Instantiation of Parameterized Data Structures for Model-Based Testing

Fabrice Bouquet, Jean-François Couchot, Frédéric Dadeau, Alain Giorgetti

Instantiation of Parameterized Data Structures for Model-Based Testing

Fabrice Bouquet, Jean-François Couchot, Frédéric Dadeau, and Alain Giorgetti

LIFC – INRIA Cassis project
FRE CNRS 2661, University of Franche-Comté
16 route de Gray, 25030 Besançon cedex, France
{bouquet,couchot,dadeau,giorgetti}@lifc.univ-fcomte.fr

Abstract. Model-based testing is bound, by essence, to use the enumerated data structures of the system under test (SUT). On the other hand, formal modeling often involves the use of parameterized data structures in order to be more general (such a model should be sufficient to test many implementation variants) and to abstract irrelevant details. Consequently, the validation engineer is sooner or later required to instantiate these parameters. At the current time, this instantiation activity is a matter of experience and knowledge of the SUT. This work investigates how to rationalize the instantiation of the model parameters.

It is obvious that a poor instantiation may badly influence the quality of the resulting tests. However, recent results in instantiation-based theorem proving and their application to software verification show that it is often possible to guess the smallest most general data enumeration. We first provide a formal characterization of what a most general instantiation is, in the framework of functional testing. Then, we propose an approach to automate the instantiation of the model parameters, which leaves the specifier and the validation engineer free to use the desired level of abstraction, during the model design process, without having to satisfy any finiteness requirement.

We investigate cases where delaying the instantiation is not a problem. This work is illustrated by a realistic running example. It is presented in the framework of the BZ-Testing-Tools methodology, which uses a B abstract machine for model-based testing and targets many implementation languages.

1 Introduction

Model-based testing (MBT) [7] is the process of using a formal model to derive tests cases that are to be run on an implementation, named the *system under test* (SUT). The model is designed by a validation engineer from an informal specification, without looking at the implementation, except for the signatures of control and observable methods. Nevertheless, several factors influence the design of this model, among which the fact that the test methodology only supports finite data structures.

When writing a formal model from an informal specification, the validation engineer develops a complementary skill of formal specifier. Thus, he/she can

J. Julliand and O. Kouchnarenko (Eds.): B 2006, LNCS 4355, pp. 94–108, 2006.

take benefit of some specific features of formal modeling. One of them is the possibility to abstract details of the informal specification by designing an initial model with parameters. In the B method, this first model is called an abstract machine, and the parameters can be either machine parameters or abstract (i.e. not enumerated) sets. Then, coming back to his/her validation activity, the engineer has to make all the model data finite, by instantiating them cleverly. Up to now, this instantiation is performed by hand from the specifier's knowledge of the SUT and the informal specification.

However, since the engineer does not (have to) know all the implementation and informal specification details, his/her instantiation work is somewhat artificial and not optimal, neither in time nor in quality. Indeed, a poor instantiation may not exploit all the possibilities of the model: it may leave "dead code" in it, and no test case will be produced for this dead part of the model, leaving a –possibly important– part of the SUT not validated.

The first contribution of this work is the formalization (as a proof obligation) of the "most general instantiation" of a formal model with respect to a coverage criterion, corresponding to the idea of leaving no execution case without an associated test. Checking this proof obligation corresponds to dead code detection. The second contribution is to show how to discharge this proof obligation in a theorem prover or a constraint solver. The third contribution is a method based on sorted logic to find an approximation of the most general instantiation. This work is presented in the framework of the BZ-Testing-Tools [1], an approach performing model-based testing from B machines.

The paper is organized as follows. Section 2 presents a running example, a GSM11-11 specification that will be used to illustrate our approach. Section 3 introduces the principles of model-based testing, as performed within the BZ-Testing-Tools. The proof obligation defining the most general instantiation is given in Sect. 4. The techniques for solving this proof obligation are detailed in Sect. 5. The novel instantiation method proposed to guide them is presented in Sect. 6. Finally, Section 7 concludes and presents future work.

2 Running Example

Our running example is a simplified B model of the interface between the Subscriber Identity Module (SIM) and the Mobile Equipment (ME) within the GSM 11.11 (Global System for Mobile communication) digital cellular telecommunications system. It is based on an informal specification [8] produced by the Special Mobile Group (SMG). Section 2.1 briefly presents the aims of the GSM 11.11 standard and describes a parameterized B model of a fragment of it, written for test purposes. Then, Section 2.2 analyzes a former experience on this example where the instantiation was at the charge of the specifier.

