Test Data Generation for Event-B Models using Genetic Algorithms

Ionut Dinca, Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose

University of Pitesti, Department of Computer Science
Str. Targu din Vale 1, 110040 Pitesti, Romania
{name.surname}@upit.ro

Abstract. Event-B is a formal modeling language having set theory as its mathematical foundation and abstract state machines as its behavioral specifications. The language has very good tool support based on theorem proving and model checking technologies, but very little support for test generation. Motivated by industrial interest in the latter domain, this paper presents an approach based on genetic algorithms that generates test data for Event-B test paths. For that, new fitness functions adapted to the set-theoretic nature of Event-B are devised. The approach was implemented and its efficiency was proven on a carefully designed benchmark using statistically sound evaluations.

1 Introduction

Event-B [1] is a modeling language used for formal system specification and analysis. Event-B was introduced about ten years ago and it quickly attracted the attention of both academic and industrial researchers. The theoretical foundations and associated tooling were developed in several research projects, among which the most notable are two large European research projects: RODIN[1], which produced a first platform for Event-B called *Rodin*, and DEPLOY[2], which is currently enhancing this platform based on industrial feedback. Theorem-proving is the core technology within Rodin, but model-checking tools have also been developed (ProB [9]). Recently, there has been an increasing interest from the industrial partners like SAP (who belongs to DEPLOY consortium) for test generation based on Event-B models [19]. This provided the main motivation for our investigations into model-based testing using Event-B models, especially test data generation.

Model-based testing (MBT) is an approach that uses formal models as basis for automatic generation of test cases [18]. For MBT using state-based models, test generation algorithms usually traverse the state space from the initial state, guided by a certain coverage criterion (e.g. state coverage), collecting the execution paths in a test suite. Event-B models do not have an explicit state

[1] *http://rodin.cs.ncl.ac.uk* - Project running between 2004-2007
[2] *http://deploy-project.eu* - Project running between 2008-2012

space; instead, their state spaces are given by the value of the variables. The ProB tool [9], which is available in the Rodin platform, has a good control of the state space, being able to explore it, visualize it and verify various properties using model-checking algorithms. Such algorithms can be used to explore the state space of Event-B models using certain coverage criteria (e.g. event coverage) and thus generating test cases along the traversal. Moreover, the input data that trigger the events provides the test data associated with the test cases. Such an approach using explicit model-checking has been applied to models from the business application area by SAP [19]. The algorithms perform well for models with data with a small finite range. However, in case of variables with a large range (e.g. integers), the known state space explosion problem creates difficulties, since the model checker explores the state space by enumerating the many possible values of the variables.

This paper addresses a slightly different, but related, problem. Given a (potentially feasible) path in the Event-B model, we use meta-heuristic search algorithms (more precisely, genetic algorithms) to generate input data that trigger the execution of the path. This is a very important issue of MBT since, for models with large state spaces, paths with restrictive triggering conditions (e.g. composed conditions involving one or more = operators) are difficult to attain using the model checking approach described above. A similar problem has been addressed by recent work on search-based testing for Extended Finite State Machines (EFSMs) [8, 5, 20]. However, there are a number of issues that differentiate search-based testing on Event-B models from these EFSM approaches as described our position paper [16] like implicit state space, non-numerical types, non-determinism and hierarchical models. In this paper, we start addressing some of these issues, especially the non-numerical types.

The main contributions of the paper are enumerated below:

- Since the data structures used by Event-B models are predominantly set-based rather than numerical, Tracey-like [17] fitness functions for such data types are newly defined. These fitness functions are used to guide the search for the solutions in large state spaces.
- Furthermore, the encoding of non-numerical types into a chromosome is investigated. As Event-B models may use a mixture of numerical and non-numerical types, the encoding has to accommodate also such a possibility.
- The proposed search-based testing approach for Event-B is applied on a number of industry-inspired case studies. The experiments show that the approach performs better in general compared to random testing approaches.

The paper is structured as follows. We start by describing the Event-B framework in Section 2 together with a couple of representative Event-B examples in Section 3. Then we present the proposed test generation framework based on search-based techniques using genetic algorithms in Section 4. The experiments are explained in Section 5 and the conclusions are drawn in Section 6.

