Model-Based Robustness Testing in Event-B using Mutation

Aymerick Savary[1,2], Marc Frappier[1], Michael Leuschel[3], and Jean-Louis Lanet[2]

[1] Université de Sherbrooke
[2] Université de Limoges
[3] University of Düsseldorf

Abstract. Robustness testing aims at finding errors in a system under invalid conditions, such as unexpected inputs. We propose a robustness testing approach for Event-B based on specification mutation and model-based testing. We assume that a specification describes the valid inputs of a system. By applying negation rules, we mutate the precondition of events to explore invalid behaviour. Tests are generated from the mutated specification using ProB. ProB has been adapted to efficiently process mutated events. Mutated events are statically checked for satisfiability and enability using constraint satisfaction, to prune the transition search space. This has dramatically improve the performance of test generation. The approach is applied to the Java Card bytecode verifier. Large mutated specifications (containing 921 mutated events) can be easily tackled to ensure a good coverage of the robustness test space.

Keywords: Robustness Testing, Specification Mutation, Model-Based Testing, Vulnerability Analysis, Intrusion Testing, Event-B, ProB

1 Introduction

Functional testing aims at finding errors in the functionality of a system, e.g., testing that the correct outputs are produced for correct inputs. In contrast, *robustness testing* aims at finding errors in a system under invalid conditions, such as unexpected inputs. Various strategies can be used for system specification. A specification may describe the behaviour for valid inputs only, for instance by using preconditions and postconditions. In that case, an input that does not satisfy the precondition is considered as invalid. A specification may describe the behaviour for all possible inputs, detailing error messages to be produced in case of invalid inputs. In that case, robustness testing coincide with functional testing, because the specification covers both valid and invalid inputs.

Model-based testing (MBT) aims at generating tests from a specification. When the analysis of a specification can be automated, MBT can automate the production of tests and provide systematic coverage of the test space at a reasonable cost. Formal specification languages are particularly suitable for automated MBT. Yet, few systems are formally specified in practice. Automated

MBT can become an incentive for using formal specifications if the coverage obtained is better than manually derived tests, at a comparable cost. However, if the specification considers only valid inputs, then automated MBT cannot exercise a good coverage of invalid inputs, because the specification is not built for that, and test generation techniques typically only cover valid input sequences. In that particular case, model-based functional testing is unable to adequately cover robustness testing.

In this paper, we propose a mutation-based approach to deal with model-based robustness testing. Mutation testing has been typically applied to programs to evaluate the adequacy of tests. A good set of tests should identify faults in mutated programs. We take a different view-point and use specification mutation to identify invalid behaviour and then apply automated MBT on mutated specifications to generate tests for robustness testing of an implementation. In particular, we focus on the mutation of preconditions, by providing a set of rules for computing the negations of a precondition. The advantages are two-fold. First, the behaviour for valid inputs can often be abstracted and simplified; for robustness testing, there is no need to describe these cases in detail, because they are not part of the test objective. A specification built for robustness testing does not need to be detailed enough to prove the correctness of an implementation. This helps in reducing the cost of building a formal specification. For instance, in this work we are targeting robustness testing of the Java Card bytecode verifier (JCBCV) [12]. We do not need to build a complete specification of the JCBCV in order to generate tests for invalid bytecode programs. We simply need to focus on the conditions that characterize valid bytecode programs, and by negation, we obtain the conditions of invalid bytecode programs. In other words, a specification built for robustness testing can be much simpler than a specification built to describe the full functional behaviour of a system. Second, the mutation process allows us to provide fine grain invalid conditions in order to ensure good coverage of invalid inputs. A model checker can then be used to exercise these fine grain negated conditions and select tests for very specific conditions. We use ProB [10] for that purpose. For instance, the condition $A = B$, where A and B are sets, can be negated in various ways: A is empty and B is not, A is strictly included in B, A and B are disjoint, A and B are not disjoint, etc. A MBT technique will provide test criteria in order to decide which cases should be covered. In our approach, we use negation rules to build mutants that identify these cases, so that we can reuse a model checker on the mutants to exercise the desired test cases. This provides a greater level of automation and simplifies the construction of model-based test generation tools. Moreover, we ensure that the test cases are disjoint and that mutants always generate invalid traces, thus no unnecessary tests are generated. This is especially important for embedded systems like Java Cards.

Our approach is particularly interesting for penetration testing, which is a special kind of robustness testing; it aims at finding security faults. For instance, a JCBCV checks that a Java Card application satisfies the security constraints specified in the Java virtual machine (JVM) specification. The JVM specification

prescribes a precondition and a postcondition for each bytecode instruction. Robustness testing aims at checking that a JCBCV will reject invalid bytecode programs. If a JCBCV accepts an invalid bytecode program, then a vulnerability has been identified in the JCBCV. Such vulnerabilities may be exploited to define attacks on Java-based smart cards.