2.1 Informal and Formal Specifications

The GSM 11.11 is a standard for the second generation of mobile phones. In this system, the mobile phone embeds a writable card (the SIM: Subscriber Identity

Module) containing security and application data. The SIM stores data in files hierarchically organized in a tree structure. The tree root and the other tree internal nodes are respectively called the *master file* (*MF*) and the *dedicated files* (*DF*). They are the *directories* of the file structure. The tree leaves are called the *elementary files* (*EF*).

```
MACHINE
  gsm1(FILES)
CONSTANTS
  MF, DF,
  EM, /* Elementary Files under the MF */
  ED, /* Elementary Files under a DF */
  FA,
  mf, dg, dt
PROPERTIES
  MF ⊆ FILES ∧ DF ⊆ FILES ∧
  EM ⊆ FILES ∧ ED ⊆ FILES ∧
  MF ∩ DF = ∅ ∧ MF ∩ EM = ∅ ∧
  MF ∩ ED = ∅ ∧ DF ∩ EM = ∅ ∧
  DF ∩ ED = ∅ ∧ EM ∩ ED = ∅ ∧
  FILES = MF ∪ DF ∪ EM ∪ ED ∧
  FA ∈ ED ⟶ DF ∧
  mf ∈ FILES ∧ dg ∈ FILES ∧ dt ∈ FILES ∧
  MF = { mf } ∧ DF = { dg, dt } ∧
  ei ∈ EM
VARIABLES
  cd, cf
INVARIANT
  cd ∈ (MF ∪ DF) ∧ cf ⊆ (EM ∪ ED) ∧
  card (cf) ≤ 1 ∧
  ( cf = ∅ ∨
    (cf ≠ ∅ ∧ cf ⊆ ED ∧ cd ∈ DF) ∨
    (cf ≠ ∅ ∧ cf ⊆ EM ∧ cd = mf) )
INITIALISATION
   cd := mf || cf := ∅
OPERATIONS
  sw ⟵ SELECT_FILE(ff) =
  PRE
    ff ∈ FILES
  THEN
    IF (ff ∈ (DF ∪ MF))
    THEN
      /* The last selected file is cd */
      IF (
        ( cd = mf ∧ ff ∈ DF ) ∨
        ( cd = dg ∧ ff = dt ) ∨
        ( cd = dt ∧ ff = dg ) ∨
        ( cd ∈ DF ∧ ff = mf ) ∨
        ( ff = cd ) ∨ ff = mf ))
      THEN
        cd := ff || cf := ∅ || sw := 9000
      ELSE
        sw := 9404 /* Not activable. */
      END
    ELSE /* ff is an EF */
      IF (
        ( ff ∈ EM ∧ cd = mf ) ∨
        ( ff ∈ ED ∧ cd ∈ DF
          ∧ FA(ff) = cd ) ∨ ff ∈ cf )
      THEN
        cf := {ff} || sw := 9000
      ELSE
        sw := 9404
      END
    END
  END
END
```

Fig. 1. A small B model for the GSM 11-11 SIM - ME interface

During a communication between the SIM and the ME (Mobile Equipment), the SIM is passive: it only answers to requests sent by the ME, which reads and modifies the SIM files through functions defined in the communication interface. Our model focuses on the SELECT function of this communication interface, because it is the only one which interacts in a complex manner with the file structure. This B machine, shown in Fig. 1, is simplified and its identifiers are shortened in order to fit in the format of this paper. We now describe it in details.

The `gsm1.mch` machine is parameterized with the non empty finite set *FILES* of all the files present on the SIM card. Since the master file is unique, it is modeled in B by the *mf* constant. The SIM card can store files for many applications but our model focuses on the main one, namely the GSM application, whose dedicated files are all directly under the *MF*. Consequently, the model distinguishes four types of files: master file, dedicated file, elementary file under the master file and elementary file under a dedicated file. These four types are respectively modeled in B by the four pairwise disjoint sets *MF*, *DF*, *EM* and *ED* whose

union is the set *FILES* of all the SIM files[1]. The property $MF = \{mf\}$ states that there is a unique master file and the property $DF = \{dg, dt\}$ fixes the set of dedicated files used by the GSM application. These two identifiers respectively represent the directories DF_GSM and DF_TELECOM containing the application data and some telecommunication service features. The file structure is completely defined by the total function *FA* which maps each elementary file in *ED* to its *FA*ther, the dedicated file containing it. The data are completed with the *ei* constant, representing the *EF_ICCID*, an *EF* at the *MF* level storing a unique identification number for the SIM.

The SELECT function is the sole function which can select a SIM file. More precisely, it aims at selecting a directory to become the new *current directory* and an *EF* to become the new *current elementary file*, in conformance with some access rules. It is modeled by the *SELECT_FILE* operation, which assigns a value to the two state variables *cd* (current directory) and *cf* (current *EF*).

The initialization chooses the master file as current directory. One can infer from the informal specification that there is always a single selected directory. Consequently, the variable *cd* takes its values in $MF \cup DF$. The informal specification also expects that there is always zero or one selected *EF*. Consequently, the variable *cf* is defined as a set of elementary files (i.e. is included in $EM \cup ED$) and its value is the empty set when no *EF* is selected. The property that its cardinality should be zero or one is added to the machine invariant. The last part of the invariant checks that the current *EF*, when selected, is always a child of the current directory.