2 Formal modeling with Event-B

Event-B [1] is a formal language based on the notion of abstract machines having set theory as its mathematical foundation. The two basic constructs in Event-B are *contexts* and *machines*. The contexts specify the static part of a model and contain the following elements: *carrier sets* (i.e. domains), *constants* and *axioms*. The axioms can define relations between the constants or define their domains. Machines represent the dynamic part of a model and can contain *variables, invariants* and *events*.

An event is composed of two main elements: *guards* which are predicates that describe the conditions that must hold for the occurrence of an event and *actions* which determine how specific variables change as a result of the event execution. An event has the following general form:

$$\text{Event} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } G(t,x) \textbf{ then } S(x,t) \textbf{ end}.$$

Above, t is a set of local parameters, x is a set of global variables appearing in the event, G is a predicate over t and x, called the guard and $S(x,t)$ represents a substitution. If the guard of an event is false, the event cannot occur and is called disabled. The substitution S modifies the values of the global variables in the set x. It can use the old values from x and the parameters from t.

For example, an event that takes a natural number parameter *value* smaller than 50 and adds it to the natural (global) variable *balance* only if *balance* is larger than 1500, can be modeled as:

$$\text{Event1} \mathrel{\widehat{=}} \textbf{any } \mathit{value} \textbf{ where } \mathit{value} \in \mathbb{N} \ \wedge\ \mathit{value} < 50 \ \wedge\ \mathit{balance} > 1500 \textbf{ then } \\ \mathit{balance} := \mathit{balance} + \mathit{value} \textbf{ end}.$$

Note that if the model has 10 integer variables with their range in [1..10,000], then the explicit state space would have 10^{40} states, which is usually too much for a brute-force traversal algorithm of an explicit model checker. In this paper, we use meta-heuristic search techniques to deal with such large state spaces.

Let us consider another example, involving a set defined as *ITEMS*=$\{$*it1, it2, ... , it20*$\}$ and an event that modifies a set variable *items*, where $\mathit{items} \subseteq \mathit{ITEMS}$ (alternatively, we can write that *items* is an element of the powerset of *ITEMS*, i.e. $\mathit{items} \in \mathbb{P}(\mathit{ITEMS})$). We can model a situation in which the value of one global variable *oneItem* is randomly picked from the set *items* (using the Event-B operator $:\in$) and the set *items* is updated with elements from a parameter *buffer* of cardinality smaller than 5 (the cardinality is denoted by *card*()):

$$\text{Event2} \mathrel{\widehat{=}} \textbf{any } \mathit{buffer} \textbf{ where } \mathit{buffer} \in \mathbb{P}(\mathit{ITEMS}) \ \wedge\ \mathrm{card}(\mathit{buffer}) < 5 \textbf{ then} \\ \mathit{oneItem} :\in \mathit{items} \ \wedge\ \mathit{items} := \mathit{items} \cup \mathit{buffer} \textbf{ end}.$$

Thus, an Event-B model is given by the defined domains, constants, variables, events that change the global variables when executed, and a set of global invariants specifying the properties required by the specification. The execution of a model starts with a special event that initializes the system, followed by the application of enabled events. At each execution step, all the guards of the events are evaluated and the set of enabled events is computed. Then, one enabled event is non-deterministically chosen and its action is executed. The Rodin platform (*http://www.event-b.org/platform.html*), built on top of Eclipse, provides different plugins that manage and perform different tasks on the Event-B models.

3 Case studies

In Section 5 we run the experiments on a benchmark of 5 Event-B models. The models are not industrial ones, but are inspired by industrial examples. We have been in contact with partners in the DEPLOY project that are interested in test generation from Event-B models, especially SAP, which is an industrial partner from the business software area. We have discussed a couple of MBT requirements together with a couple of sample models. For the benchmark, we made model variations such that we cover different guard and variable types.