The rest of this paper is structured as follows. Section 2 provides an overview of our robustness test generation approach. Section 3 describes the Event-B model of the JCBCV used for our case study. Section 4 describes our approach for mutating Event-B specifications and negating predicates of the Event-B language. Section 5 describes the improvements made to ProB in order to efficiently carry out model-based test generation. Section 6 describes the application of our approach to the case study and compares the results with a previous version of this work presented in [15]. Section 7 compares our approach with similar work in MBT and mutation testing. Section 8 concludes this paper with an appraisal of our work and an outlook on future work.

2 Overview of the Approach

2.1 The Event-B Method

Event-B [1] is a state-based, event-driven modelling notation. Event-B models are developed through stepwise refinement. An Event-B model is composed of two parts, a static part composed of *contexts* and a dynamic part composed of *machines*. A context defines constants and constraints on its constants called axioms. A machine has state *variables*, *invariants*, which describe properties of state variables, and *events*. An event is composed of a guard and an action. An event can be triggered when its guard is satisfied; the execution of the event's action can modify the machine state. One must prove that each event execution preserves the machine invariants. Event-B refinement allows for *behaviour refinement* (*i.e.*, reducing non-determinism, guard strengthening, event splitting/merging, and introduction of new events) and for *data refinement* (*i.e.*, adding new state variables and replacing state variables).

2.2 Overview of Robustness Test Generation

Figure 1 provides an overview of the robustness test generation process. It takes as input an Event-B project which contains a set of refinements and a set of contexts. In this paper, we focus on the mutation of machines. We assume that this input model describes the valid traces of the system. Our approach generates invalid traces of this model by mutating its events. An Event-B model that refines another model inherits all events of the model it refines. In order to have all events in a single model, the first step is to merge these refinements into a single model that contains all events. This flattened model is then analyzed by the model mutator to generate mutants for each event. A mutant event is obtained by negating the guard of an event. Negation rewrite rules are applied

to the guard of an event. A negation rule can produce several mutants, thus an event can be mutated into several mutants. The mutant events are added to the original model, so that the final mutant model contains both the original events and their mutants. The mutant model is then analyzed with the constraint-based checker of ProB to generate traces of the mutant model. Coverage in MBT is typically decomposed into data coverage criteria and structural coverage criteria. Our data coverage criteria are determined by the negation rules. Each mutant event identifies an invalid case to cover. Structural coverage is ensured by our breadth-first search algorithm in ProB. A test consists of a trace of the form $[e_1, \ldots, e_n, \overline{e}_{n+1}]$, where e_i with $i \in 1..n$ is a valid event, and $\overline{e}_{n+1}$ is a mutant event which denotes an invalid event. A mutant event is contained in at most one trace (unreachable mutants appear in no trace), in order to minimize the number of tests generated. No false negatives are generated, that is, each generated trace should be rejected by the system under test (SUT). Each test considers a specific test case for a given event. For robustness testing of most systems, there is no need to include more than one invalid event in a test, because the first invalid event should be detected by the SUT and rejected. If it is not possible to observe that an invalid input was rejected immediately after it was submitted, then additional events may be added after the invalid event. However, this requires a specification that covers both valid and invalid cases, which is more time consuming to build, but it does provide a better coverage than our approach, which aims at automating robustness testing with the least specification effort. Our algorithm uses a breadth-first search to generate traces of the mutant Event-B model. The search is bounded by a maximal depth.

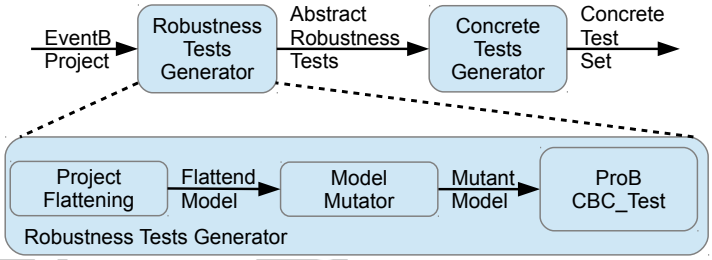

Fig. 1. Overview of the Robustness Test Generation Process

3 Formal Model of the bytecode Verifier

The JCBCV is a complex system and we model it using stepwise refinement to deal with complexity. We do not model the bytecode verifier itself, but the bytecode instructions. A trace of our model represents the byte array of a method. To simplify the test generation, we always generate a static method with three

parameters whose types are set to byte, short and reference. We use some predetermined classes, called *TestClassA*, *TestClassB* and *TestClassC*, which are modelled in an EVENT-B context.

3.1 Java Card Instruction sets

We partition Java Card instructions, called *JCInstructions*, into four sets.