The SELECT function works as follows. If the candidate for selection *ff* is a directory, it is selected iff it is the master file, the current directory, an immediate child, a sibling or the father of the current directory. Now, if *ff* is an *EF*, it is selected iff it is a child of the current directory or was already selected.

2.2 A Previous Experience

In a former work [2], members of our team have published a B model for the GSM 11.11 (named `gsm_revue.mch`), where all the sets were enumerated and where the file structure was modeled by a binary relation, itself enumerated as a set of pairs. After generating tests sequences, the authors have noticed that a branch of the model was never activated (this phenomenon is reproduced by the first ELSE branch of our running example, marked with `/* Not activable. */`).

The explanation for this phenomenon can be threefold. Firstly, the informal specification may be contradictory. Secondly, there may be a discrepancy between the informal specification, assumed contradiction-free, and branches of the B model, making these branches inconsistent. Thirdly, the enumeration of sets in `gsm_revue.mch` may be too restrictive to activate each branch.

After a closer look, the main explanation appeared to be a discrepancy between specifications: The informal specifier has indeed written that "Selecting a *DF* or the *MF* [always] sets the current directory", whereas the formal specifier

[1] Note that, for this use of the B machine for generating tests, it is equivalent to consider the *FILES* machine parameter as an abstract set.

has added a case of failure for this selection. The test campaign revealed that this case was not activable, for the enumerated file structure of `gsm_revue.mch`, but the question remained open, whether this property was general or due to a too restrictive enumeration. Our contribution is a methodology, described in Sect. 4, 5 and 6, to answer such a question. In the present case, it gives two answers. Firstly, the branch remains dead for any tree structure of height 1, meaning that the specifier choice of one *DF* and four *EF* was general enough for such a detection. Secondly, this branch becomes activable when one considers at least one dedicated file of depth 2.

From this example, it is clear that detecting "dead code" in a B model is a cumbersome and error-prone activity. We want to investigate ways to assist it with tools. Since the first two explanations for dead branches involve an informal side, they cannot be fully automatized. We therefore focus on the third explanation, i.e. an instantiation of data which is not general enough to activate each branch, by setting a framework where this instantiation is automatically performed.

This instantiation guessing method is based on tools supporting hereditary finite data structures. This excludes inductive structures and binary relations in all their generality. Thus, the model `gsm_revue.mch` has been revised by replacing the binary relation defining the file structure with a total function associating its father to each elementary file. When limited to *DF* of depth 1, this leads to `gsm1.mch`, used in the following sections to illustrate how the guessing method proceeds to find a good instantiation of the sets of files in this tree structure. The method will prove that no instantiation of these sets can activate the branch marked `/* Not activable. */` in Fig. 1, i.e. that the specifier choice in `gsm_revue.mch` was not that restrictive.

Now, in order to prove that a *DF* of depth 2 is sufficient to activate this dead branch, we have also written a larger model, named `gsm2.mch`[2]. Finally, note that the idea of considering such DFs, excluded in version 5.0.0 of the standard, is not artificial, since version 6.2.0. of the GSM 11.11 standard allows their existence.

3 Principles of Model-Based Testing from B Machines

This section describes model-based testing (MBT) from B machines, as performed in the BZ-Testing-Tools [1]. This process takes as an input a B abstract machine, representing the system under test (e.g. wiper controller, smart card, speed control device, etc.) from a functional point of view. Test targets are derived from this model according to different coverage criteria, chosen by the validation engineer. Once the test target is defined, the model is animated (using a boundary model-checking approach) in order to build a complete test case. This test generation process relies on a set-theoretic solver, named CLPS-BZ [3], interfaced with constraint logic programming.

[2] Available at `http://lifc.univ-fcomte.fr/~couchot/specs/gsm2.mch`.

We present in this section the principles of test target definition, and the associated coverage criteria. Then we introduce the CLPS-BZ solver. Finally, we present the test target conditions of consistency.

3.1 Definition of the Test Targets

The BZ-Testing-Tools approach considers a test case as the activation of a system behavior within a pertinent system state. This represents the *behavioral coverage*. In addition, a *decision coverage* is considered to cover the different possibilities of a disjunctive decision predicate, providing a specific coverage criterion. Finally, the *data coverage* is obtained by a boundary analysis of the data –input parameters and state variables– involved in the behavior. These three items give the outline of the subsection.

Behavioral Coverage. A behavior can be seen as an operation in which no branching exists. It is computed as a path in the control flow graph of a B machine operation, in which each branching structure (IF, ASSERT or CHOICE substitutions) creates a choicepoint. The behavioral coverage of the B machine consists in producing one test target for the activation of each behavior, by considering an *activation condition* for each behavior, as the conjunction of all the predicates along the considered path.