We describe 2 out of the 5 Event-B models that we used for the benchmarks. The events of the first one contain numerical parameters, while the events of the second model focus on set parameters. The presentation of each model starts with a short description, followed by the types of the global variables (defined in the context of the Event-B model). Then, the events of the Event-B machines are listed together with their parameters. The guards and actions associated to each event are presented in a separate table.

Numerical-based model: Bank Account. The first example models a simple bank account system. The system allows the user to deposit money in the account or to withdraw money from it. The bank pays interest and charges fees. Depending on the current balance, a deposit can be in four states: overdraft, empty, silver and gold. Thus, the Event-B variables are: $balance \in \mathbb{Z}$, $transaction \in BOOL$ and $state \in STATES{=}\{overdraft, empty, silver, gold\}$. The machine events, whose guards and actions are given in Table 1, are the following:

E1. *Initialization*, that initializes the bank account
E2. *Deposit*, having the numerical parameters $amount1$ and $amount2$
E3. *Withdraw*, having the numerical parameters $amount1$ and $amount2$
E4. *ValidateOverdraft*
E5. *ValidateEmpty*
E6. *ValidateSilver*
E7. *ValidateGold*
E8. *PayInterest*, having the numerical parameter *value*
E9. *ChargeFee*, having the numerical parameter *fee*.

Set-based model: Basket of Items. Here we model a basket of items. The system allows the user to add items, to remove items and to pick items from the basket. The system checks if the basket is empty or full or can make a special check. The global variables are: $items \in \mathbb{P}(ITEMS)$, $buffer \in \mathbb{P}(ITEMS)$, $isEmpty \in BOOL$, $isFull \in BOOL$, $CAPACITY \in \mathbb{N}$, and $count \in \mathbb{N}$ with the invariants $count \geq 0 \wedge count \leq CAPACITY$ and $count = \text{card}(items)$, where

Table 1. Guards and actions of Bank Account events

Ev	Guards	Actions
E1:	$TRUE$	$balance := 0$, $state := empty$ $transaction := FALSE$
E2:	$amount1 + amount2 > 200 \wedge$ $amount1 \in \mathbb{N} \wedge amount2 \in \mathbb{N}$	$balance := balance + amount1 + amount2$ $transaction := TRUE$
E3:	$balance > 0 \wedge amount1 + amount2 < 1000 \wedge$ $balance - amount1 - amount2 > -100 \wedge$ $amount1 \in \mathbb{N} \wedge amount2 \in \mathbb{N}$	$balance := balance - amount1 - amount2$ $transaction := TRUE$
E4:	$balance < 0 \wedge balance > -100$	$state := overdraft$
E5:	$balance = 0$	$state := empty$
E6:	$balance > 0 \wedge balance < 1000$	$state := silver$
E7:	$balance \geq 1000$	$state := gold$
E8:	$balance > 1500 \wedge$ $value \leq 50 \wedge value > 0 \wedge value \in \mathbb{N}$	$balance := balance + value$
E9:	$fee > 0 \wedge fee < 50 \wedge fee \in \mathbb{N} \wedge$ $transaction = TRUE$	$balance := balance - fee$ $transaction := FALSE$

$ITEMS = \{it1, it2, \ldots, it20\}$. The Event-B events, whose guards and actions are given in Table 2, are the following:

E1. *Initialization*, that initializes the basket of items
E2. *PickItems*, with the set parameter *its*
E3. *AddItems*
E4. *RemoveItems*
E5. *ValidateEmpty*
E6. *CheckSpecial*
E7. *ValidateFull*.

4 Test data generation for Event-B models using genetic algorithms

Before describing our test generation approach, let us establish the problem to be solved. First, let us note that Event-B specifications are event-based rather than state-based models. Formally, these are abstract state machines [4] in which the (implicit) states are given by the (global) values of the *variables* on which the events operate. Each event is given by a triplet consisting of (1) the *parameters* (local variables) used by the event, (2) the guards which constrain the event application (the guards may involve both local and global variables) and (3) the actions of event, which may change the values of the global variables. The events produce the transitions between states: the guards establish the valid source state(s) of the transition while the actions produce the target state(s). In