1. *Return* : instructions that exit a method,
2. *Branching* : branching instructions,
3. *FieldAccess* : instructions for reading or writing object fields,
4. $Linear = JCInstructions \setminus Return \setminus Branching$: instructions that do no branch or return.

Our model covers $JCInstructions \setminus Branching \setminus FieldAccess$, for a total of 62 bytecode instructions, using 66 events. Branching instructions (41) require a more complex control structure that we plan to add in the near future. Field access instructions (32) are not difficult to model; we simply need to add a model of objects to take these instructions into account.

3.2 The Refinements

Our model is decomposed into seven layers of refinement. Each model introduces a new concept and defines abstract instructions which are successively refined. Concrete instructions corresponding to bytecode instructions are introduced only in the final level. The first level represents the return concept. It introduces two abstract instructions, one that denotes linear instructions and one that denotes returning instructions. A state variable *programRunning* is initialized to *true* and set to *false* by a return instruction. A guard prevents instructions to be executed after a return. The second refinement introduces variable *stackSize*, whose value is bounded by constant *MaxStackSize*, and defines an abstract instruction for each type of stack size update. The third refinement introduces guards to check that enough elements or enough space is available in the stack. The fourth refinement introduces the stack itself, whose elements are java types, not java values. The inheritance tree on types is also introduced. The fifth refinement deals with local variables, which represent either method parameters or method local variables. The sixth refinement deals with object initialization and the constant pool. The seventh refinement introduces the concrete instructions.

3.3 The State Model

The Java Card Inheritance Tree Java Card types can be represented by a semilattice, with an artificial *Top* element and type compatibility can be checked using this semilattice. Two types are compatible if their least upper bound is not *Top*. Figure 2 represents the semilattice. The interfaces and arrays of references are not currently taken into account. The darker elements are only usable

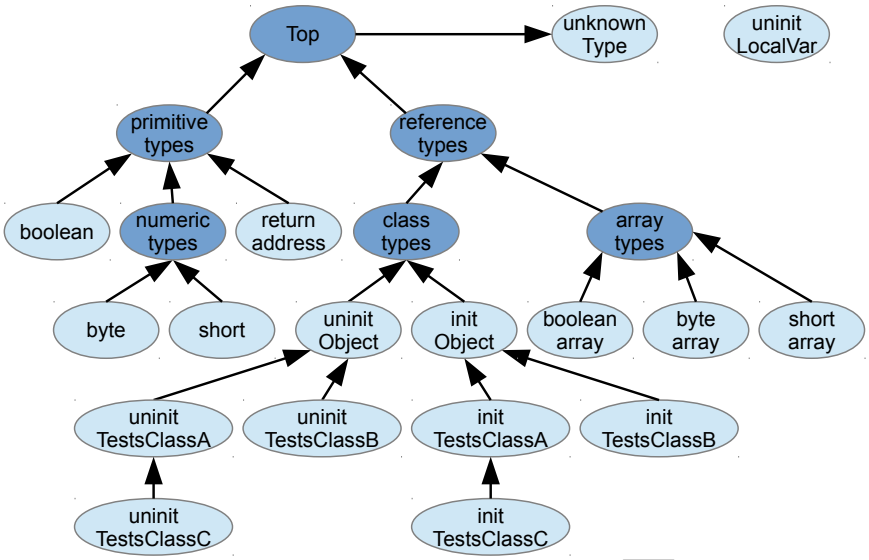

Fig. 2. Java Card semilattice

for type inference. The lighter elements represent concrete Java types. Type *unknownType* is used to represent memory access violations like stack overflow. Type *unInitLocalVar* denotes values of uninitialized local variables.

The stack in a specification of valid inputs only would be modeled as partial function $stack \in 0\,..\,MaxStackSize - 1 \nrightarrow TYPES$. Such a model is inadequate to represent stack overflows or stack underflows, since a model checker like ProB will not find elements outside a valid stack. The mutation of the invariant could potentially solve such problems, but it is hard to derive a general rule that would properly manage all types of Event-B variable. Instead, we decided to manually change this type to an appropriate value that models invalid cases, that is, *stack* domain si extended to $-4\,..\,MaxStackSize - 1$. We have determined by analysis of the bytecode language that at least 4 negative index positions are needed to generate stack underflow attacks (*e.g.*, instruction *swap*).

Object initialization involves four instructions. Since we use a bread-first search to generate traces, tests that require an initialized object are longer to generate. To simplify this, we use a single event to represent these four instructions of initialization.