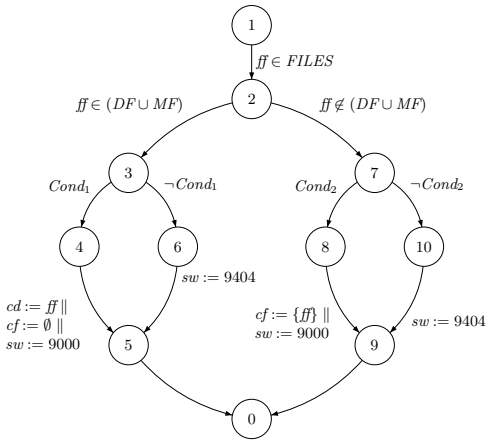

Fig. 2. Control-flow graph of the *SELECT_FILE* operation

The Fig. 2 presents the control flow graph of the SELECT_FILE operation, from the running example, where $Cond_1$ is the first IF condition that is

$$(cd = mf \wedge ff \in DF) \vee (cd = dg \wedge ff = dt) \vee$$
$$(cd = dt \wedge ff = dg) \vee (cd \in DF \wedge ff = mf) \vee$$
$$(ff = cd) \vee (ff = mf)$$

and $Cond_2$ is the second IF condition that is

$$(ff \in EM \wedge cd = mf) \vee (ff \in ED \wedge cd \in DF \wedge FA(ff) = cd) \vee ff \in cf.$$

Definition 1 (Set of Activation Conditions). *The set of activation conditions of a substitution is the set of activation conditions for each behavior extracted from an operation. We denote by act(Op) the set of activation conditions for an operation Op.*

Decision Coverage. The decision coverage is achieved by performing different rewritings on the disjunctive predicates labeling the control-flow graph.

We consider four rewritings, each one deserving a particular decision coverage criterion. Table 1 distinguishes these rewritings. It consists in creating a bounded choice ([]) between the different elements of the rewriting, expanding the control flow graph in as many subgraphs.

Table 1. Definition of the rewritings of the disjunctive predicates

Id	Rewriting of $P_1 \vee P_2$	Decision Coverage
1	$P_1 \vee P_2$	DC and SC
2	P_1 [] P_2	D/CC
3	$(P_1 \wedge \neg P_2)$ [] $(\neg P_1 \wedge P_2)$	FPC
4	$(P_1 \wedge P_2)$ [] $(P_1 \wedge \neg P_2)$ [] $(\neg P_1 \wedge P_2)$	MCC

Rewriting 1 (RW1, for short) consists in leaving all the disjunctions unchanged. This rewriting satisfies the Decision Coverage (DC, for short), and Statement Coverage (SC) criterion. Rewriting 2 (RW2, for short) consists in creating a choice between the two predicates. Thus, the first branch and the second branch independently have to succeed when being evaluated. This rewriting satisfies the Decision/Condition Coverage criterion (D/CC) since it satisfies the DC and the Condition Coverage (CC) criteria. Rewriting 3 (RW3, for short) consists in creating an exclusive choice between the two predicates. Only one of the sub-predicates of the disjunction is checked at one time. This rewriting satisfies the Full Predicate Coverage (FPC) [11] criterion. Rewriting 4 (RW4, for short) consists in testing all the possible values for the two sub-predicates to satisfy the disjunction. This rewriting satisfies the Multiple Condition Coverage (MCC) criterion.

The decomposition of operation *SELECT_FILE* from the example into RW1-behaviors is given by Table 2.

Data Coverage. The data coverage consists in performing a boundary analysis of the data that are involved in the behaviors, depending on their types. A boundary analysis consists in selecting a data value at its extremum (either minimum or maximum) of its domain within the context of the behavior activation. The extremum is chosen depending on the data types; basically, atoms are enumerated, integers are selected at their bounds, sets are selected as their

Table 2. RW1-Behaviors extracted from the example

Behavior	Activation Condition
b_1	$ff \in FILES \wedge ff \in MF \cup DF \wedge ((cd = mf \wedge ff \in DF) \vee$ $(cd = dg \wedge ff = dt) \vee (cd = dt \wedge ff = dg) \vee$ $(cd \in DF \wedge ff = mf) \vee (ff = cd) \vee (ff = mf))$
b_2	$ff \in FILES \wedge ff \in MF \cup DF \wedge (cd \neq mf \vee ff \notin DF) \wedge$ $(cd \neq dg \vee ff \neq dt) \wedge (cd \neq dt \vee ff \neq dg)$ $(cd \notin DF \vee ff \neq mf) \wedge (ff \neq cd) \wedge (ff \neq mf)$
b_3	$ff \in FILES \wedge ff \notin MF \cup DF \wedge ((ff \in EM \wedge cd = mf) \vee$ $(ff \in ED \wedge cd \in DF \wedge FA(ff) = cd) \vee ff \in cf)$
b_4	$ff \in FILES \wedge ff \notin MF \cup DF \wedge (ff \notin EM \vee cd \neq mf) \wedge$ $(ff \notin ED \vee cd \notin DF \vee FA(ff) \neq cd) \wedge ff \notin cf$

minimal and maximal cardinality. This data selection is recursively performed on the elements of pairs.