Table 2. Guards and actions of Basket of Items events

Ev	**Guards**	**Actions**
E1:	TRUE	$\mathit{items} := \varnothing$, $\mathit{buffer} := \varnothing$, $\mathit{count} := 0$ $\mathit{CAPACITY} := \mathrm{card}(\mathit{ITEMS})$ $\mathit{isEmpty} := \mathit{TRUE}$, $\mathit{isFull} := \mathit{FALSE}$
E2:	$\mathit{its} \subseteq \mathit{ITEMS}$	$\mathit{buffer} := \mathit{its}$
E3:	$\mathit{buffer} \subseteq \mathit{ITEMS} \wedge \mathrm{card}(\mathit{buffer}) > 5 \wedge$ $\mathrm{card}(\mathit{buffer}) + \mathit{count} \leq \mathit{CAPACITY}$	$\mathit{items} := \mathit{items} \cup \mathit{buffer}$ $\mathit{count} := \mathrm{card}(\mathit{items} \cup \mathit{buffer})$ $\mathit{isEmpty} := \mathit{FALSE}$
E4:	$\mathit{buffer} \subseteq \mathit{items} \wedge \mathrm{card}(\mathit{buffer}) > 3 \wedge$ $\mathit{count} - \mathrm{card}(\mathit{buffer}) \geq 0$	$\mathit{items} := \mathit{items} \setminus \mathit{buffer}$ $\mathit{count} := \mathrm{card}(\mathit{items} \setminus \mathit{buffer})$
E5:	$\mathit{items} = \varnothing \wedge \mathit{count} = 0$	$\mathit{isEmpty} := \mathit{TRUE}$
E6:	$\{\mathit{it}1, \mathit{it}20\} \subseteq \mathit{items} \wedge \mathrm{card}(\mathit{items}) < 6$	$\mathit{buffer}{:=}\mathit{items}$
E7:	$\mathit{count} = \mathit{CAPACITY}$	$\mathit{isFull} := \mathit{TRUE}$ $\mathit{items} := \varnothing$, $\mathit{count} := 0$

general, the application of an event depends on the values of the parameters it receives. If we want to execute a path (sequence of events) through the model, we will need to find appropriate parameter values for each event in the sequence (i.e. which satisfy the corresponding guards). This is the problem we will solve using a genetic algorithm. Naturally, the prerequisite is that a set of paths, which satisfies the given test requirement has already been found.

In general, this requirement is expressed as a level of coverage of the model. Various levels of coverage for finite state machines exist in the literature [3, 18] and some can be adapted to Event-B models without the need to transform the model into an explicit state machine (for large systems this transformation may be impractical). For example, transition coverage for a finite state machine requires every transition to be triggered at least once. Similarly, for Event-B models, event coverage will involve the execution of every event at least once. This type of coverage can be generalized by requiring that each feasible sequences of events of a given length k is executed at least once. Obviously, in order to decide if a path is feasible or not it may be necessary to effectively find test data (parameter values) which triggers it. Consequently, the potentially feasible paths can be selected first by deleting paths which contain obvious contradictory constraints (e.g. both C and $\neg C$) and then the test data generation algorithm is applied to each such path. Other types of coverage may also be defined but this beyond the scope of this paper.

In this paper, we assume that we have a set of paths (that cover, for instance, all events of the model). For each path of the given set, we seek appropriate test data, i.e. event parameters which enable the events in the path. It may be possible that the test data for the selected path has not been found, either because of the complexity of the guard constraints or simply because the path is infeasible;

if this is the case, a new path is selected. Note that the paper does not address the issue of path selection, but only the test generation for the chosen path(s).

Below we present the theoretical instruments based on genetic algorithms for the above problem. First, Subsection 4.1 provides the background on genetic algorithms. Then, the Subsections 4.2 and 4.3 describe the main ingredients of the approach, i.e. the encoding of the sought solutions into chromosomes and the fitness function that guides the search into the solution space, respectively.

Note that among the different meta-heuristic algorithms, for convenience, in this paper we have chosen to use the class of genetic algorithms [13], because they are widely used in search-based testing approaches and have good tooling support. However, we plan in the future to experiment with other types of algorithms like simulated annealing or particle swarm optimization.