4 Event-B Specification Mutation

4.1 Mutation of an Event-B Machine

An Event-B machine contains several parts. The mutation process only alters the list of events of a machine, by adding new events which are mutations of

existing events. Let $M.E$, $M.V$, $M.I$ respectively denote the set of events, the set of variables and the set of invariants of machine M.

$$\begin{aligned} M.V &:= M.V \cup \{eutExecuted\} \\ M.I &:= M.I \cup \{eutExecuted \in BOOL\} \\ M.E &:= M.E \cup \bigcup_{e \in M.E} mutate(e) \end{aligned}$$

Variable *eutExecuted* is added to control the generation of tests. It ensures that an invalid event has been added to the trace. It is initialised to *FALSE* in the initialisation event of M.

4.2 Mutation of an Event

An event of an Event-B machine has the following general form.

$$\begin{array}{lll} \textbf{Event}\ e \mathrel{\widehat{=}} & \textbf{any} & \ldots, \quad v_i \quad, \ldots \\ & \textbf{where} & \ldots, \quad \mathtt{grd_j} : f_j \quad, \ldots \\ & \textbf{then} & \ldots, \mathtt{act_k} : w_k := t_k, \ldots \\ & \textbf{end} & \end{array}$$

The "**any**" part introduces local variables v_i of the event, which represent event parameters. The "**where**" part introduces the guard of the event, which consists of a list of labeled formula $\mathtt{grd_i} : f_i$, where $\mathtt{grd_i}$ is the label of formula f_i. The formula of the list are implicitly conjoined to make the event's guard. Guards are also used to type the local variables.

A mutation of an event is computed as follows. Only a subset of the guard's formula are negated; these formula are manually tagged by the specifier by adding suffix "`_t`" at the end of their label. The tagged formula are conjoined into a single formula f_t, and untagged formula are also conjoined to form a second formula f_u. The choice of the formula to negate depends on the problem at hand and the test objectives. A mutant event computed by $mutate(e)$ has the following form, where f'_t is a negation of f_t. We define in the next section how f'_t is computed.

$$\begin{array}{lll} \textbf{Event}\ e \mathrel{\widehat{=}} & \textbf{any} & \ldots, v_i\ , \ldots \\ & \textbf{where} & \mathtt{grd_t} : f'_t, \\ & & \mathtt{grd_u} : f_u \\ & & \mathtt{grd_{eut}} : eutExecuted = FALSE \\ & \textbf{then} & \mathtt{act_{eut}} : eutExecuted := TRUE \\ & \textbf{end} & \end{array}$$

The guard of a mutant event is composed of the untagged formulas left unchanged and a negation of the tagged formulas. The actions of the original event are replaced with *SKIP*, which leaves the state unchanged. This choice is as good as any other state modification, since we assume that the specification only deals with valid inputs. Moreover, we do not extend a trace ending with an invalid event, as discussed in Section 2.2.

4.3 Negation of a Formula

We say that f' is a negation of a formula f iff it satisfies one of the following two constraints.

$$f' \Rightarrow \neg f \quad (1) \qquad\qquad f' \Rightarrow \neg\, WD(f) \quad (2)$$

Constraint (1) says that when f' holds, $\neg f$ holds. Thus, a negation f' is not equivalent to $\neg f$; it can be stricker. Constraint (2) caters for partial operators of the Event-B language. Event-B uses a two-value logic. To ensure that each formula has a meaning, Event-B contains proof obligations that must be discharged for each formula that uses a partial operator. For instance, to make sure that $x = y \div z$ is well-defined, one must prove that $z \neq 0$. Thus, such a predicate involving a partial operator is typically used within a formula of the form $z \neq 0 \Rightarrow x = y \div z$, so that the well-definedness proof can be discharged. Predicate $WD(f)$ holds when formula f is well-defined, that is, all operators used in f are called within their domain of definition. By negating it, we ensure that we test partial operators for undefinedness.

A negation f' denotes a robustness test case of f. By controlling the form of f', we determine the data coverage criteria of our MBT approach. The negations of a formula f for constraint (1) are computed using a set of rewrite rules of the form $neg(f) \rightsquigarrow \{f'_1, \ldots, f'_n\}$, where $neg(f)$ is an inductively defined operator. A single rule is defined for each connective and predicate of the Event-B language. To ensure coherence and completeness of the negation process, one should prove for each rule that the set of negations is equivalent to $\neg f$ (*i.e.*, $\neg f \Leftrightarrow f'_1 \vee \ldots \vee f'_n$). To ensure that test cases are disjoint and to minimize the set of tests generated, one should also prove for each rule that negations are mutually disjoint (*i.e.*, $\bigwedge_{i \neq j} \neg(f'_i \wedge f'_j)$). We have defined negation rules for each predicate and logical connective of the Event-B language. We provide below a few illustrative examples. For the sake of concision, we use the following convention: all connectives are lifted point-wise to sets, such that, for instance, $f \wedge neg(g)$, with $neg(g) = \{g_1, \ldots, g_n\}$, denotes $\{f \wedge g_1, \ldots, f \wedge g_n\}$.