It is important to notice that this step requires finite data structures so that a bound for data can be selected.

3.2 The CLPS-BZ Constraint Solver

Originally designed to animate B machines, the CLPS-BZ constraint solver [3] is a set-theoretic solver combined with a finite domain solver on integers. It allows the acquisition and the evaluation of constraints written using B-like basic operators.

CLPS-BZ uses an arc-consistency algorithm, whose worst-case complexity is $\mathcal{O}(ek^3)$ where e is the number of constraints and k is the cardinality of the largest data domain, for checking the satisfiability of the constraint system. Such an algorithm checks only the consistency between the adjacent edges within the constraint graph. As a consequence, the consistency of the whole constraint system can only be ensured by the enumeration of the solutions, performed using a forward-checking labeling algorithm, whose complexity is $\mathcal{O}(ek^2)$.

3.3 Behavior Consistency Condition

Each test case targets one behavior, extracted from the machine operations. Thus, a test case is relevant only if the target behavior is activable, i.e. its activation condition is consistent. The following definition formalizes this notion of behavior consistency, in the context of the machine properties and invariant.

Definition 2 (Behavior Consistency). *A behavior b_i is* consistent *iff the formula*

$$P \wedge B \wedge (\exists X \,.\, I \wedge a_i) \tag{1}$$

is satisfiable, where P (resp. I) is the predicate of the PROPERTIES (resp. INVARIANT) clause, X is the tuple of the machine state variables, a_i is the activation condition corresponding to the behavior b_i and B is the formula

$$\bigwedge_{E} \bigwedge_{e_j, e_k \in E}^{1 \leqslant j < k \leqslant l} e_j \neq e_k$$

precising that the elements of the set E are pairwise distinct, for each set $E = \{e_1, \ldots, e_l\}$ enumerated in the SETS clause.

$$\begin{array}{l}
\textit{/* Part coming from the PROPERTIES clause P */} \\
MF \subseteq FILES \wedge DF \subseteq FILES \wedge EM \subseteq FILES \wedge ED \subseteq FILES \wedge \\
MF \cap DF = \emptyset \wedge MF \cap EM = \emptyset \wedge MF \cap ED = \emptyset \wedge DF \cap EM = \emptyset \wedge \\
DF \cap ED = \emptyset \wedge EM \cap ED = \emptyset \wedge FILES = MF \cup DF \cup EM \cup ED \wedge \\
FA \in ED \longrightarrow DF \wedge mf \in FILES \wedge dg \in FILES \wedge dt \in FILES \wedge \\
MF = \{mf\} \wedge DF = \{dg, dt\} \wedge e_i \in EM \wedge \\
\exists\, cd, cf \,. \quad \textit{/* Part coming from the INVARIANT clause I */} \\
\quad cd \in (MF \cup DF) \wedge cf \subseteq (EM \cup ED) \wedge \mathtt{card}(cf) \leq 1 \wedge \\
\quad \big(cf = \emptyset \vee (cf \neq \emptyset \wedge cf \subseteq ED \wedge cd \in DF) \vee \\
\qquad (cf \neq \emptyset \wedge cf \subseteq EM \wedge cd = mf)\big) \wedge \\
\quad \exists\, ff \,. \quad \textit{/* Activation condition of behavior } b_4 \textit{ */} \\
\qquad ff \in FILES \wedge ff \notin MF \cup DF \wedge (ff \notin EM \vee cd \neq mf) \wedge \\
\qquad\quad (ff \notin ED \vee cd \notin DF \vee FA(ff) \neq cd) \wedge ff \notin cf
\end{array} \tag{2}$$

Fig. 3. b_4 consistency proof obligation

For instance, Fig. 3 shows the consistency condition of the fourth RW1-behavior extracted from our example, provided the *FILES* set is enumerated.

Up to now, all the sets were enumerated and the existential quantifications in (1) were expanded in disjunctions. This consistency was checked by the CLPS-BZ solver and the inconsistent behaviors were used to eliminate test cases.

However, the detection of many inconsistent behaviors could be a sign of weakness of the formal model with respect to the testing methodology, namely a too narrow instantiation of its data structures. Our purpose is to improve the testing methodology by adding a tool that guesses a "good" instantiation. We therefore formalize in the next section a notion of most general instantiation for a B model, that makes all its behaviors activable.

4 Most General Instantiation

Intuitively, we are looking for an instantiation of a B machine that makes each of its behaviors activable from at least one of the reachable machine states.

Under the assumption that the reachable states are characterized by the INVARIANT clause, this condition can be formalized by the following definition, where $F(X_i)$ denotes the formula obtained from formula F by replacing the tuple of state variables X with X_i.

