde := data ||
      out := sw Success
    ELSE
      out :- sw_Error_life_cycle
    END
  ELSE
    IF
      type_data = SET_MAX_DEBIT & data >= 0
    THEN
      IF
        etat_carte = perso
      THEN
        max_debit :- data ||
        out := sw_Success
      ELSE
        out := sw_Error_life_cycle
      END
    ELSE
      IF
        type_data - SET_HOLDER_PIN
      THEN
        IF
          data : 0 .. 9999
        THEN
          IF
            etat_carte = perso
          THEN
            pin_2_value(PIN_USER) := data ||
            out := sw_Success
```

Figure 7. Operation parameterizing frame



```
Tests for scenario 'sc1'
  Test #1
    out=sw_Success <-- dmoney.PUT_DATA(type_data=0,data=0)
    out=sw_Success <-- dmoney.PUT_DATA(type_data=1,data=0)
    out=sw_Success <-- dmoney.PUT_DATA(type_data=2,data=0)
    out=sw_Success <-- dmoney.PUT_DATA(type_data=3,data=11)
    out=sw_Success <-- dmoney.STORE_DATA()
```

out=sw_Success <-- dmoney.PUT_DATA(type_data=3,data=11)

Constants

Name	Value
dmoney.SET_HOLDER_PIN	2
dmoney.PIN_BANK	0
dmoney.ETAT_CARTE	{dead,invalid,perso,use}
dmoney.BOOLEEN	{false,true}
dmoney.MAX_RETRY	{(0 /-> 4),(1 /-> 3)}
dmoney.PIN_USER	1
dmoney.PIN_NONE	13

Variables before operation

Name	Value
dmoney.authenticated_pin	13
dmoney.max_solde	0
dmoney.etat_carte	perso
dmoney.debit	0
dmoney.max_debit	0
dmoney.retry_counter	{(0 /-> 4),(1 /-> 3)}
dmoney.pin_2_value	{(1 /-> 0),(0 /-> -1)}
dmoney.solde	-1

Variables after operation

Name	Value
dmoney.authenticated_pin	13
dmoney.max_solde	0
dmoney.etat_carte	perso
dmoney.debit	0
dmoney.max_debit	0
dmoney.retry_counter	{(0 /-> 4),(1 /-> 3)}
dmoney.pin_2_value	**{(0 /-> 11),(1 /-> 0)}**
dmoney.solde	-1

Figure 8. Visualization of the test cases

possibilities that may be used to easily master the combinatorial explosion, inherent to such approach. Nevertheless, the TOBIAS input language is more expressive than our language and a combination of these two approaches, that would employ the TOBIAS tool for describing the test cases, is currently under study.

Testing from B specifications is the purpose of two tools. The first one is the Leirios Test Generator [12], a commercial version of the BZ-Testing-Tools environment that is based on the structural coverage of the operations of a B abstract machines. The second one is ProTest [22], that is based on using the ProB model checker [18] for building a state coverage graph of the considered system by exhaustive model checking. The subsequent exploration of this graph builds the test cases. Our approach is different since it does not consider the exploration of concrete state but uses symbolic techniques to avoid such a potentially large enumeration.

Finally, the use of symbolic techniques is also used in white-box testing, notably in Microsoft's Pex [23] that performs .NET program analysis and bounded symbolic execution. Even if the purpose of Pex is not the same as ours –concrete code vs. model operations– this tool works on the same principles as those presented in this paper and both techniques can be seen as interchangeable, modulo data support.

Nevertheless, the combination of symbolic techniques and scenarios for generating tests based on models is an original combination that does not seem to have been targeted before.

7 Conclusion and Future Works

This paper has presented the *jSynoPSys* tool, that combines Scenario-Based Testing techniques with symbolic model animation. This tool relies on a Prolog constraint solver used to perform the animation of the model, in terms of operations chaining, and to instantiate their abstracted parameters. The tool is available for free download at the following address: `http://lifc.univ-fcomte.fr/home/fdadeau/tools/#jSynoPSys`

The tool works with an expressive scenario language based on regular expressions combined with test driving possibilities, that make it possible, for example, to restrict the behaviors appearing in the resulting test cases. Such a feature may also be used to master the combinatorial explosion, and to accelerate the test generation itself, by restricting the possible combinations of behaviors.

Notice that the principles of the tool are generic and can be applied to any formal notation that can be (symbolically) animated. For example, a previous work on the symbolic animation of JML specifications [7], also based on the BZ-Testing-Tools animation engine, could be revisited to use this Scenario-Based Testing feature.

Our future work is led by three main directions. First, we are planning to develop a way for generating the scenarios automatically from higher level properties, formalized in a generic language. In this context, Dwyer's pattern language, introduced in [11] represents an interesting starting point. Second, we are now improving the data coverage algorithm, in order to introduce a boundary analysis of specific state variables or parameters. This principle already existed in the BZ-Testing-Tools framework and is more a technical extension than a real scientific challenge. Nevertheless, we believe that introducing varity in the test data can improve the quality of the resulting test suites. Third, and finally, we are investigating the use of abstractions for producing the test scenarios. Based on an abstraction of the system, a labeled transition system can be automatically produced, from which test scenarios might be derived by an adequate graph exploration algorithm.

References

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

[2] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Proc. of Formal Approaches to Testing of Software, FATES 2002 (workshop of CONCUR'02)*, pages 105–120, Brnö, République Tchèque, August 2002. INRIA report.

[3] 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.

[4] P. Behm and al. Météor: A Successful Application of B in a Large Project. In *FM'99 - Formal Methods -volume 1*, number 1708 in LNCS, pages 348–387. Springer, September 1999.

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

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

[7] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. In J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *Procs of the Int. Conf. on Formal Methods (FM'2005)*, volume 3582 of *LNCS*, pages 75–90, Newcastle Upon Tyne, UK, July 2005. Springer-Verlag.

[8] 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.

[9] D. Clarke, T. Jéron, V. Rusu, and E. 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.

[10] F. Dadeau, A. de Kermadec, and R. Tissot. Combining scenario- and model-based testing to ensure posix compliance. In *ABZ'2008, International Conference on ASM, B and Z*, volume 5238 of *LNCS*, pages 153–166, London, United Kingdom, September 2008. Springer. To appear.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[12] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated Test Generation from B Models. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 277–281, Besançon, France, January 2007. Springer.

[13] 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.

[14] 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.

[15] J-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In *CARDIS'98*, number 1820 in LNCS. Springer, 1998.

[16] 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.

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

[18] 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.

[19] R. Marlet and D. Le Metayer. Security properties and java card specificities to be studied in the secsafe project, 2001.

[20] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, addison-wesley edition, 1999.

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

[22] 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.

[23] N. Tillmann and J. de Halleux. Pex–white box test generation for .net. In *2nd International Conference on Tests and Proofs*, pages 134–153, April 2008.

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

[25] 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.