$$\begin{aligned}
neg(p_1 \wedge p_2) &\rightsquigarrow (neg(p_1) \wedge p_2) \cup (p_1 \wedge neg(p_2)) \cup (neg(p_1) \wedge neg(p_2)) \\
neg(p_1 \vee p_2) &\rightsquigarrow neg(p_1) \wedge neg(p_2) \\
neg(\forall x \cdot p) &\rightsquigarrow \exists x \cdot neg(p) \\
neg(i_1 < i_2) &\rightsquigarrow \{i_1 = i_2\} \cup \{i_1 > i_2\} \\
neg(i_1 \leq i_2) &\rightsquigarrow \{i_1 > i_2\} \\
neg(e \in E) &\rightsquigarrow \{e \notin E\} \\
neg(E_1 \subset E_2) &\rightsquigarrow \{E_1 \neq \varnothing \wedge E_2 = \varnothing\} \cup \{E_1 = \varnothing \wedge E_2 = \varnothing\} \cup \\
&\quad \{E_1 \neq \varnothing \wedge E_1 = E_2\} \cup \{E_2 \neq \varnothing \wedge E_2 \subset E_1\} \cup \\
&\quad \{E_1 \cap E_2 \neq \varnothing \wedge E_1 \not\subseteq E_2 \wedge E_2 \not\subseteq E_1\} \cup \\
&\quad \{E_1 \neq \varnothing \wedge E_2 \neq \varnothing \wedge E_1 \cap E_2 = \varnothing\}
\end{aligned}$$

The negation rule for conjunction generates all possible subsets of negated conjuncts. The negation rule for strict subset inclusion generates all cases consid-

ering disjointness and emptiness of operands. This last rule illustrates the importance of proving coherence, completeness and disjointness. Model finders like ProB and Alloy are quite useful to debug negation rules on discrete structures like sets, relations and functions.

The negation rules for well-definedness are of two kinds. The rules for connectives, predicates and total operators simply propagate well-definedness negation to their operands, *e.g.*,

$$negWD(p_1 \wedge p_2) \rightsquigarrow negWD(p_1) \cup negWD(p_2)$$

The following rules cater for the partial operators of Event-B.

$$negWD(i_1 \div i_2) \rightsquigarrow negWD(i_1) \cup negWD(i_2) \cup \{i_2 = 0\}$$
$$negWD(f(x)) \rightsquigarrow negWD(f) \cup negWD(x) \cup \{x \notin dom(f)\}$$
$$negWD(min(E), max(E), inter(E)) \rightsquigarrow negWD(E) \cup \{E = \varnothing\}$$

The rules for terms which are either constants or variables terminate the recursion defining $negWD$ with $negWD(symbol) \rightsquigarrow \varnothing$. Rules for well-definedness negation are complementary to rules for negation. For instance,

$$neg(z \neq 0 \Rightarrow x = y \div z) = \{z \neq 0 \wedge x \neq y \div z\}$$

whereas

$$negWD(z \neq 0 \Rightarrow x = y \div z) = \{z = 0\}$$

5 Model-Based Testing Algorithm Improvements and Performance

In MBT one wants to generate traces and values which satisfy a certain coverage criterion. There are two ways this coverage can be achieved systematically: using model checking or using a constraint-based approach. In this paper we have used the constraint-based approach: it can deal much better with constants which have many possible values and with events whose parameters have many possible values (like the Java Card instructions). Driven by this case study, the performance of the constraint-based test generation has been considerably improved, and the algorithm has been made more intelligent by using statically computed enabling and feasibility information. This can both considerably speed up the test generation algorithm and provide better user feedback. Indeed, often the algorithm can terminate earlier, as the algorithm is not trying to cover infeasible events, and then provide the user with informative feedback that certain events can definitely never be covered.

5.1 Feasibility Analysis

In this case study one has a very large number of events: events corresponding to the original bytecode instructions (66) and their mutations (921). It is to be

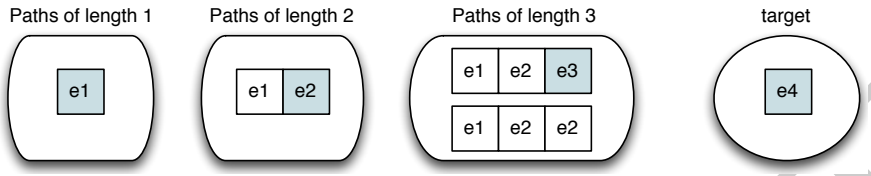

Fig. 3. Sample Run of the MBT Algorithm

expected that many mutants cannot ever be covered, and a first improvement lies in detecting as many of the uncoverable events before starting the MBT algorithm proper. This is done by the feasibility analysis, which calls the ProB constraint solver to check for every event with guard G whether it can find a solution for the axioms, invariant and G. If no such solution exists, then the event is marked as *infeasible* and ignored in the main MBT algorithm below. If a solution was found, the invariant of the machine admits a state where the guard G of the event is true; whether such a state can actually be reached is precisely the task of the main MBT algorithm. If a time-out occurs, then the event is considered potentially feasible and not ignored in main MBT algorithm.