Definition 3 (Activation Condition). *All the behaviors of a B machine are* activable *if*

$$P \wedge B \wedge \bigwedge_{\{i \mid b_i \textit{ is a machine behavior}\}} \exists\, X_i \,.\, I(X_i) \wedge a_i(X_i) \tag{3}$$

is satisfiable for each behavior b_i, where P, B and a_i have the same meaning as in Sect. 3.3 and X_i is a distinct tuple of state variables for each behavior b_i.

The tuple of state variables in (3) is distinct for each behavior, because each behavior may be activable from a different reachable state.

Now, a model of (3) is an instantiation with enumerated sets of all the B machine parameters and abstract sets. We suggest to call it the ***most general instantiation***, since it makes the instantiated machine as general as the parameterized one, for a given testing coverage criterion. Another feature of this instantiation is that it minimizes the sum of the cardinalities of the instantiating sets.

In practice, the method to compute this most general instantiation is twofold. First of all, the behaviors that are not activable for any instantiation are detected by checking the satisfiability of (1) without enumerating the parametric sets. The specifier is informed that his/her specification contains some dead code whatever the parameter values are. Then, the inconsistent behaviors are ignored and a constraint solver is combined with an instantiation procedure to find an instantiation that make all the remaining consistent behaviors activable in a reachable state.

The next sections detail the techniques involved in this method.

5 Checking the Consistency

Finding a model for (3) may take benefit of any satisfiability decision procedure: a negative answer suggests that the specifier should modify the model whereas a positive one ensures the existence of a general instantiation and is therefore an intermediate step before computing it. This satisfiability can be checked either with a suitable prover or with a constraint solver, provided the data structures are first made finite.

In the proof-based approach, we exploit the existing bam2rv[3] tool that translates set-theoretic formulas into first order equational formulas ready to be discharged in the haRVey prover [6]. The choice of this tool is motivated by its compliance with set-theoretic formulas and its scalability [5].

Before applying the constraint-based approach, each machine parameter is constrained to be equal to (or included in) an arbitrary enumerated superset. For instance, the set $S = \{a01, a02, a03, \ldots, a30\}$ can be used as a superset of *FILES*, since the informal specification [8] allows a maximum of 30 files to exist on the card. The resulting constraints are then discharged into the CLPS-BZ solver, already used in the model-based testing methodology presented in Sect. 3.

In practice, for efficiency reasons, we first check whether each behavior is consistent, and then check the satisfiability of (3) restricted to the consistent behaviors. Table 3 summarizes the experimental results[4] obtained with each tool

[3] `http://lifc.univ-fcomte.fr/~giorgett/Rech/Software/bam2rv/index.html`

[4] Run on a P4 2.4 GHz with 640Mb RAM.

applied to the RW1-behaviors from Table 2. The third (resp. fourth) column gives the time consumed to check the satisfiability with the constraint $FILES \subseteq S$ (resp. $FILES = S$). The fifth column gives the first instantiation found by CLPS-BZ. The last column gives the time consumed by the proof-based approach with haRVey.

Table 3. Consistency results for the RW1-behaviors

Behavior	Satisfiable?	$\text{CLPS}_{\subseteq}$	$\text{CLPS}_{=}$	Instance	haRVey
b_1	yes	0.4 s	0.3 s	$FILES = \{dg, dt, mf\}, ff = dg,$ $DF = \{dg, dt\}, MF = \{mf\},$ $ED = \emptyset, EM = \{e_i\}, FA = \emptyset$	0.4 s
b_2	no	0.3 s	0.3 s		0.4 s
b_3	yes	0.4 s	0.3 s	$FILES = \{dg, dt, mf, e_i\}, ff = e_i,$ $DF = \{dg, dt\}, MF = \{mf\},$ $ED = \emptyset, EM = \{e_i\}, FA = \emptyset$	0.5 s
b_4	yes	> 1 h	0.5 s		0.3 s

Globally, an interesting result is that proving a consistency **for any value** of *FILES* with the haRVey prover is not much more time consuming than checking it with CLPS-BZ **for a unique enumeration** of *FILES*, with 30 elements.

A second result is that all the methods answer that the behavior b_2 is not activable. As announced in Sect. 2, the corresponding behavior extracted from the larger model `gsm2.mch` is proved to be activable by haRVey (or by CLPS-BZ after enumerating *FILES* with 30 elements) in less than one second.

Finally, the main result concerns b_4. For that behavior the enumeration strategy of CLPS-BZ takes too much time (more than one hour) when *FILES* is constrained to be included in S. It is so because the detection by arc-consistency is not sufficient and CLPS-BZ continues by instantiating from the initial *FILES* enumeration, which is too large. The next section will address this problem. For the moment, since haRVey gives a result, the combination of both tools is satisfactory.