5.2 Enabling Analysis

Our enabling analysis for MBT computes two types of information for every event e:

- $enable(e)$: the set of events f that can go from disabled to enabled after executing e in states that satisfy the invariant. The ProB constraint solver solves the constraint $\neg Grd_f \wedge Inv \wedge BA_e \wedge Grd'_f$, where BA_e is the before-after predicate of e and Grd_f, Grd'_f are respectively the guard of f applied to the before state and the after state. In case of a time-out it is assumed that f is in $enable(e)$. We extend the function $enable$ for paths p to be either $Events$ if $p = []$ and equal to $enable(last(p))$ otherwise.
- $feasibleAfter(e)$: the set of events f that can be enabled after executing e. The constraint solver solves the constraint $Inv \wedge BA_e \wedge Grd'_f$. In case of a time-out it is assumed that f is in $feasibleAfter(e)$. We again extend the function $feasibleAfter$ for paths p to be either $Events$ if $p = []$ and equal to $feasibleAfter(last(p))$ otherwise.

In the absence of time-outs, we have that $enable(e) \subseteq feasibleAfter(e)$.

5.3 Main MBT Algorithm

The constraint-based test generation algorithm implemented within ProB is a breadth-first algorithm, which maintains a list of paths (aka sequences of events) which are feasible, i.e., for which ProB has found a solution for the constants,

initialisation and parameters of all involved events. Figure 3 shows a sample of such paths; the paths ending with a newly covered event (shown in blue) are tests; the other paths have not yet been useful in covering new events, but may be extended into paths which cover new target events.

To not distract from the essentials, we present a simplified version of the algorithm in Fig. 4. The full algorithm, can also deal with target predicates in addition to target events.

The breadth-first algorithm gives priority to generating new test cases: for a given depth, it will first try to cover new target events (by appending a target event t to existing paths p of length *depth*). For example, when reaching depth 3 in Figure 3, we would first try $[e_1, e_2, e_3, e_4]$ (provided $e_4 \in enable(e_3)$) and then $[e_1, e_2, e_2, e_4]$ (provided $e_4 \in enable(e_2)$). Only after all candidates $p \leftarrow t$ of a given length have been processed, will it generate paths which end with an event e that has already been covered. In Figure 3 this would be $[e_1, e_2, e_3, e_1]$ (provided $e_1 \in feasibleAfter(e_3)$) and $[e_1, e_2, e_3, e_2]$ and so on. Note that the use of the auxiliary variable $target'$ in line 16 of Fig. 4 is to avoid checking an event e a second time (in case we generated a new test case for it in line 9). For efficiency, the user can also specify certain events to be *final*; the algorithm will never try to extend a path ending with a final event. For example, if e_3 were declared final we would not attempt any path extending $[e_1, e_2, e_3]$. In our case study, the target events are the mutants, and all mutants events are also declared final.

6 Experimentation and Comparison

In [15], an earlier version of this robustness testing approach is described. This paper improves on [15] in the following ways. First, it speeds up the mutation process. In [15], mutants were generated using the Rodin platform. One mutant model was generated for each mutant of an event, since ProB could not be efficiently used on a large model containing all mutants. This could not scale up as we tried to cover more bytecode instructions in our tests. The model used in [15] covered 12 instructions of the Java Card bytecode language. We are now taking into account 62 bytecode instructions using 66 events. It takes 24 hours using Rodin to build the environment for all 921 mutant models generated from these 66 events. Rodin being based on Eclipse, it generates an internal representation of each Event-B model, which is very time consuming for large models. In the new approach, we directly use ProB to produce a single mutant model containing all mutant events. The 921 mutant events are now generated in 10 minutes. Negation rules can produce mutants who are unreachable, either because their guard is unsatisfiable or because it can never be reached from the initial state. These unreachable mutants were dramatically slowing down the MBT process of ProB. We have estimated that the approach presented in [15] would take 2 years to analyze the 921 mutant models. With the new approach, a model containing all mutant events can be analyzed for MBT in 45 mins. It generates 223 tests (one for each reachable mutant).