The next step of the method, checking the satisfiability of (3), presents the same difficulty as the former one, since (3) is just a generalization of (1) to many behaviors. Hence, in (3), a coarse choice of the superset size would again make the solver diverge. The next section presents a method that aims at avoiding such a combinatorial explosion by restricting the size of the superset S.

6 Sort-Based Instantiation

The previous section ends with the idea that coarsely bounding the size of the machine parameter may make the instantiation process diverge. In order to reduce this problem, this section puts a bridge between the investigated challenge of guessing a "good" instantiation and classical methods from automated reasoning in first order many-sorted logics.

The underlying idea is that the sets we want to instantiate often come from a partitioning of more global sets, hence are pairwise disjoint. Consequently, they can be seen as sorts and the consistency and activation conditions can be seen as formulas to satisfy in a first order many-sorted logic. In such a logic, the choice of an Herbrand universe for each sort corresponds to the instantiation of the associated set. Another use of a many-sorted version of the Herbrand interpretation to verify software can be found for instance in [9].

Let us now detail the instantiation method derived from this simple idea and illustrate it on the consistency condition of behavior b_4.

A static analysis of the CONSTANTS and PROPERTIES clauses can detect that some abstract sets partition larger abstract sets. This analysis starts with the abstract sets declared as machine parameters (or in the SETS clause). They are considered as primary sorts. Then, the analysis iterates the following two steps: firstly, it considers that any set inclusion whose right member is a sort defines its left member as a sort too; secondly, it checks whether the sorts introduced so are pairwise disjoint. In our example, it is obvious that the primary sort is *FILES* and that the other sorts are *MF*, *DF*, *EM* and *ED*. The predicates defining the sorts are then removed from the formula. In (2), the first three lines of predicates are removed so.

It is now possible to assign one sort or more to each variable. We consider the non obvious case where a variable may have many sorts i.e. when it belongs to (or is included into) a union of sets. For instance the predicate $cd \in MF \cup DF$ is interpreted by cd is either of sort MF or of sort DF. Similarly, the elements of cf are either of sort EM or of sort ED.

Each set-theoretic predicate (cardinality, inclusion, equality) is firstly translated into a formula where the sole predicates are equality and membership. The set equality is decomposed into two inclusions. Each union (resp. intersection) is subsequently translated into a disjunction (resp. conjunction). For instance

$$cf \subseteq (EM \cup ED) \wedge \texttt{card}(cf) \leqslant 1 \tag{4}$$

is translated into

$$\forall x \,.\, (x \in cf \Rightarrow (x \in EM \vee x \in ED)) \wedge \forall x, y \,.\, (x \in cf \wedge y \in cf) \Rightarrow x = y \tag{5}$$

where x and y are fresh variables whose sort comes from cf, i.e. is either EM or ED.

We are left with deciding the satisfiability of the formula in a many-sorted first order logic with equality and membership. We classically begin with considering a Skolem form of the formula. For instance, the quantifications on cd, cf and ff in (2) are removed and these variables are replaced with the fresh constants $\overline{cd}$, $\overline{cf}$ and $\overline{ff}$.

A formula of a first order many-sorted logic is satisfiable if and only if it has a many-sorted Herbrand model [9]. We compute then the *graph of sort dependency* whose nodes are labelled with the sorts and whose oriented edges encode the dependency between sorts: a sort s depends of the sorts $s_1, \ldots, s_n$ if there is a functional symbol whose signature is $s_1, \ldots, s_n \rightarrow s$. These functional symbols

are either already present in the quantified formula or are introduced by the skolemization step. For instance, Figure 4 shows the graph resulting from the skolemization of (2). It contains only one edge, which goes from the sort *ED* to the sort *DF* since the sole non constant functional symbol in the skolemization of (2) is *FA*.

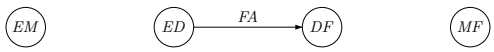

Fig. 4. Graph of sort dependency of (2)

We compute then the Herbrand universe by firstly considering the constants of each sort and secondly building new terms with the functional symbols in the formula in accordance with the sorts. The computation (and thus the resulting Herbrand universe) is finite if and only if the graph of sort dependency is acyclic, which is the case for our example.

An interesting point in this procedure is the treatment of terms that may have many sorts, like $\overline{cd}$ with the sorts *MF* and *DF* in our example. This corresponds to a disjunction that a thinner notion of behavior could decompose. We suggest in this case to add a distinct fresh constant for each sort. For instance, for the behavior b_4, cd_{4a} and cd_{4b} are respectively added in the sorts *MF* and *DF*.

This choice has the advantage that it is meaningful for the specifier: each new constant corresponds to a distinct execution case in a behavior. Moreover, the constant name can encode this case (like *a* and *b* in our example), thus offering a complete traceability of the origin of each constant. The result is obviously an upper approximation of the desired instantiation, but it reflects the degree of precision of the coverage criterion that has produced these behaviors.

One may think that this procedure could be refined by interpreting each enumerated set on its set of constants (for instance *MF* interpreted on $\{mf\}$ and *DF* on $\{dg, dt\}$). However, such an optimization would require equating two constants and propagating this information through terms and sorts. Since this propagation is already developed in the CLPS-BZ solver, we suggest not to implement this optimization, but let the solver do the remaining work.

For the behavior b_4 of the running example, the sort-based instantiation is $MF \subseteq \{mf, \overline{cd_{4a}}, \overline{ff_{4a}}\}$, $DF \subseteq \{dg, dt, \overline{cd_{4b}}, \overline{ff_{4b}}, FA(\overline{ff_{4d}})\}$, $EM \subseteq \{e_i, \overline{ff_{4c}}\}$ and $ED \subseteq \{\overline{ff_{4d}}\}$. Then, starting from this result, CLPS-BZ finds the following most general instantiation, $MF = \{mf\}$, $DF = \{dg, dt\}$, $EM = \{e_i\}$, $ED = \{\overline{ff_{4d}}\}$, later called mgi_1, in less than one second.

Finally, the sort-based instantiation found for the conjunction of the three consistent behaviors b_1, b_3 and b_4 is $MF \subseteq \{mf, \overline{cd_{1a}}, \overline{ff_{1a}}, \overline{cd_{3a}}, \overline{ff_{3a}}, \overline{cd_{4a}}, \overline{ff_{4a}}\}$, $DF \subseteq \{dg, dt, \overline{cd_{1b}}, \overline{ff_{1b}}, FA(\overline{ff_{1d}}), \overline{cd_{3b}}, \overline{ff_{3b}}, FA(\overline{ff_{3d}}), \overline{cd_{4b}}, \overline{ff_{4b}}, FA(\overline{ff_{4d}})\}$, $EM \subseteq \{e_i, \overline{ff_{1c}}, \overline{ff_{3c}}, \overline{ff_{4c}}\}$ and $ED \subseteq \{\overline{ff_{1d}}, \overline{ff_{3d}}, \overline{ff_{4d}}\}$. Again in less than one second, CLPS-BZ finds a most general instantiation that appears to be mgi_1 again.

7 Conclusion and Future Work

The global aim of the present work was to unburden the specifier from instantiating the parameters of his/her formal model, when this model is designed to guide the automated generation of tests. When this formal model is a B machine, it proposed a way to assist the instantiation phase, by guessing an enumeration that is sufficient to be employed within the test generation process. The resulting enumeration is general enough for activating all the consistent behaviors extracted from the B machine operations. If it exists, then this enumeration serves to initialize the SUT. Otherwise, it means that no data is suitable for testing the wholeness of the SUT. The specifier is then invited to cut his/her model in separate parts, in a way that remains to be defined.

This approach is related to test data generation with tools such as Korat [4]. Korat aims at producing complex Java structures (such as balanced trees, etc.) from a boolean method describing the properties of the structure and a bound on the size of the structure. This approach aims at providing test data as inputs for Java unit tests. Basically, one may think that our approach is similar, since we both rely on constraint solving for instantiating the data structures. Nevertheless, the bound used by Korat is user-defined, whereas our approach proposes to automatically compute it.

For conciseness, this work has been restricted to the RW1 decision coverage, but it is directly extensible to the RW2, RW3 and RW4 ones. Furthermore, the sort-based instantiation is presented for one level of sorts, but holds for many levels too. Since the key is an acyclic graph of sorts, the method is suitable for many data structures like arrays, trees of bounded depth (as seen in the `gsm2.mch`) ...

It is important to notice that although our approach has been presented in the context of model-based testing, it can be employed for other purposes such as model-checking using ProB [10]. Indeed, the ProB model-checker also requires finite data domains. Such an instantiation phase could be performed as a preprocess, that would allow all the operations to be activable, improving the results of a model verification.

References

1. 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 Contraint Logic Programming. In *Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR'02*, pages 105–120, 2002.
2. 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.
3. 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, 2004.
4. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA'02: Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133. ACM Press, 2002.

5. J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Scalable automated proving and debugging of set-based specifications. *Journal of the Brazilian Computer Society (JBCS)*, 9(2):17–36, 2003. ISSN 0104-6500.
6. D. Déharbe and S. Ranise. Applying light-weight theorem proving to debugging and verifying pointer programs. In *ENTCS*, volume 86. Elsevier, 2003.
7. I.K. El-Far and J.A. Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 1:825–837, 2002.
8. European Telecommunications Standards Institute. *GSM Technical Specification*, 1995. `http://www.ttfn.net/techno/smartcards/gsm11-11.pdf`.
9. Pascal Fontaine and E. Pascal Gribomont. Decidability of invariant validation for parameterized systems. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, pages 97–112. Springer, 2003.
10. M. Leuschel and M. Butler. ProB: A model checker for B. In *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.
11. A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *5th International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–. IEEE Computer Society, 1999.