1. **Input:** set of events $target \subseteq Events$ and a set of events $final \subseteq Events$
2. Initially we set $paths := \{[]\}$ where [] is the path of length 0
3. $depth := 0$
4. $final$ is the set of events which have to be final; in our case study: $target$
5. $target' := target$
6. **for** every $p \in paths$ of length $depth$ **do**:
7. **for** every $t \in target \cap enable(p)$ **do**:
8. **if** solve constraints of path $p \leftarrow t$ **then**
9. $target := target \setminus \{t\}$, store solution as test for t
10. $path := path \cup \{p \leftarrow t\}$
11. **fi**
12. **od**
13. **od**
14. **if** $target = \varnothing$ or maximum depth reached **then** return **fi**
15. **for** every $p \in paths$ of length $depth$ **do**:
16. **for** every $e \in feasibleAfter(p) \setminus target' \setminus final$ **do**:
17. **if** solve constraints of path $p \leftarrow e$ **then** $path := path \cup \{p \leftarrow e\}$ **fi**
18. **od**
19. **od**
20. $depth := depth + 1$; **goto** 5

Fig. 4. MBT Algorithm using Enabling Analysis

We are also proposing a new Event-B model that takes into account more features of the Java Card bytecode language. We use a semilattice of types to cater for type hierarchy. We also take into account operand stack underflows and access to local variables outside a method's frame. This leads us to identify guidelines for modelling in Event-B in the context of robustness testing. Typing of Event-B variables must be adapted to cater for invalid access. Finally, we have added new negation rules in order to get 100 % coverage with respect to a manually derived set of tests for the JCBCV and to cater for quantifiers in first-order logic. In particular, we take into account the well-definedness of expressions with partial operators (*e.g.*, function application outside its domain, like a division by zero).

The following example shows the modelling of instruction *aload_3* that pushes the local variable (*localVariables*) at index 3 ($prm\_index = 3$) on the stack. Event parameter *push_1* is used as an alias to represent the element, which is a Java type, to push. The guards to mutate (2, 5 and 8) are tagged with `_t`. The other guards do not need to be mutated, since they represent execution control information rather than the functional behaviour of the instruction.

Event *aload_3_R07* $\widehat{=}$
  any
    prm_index
    push_1
  where
    `grd1 :` $programRunning = TRUE$

grd2_t : $stackSize < MaxStackSize$
grd3 : $push\_1 \in TYPES$
grd4 : $prm\_index \in dom(localVariables)$
grd5_t : $prm\_index \leq MaxLocalVariablesIndex$
grd6 : $push\_1 = localVariables(prm\_index)$
grd7 : $prm\_index = 3$
grd8_t : $referenceTypes \mapsto push\_1 \in Lattice$
then
act1 : $stackSize := stackSize + 1$
act2 : $stack := stack \cup \{stackSize \mapsto push\_1\}$
end

We illustrate below two mutations (amongst 11) of instruction **aload_3**. We only show the mutated parts.

Event *aload_3_EUT_47* $\widehat{=}$
...
where
grd1 : ...
grd2_t : stackSize = MaxStackSize
grd_eut : eutExecuted = FALSE
then
act_eut : eutExecuted := TRUE
end

Event *aload_3_EUT_53* $\widehat{=}$
...
where
grd1 : ...
grd2_t : stackSize = MaxStackSize
grd8_t : referenceTypes $\mapsto$ push_1 $\notin$ Lattice
grd_eut : eutExecuted = FALSE
then
act_eut : eutExecuted := TRUE
end

These two mutants generate the following two tests.

1. $[INIT, aconst\_null, astore(3), aconst\_null, aload\_3\_EUT\_47, return]$
2. $[INIT, aload\_3\_EUT\_53, return]$

The first trace does a stack overflow with $MaxStackSize = 1$. The second one pushes an uninitialized local variable on the stack and does a stack overflow with $MaxStackSize = 0$, which exercises two faults.

The 223 generated tests were executed on the JCBCV provided by Oracle in the Java Card SDK [13]. It failed on three tests, which are all related to implicit type conversion on local variables. In the Java Card specification, a distinction is made between byte and short, but Oracle's implementation of the JCBCV permits these implicit castings in these cases, which indeed, do

not cause potential security vulnerability. In [15] a subset of these tests for only twelve instructions were executed on five Java Cards and most of the tests failed; one fault was exploitable. Oracle's JCBCV is typically not used on Java Cards, because it is too big to be embedded in a smart card. It is used for offline verification before loading bytecode programs on a card.

7 Related Work

MBT has been extensively studied (*e.g.*, [3,4,11,16,17]) for extended state machines and automata. These notations use a more basic type system, compared to (so called) model-based notation like Z, B, ASM, and Alloy, and thus use different test criteria than ours. MBT has been applied to model-based notation on a lesser extent, and mostly for functional testing [18]. Our work seems to be the first to address robustness testing using mutation by negation. As already discussed in the introduction, functional MBT cannot cater for robustness testing, unless invalid cases are modelled in detail, which requires significantly more resources to build formal specifications. In [14], ProB is used to generate functional tests using B specifications; no data coverage criteria are used; structural coverage is less specific than the one used in this paper. In [6], mutations targeting typical syntactical errors are applied to ASM specifications, in order to derive robustness tests. However, syntactical errors may generate both valid and invalid tests, when the mutation does not affect the valid behaviour. In [8], Alloy is used for applying classical functional MBT techniques and test criteria for Java programs. Functional MBT has also been applied to Circus for data-flow coverage [5] using specification traces.

Mutation testing has been extensively studied for programs (*e.g.*, [2,7,9]), in order to evaluate the quality of test suites. Mutation rules are designed for basic programming data structures. Mutation testing for extended timed-automata is used in [3] to detect faults in a car alarm. Mutation rules are guided by typical modelling errors on automata transitions and guards, including simple negation.

8 Conclusion

Robustness testing aims at finding errors in a system under invalid conditions, such as unexpected inputs. We have proposed a robustness testing approach for Event-B based on specification mutation using guard negation and model-based testing using ProB. Data coverage criteria are described by the negation rules and structural coverage criteria are driven by the constraint-based checking of ProB, which was optimized to rapidly exclude unfeasible mutants. These enhancements allow our approach to scale up to large Event-B specifications containing hundreds of events. The approach has been applied to type checking of Java Card programs for a subset of 61 bytecode instructions generating more than 900 mutants.

We plan to extend our Event-B model of the Java Card language to deal with branching instructions and object field access. We are also currently working

on the negation of Event-B contexts, which will allow us to generate, using ProB, complex class file structures to test structural verifications. Scaling up is also an issue in this problem, since the Java Card specification contains a large number of constraints on class files with complex interrelated data structures.

One lesson learned in this case study is that modelling for robustness testing is different than modelling for proving the correctness of an implementation. On the one hand, some invariants must be relaxed in order to cover invalid cases and guards must be manually tagged to identify what should be negated. On the other hand, a robustness specification can be simpler to build than a complete functional specification for implementation correctness proof.

References

1. Abrial, J.: Modeling in Event-B. Cambridge University Press (2010)
2. Agrawal, *et al.*: Design of Mutant Operators for the C Programming Language. Tech. rep., Software Engineering Research Center, Purdue University (1989)
3. Aichernig, B.K., Lorber, F.: Model-based Mutation Testing with Timed Automata. Technical Report IST-MBT-2013-02, TU Graz pp. 1–21 (2013)
4. Bouquet, F.*et al.*: A subset of precise UML for model-based testing. In: 3rd international workshop on Advances in model-based testing. pp. 95–104. ACM (2007)
5. Cavalcanti, A., Gaudel, M.: Data flow coverage for circus-based testing. In: Fund. App. to Soft. Eng., FASE. LNCS, vol. 8411, pp. 415–429. Springer (2014)
6. Gargantini, A.: Using model checking to generate fault detecting tests. In: Tests and Proofs. LNCS, vol. 4454, pp. 189–206. Springer (2007)
7. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
8. Khurshid, S., Marinov, D.: TestEra: Specification-based testing of java programs using SAT. Automated Software Engineering 11(4), 403–434 (2004)
9. Kim, S., Clark, J., McDermid, J.: The Rigorous Generation of Java Mutation Operators Using HAZOP. Tech. rep., University of York (1999)
10. Leuschel, M., Butler, M.: Prob: an automated analysis toolset for the b method. Int. J. on Software Tools for Technology Transfer 10(2), 185–203 (2008)
11. Mikucionis, M., Larsen, K.G., Nielsen, B.: T-UPPAAL: Online model-based testing of real-time systems. In: 19^{th} Aut. Soft. Eng. (ASE2004). pp. 396–397 (2004)
12. Oracle Corporation: Java Card 3 Platform Virtual Machine Specification. `http://www.oracle.com/` (2011)
13. Oracle Corporation: Java Card SDK. `http://www.oracle.com/` (2014)
14. Satpathy, M., Butler, M., Leuschel, M., Ramesh, S.: Automatic testing from formal specifications. In: Tests and Proofs. LNCS, vol. 4454, pp. 95–113. Springer (2007)
15. Savary, A., Frappier, M., Lanet, J.L.: Detecting vulnerabilities in Java Card bytecode verifiers using model-based testing. In: Integrated Formal Methods (IFM2013). vol. 7940 LNCS, pp. 223–237 (2013)
16. Shafique, M., Labiche, Y.: A systematic review of state-based test tools. Int. J. on Software Tools for Technology Transfer 17(1), 59–76 (2015)
17. Utting, M., Legeard, B.: Practical Model Based Testing: A Tools Approach. Kaufmann, Morgan (2007)
18. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing Verification and Reliability 22(5), 297–312 (2012)