4.1 Genetic algorithms

Genetic algorithms (GAs) [13] are a particular class of *evolutionary algorithms*, that use techniques inspired from biology, such as selection, recombination (crossover) and mutation. GAs are used for problems which cannot be solved using traditional techniques and for which an exhaustive search of the solution space is impractical. In particular, the application of GAs to the difficult problem of test data generation recently received an increased attention from the testing research community [11, 10].

GAs basic approach is to encode a population of potential solutions on some data structures, called *chromosomes* (or *individuals*) and applying recombination and mutation operators to these structures. A high-level description of a genetic algorithm [10, 13] is given in Fig. 1. The *fitness (or objective) function* assigns a score (called fitness) to each chromosome in the current population. The fitness of a chromosome depends on how close that chromosome is to the solution of the problem. Throughout this paper, the fitness is considered to be positive and finding a solution corresponds to minimizing the fitness function, i.e. a solution will be a chromosome with fitness 0. The algorithm terminates when some stopping criterion has been met, for example when a solution is found, or when the number of generations has reached the maximum allowed limit.

Various mechanisms for selecting the individuals to be used to create offspring, based on their fitness, have been devised [6]. GA researchers have experimented with mechanisms such as sigma scaling, elitism, Boltzmann selection, tournament, rank and steady-state selection [13].

After the selection step, recombination takes place to form the next generation from parents and offspring. The mutation operator is then applied. These two operations, crossover and mutation, depend on the type of encoding used and so they are discussed in more detail in the next subsection.

4.2 Chromosome encodings

Consider a path $event_1 \ldots event_n$ in the Event-B model. A *chromosome* (possible solution) is a list of values, $x = (x_1, \ldots, x_m)$ for the event parameters of the

```
Randomly generate or seed initial population $P$
Repeat
    Evaluate fitness of each individual in $P$
    Select $P'$ from $P$ according to selection mechanism
    Recombine parents from $P'$ to form new offspring
    Construct $P'$ from parents and offspring
    Mutate $P'$
    $P \leftarrow P'$
Until Stopping Condition Reached
```

Fig. 1. Genetic Algorithm

path events (in the order they appear). More formally, if $p_{i1}, \dots, p_{ik_i}$ are the parameters of $event_i$,$1 \leq i \leq n$, then x represents a list of values for parameters $p_{11} \dots p_{nk_n}$. Naturally, $m = k_1 + \dots + k_n$ can differ from the number n of events in the sequence. If the values x satisfy all guards and, consequently, trigger the path, then x is a solution for the given path. For numerical data, the chromosomes are integer-encoded, each gene representing one parameter.

Consider, for example, the path $E2$ $E8$ $E9$ $E3$ $E7$ from the Bank Account example presented earlier (technically, any path of a model starts with the special event *Initialization* ($E1$), but for simplicity when we mention the events of a path we skip $E1$). There are five events in the path: $E2$ (*Deposit*), which receives *amount*1 and *amount*2 as parameters, $E8$ (*PayInterest*), with parameter *value*, $E9$ (*ChargeFee*), with parameter *fee*, $E3$ (*Withdraw*), with parameters *amount*1 and *amount*2 and $E7$ (*ValidateGold*), with no parameters. Since all 6 parameters have numerical types, a chromosome for the above path will be a list of 6 integers.

An additional problem occurs when non-numerical types are involved since such values will have to be encoded into the chromosome. The applications we have considered use enumeration types as well as types derived from these using traditional set operators ($\cup, \setminus, \times$). For a k-valued type $T = \{v_1, \dots, v_k\}$, a set parameter S which is a subset of T, i.e. $S \subseteq T$, is represented by a bitmap of length k, which has 1 on the ith position in the bitmap if $v_i \in S$, and 0 otherwise. Then, a chromosome corresponding to parameters $p_1 \dots p_m$ will be a list of values $x_1 \dots x_m$, in which each value is encoded as appropriate. The applications we have considered use both numerical and non-numerical types and so some values in the chromosome are represented by simple integers whereas other values are encoded as bitmaps. Once we generated a population of chromosomes, the operations of crossover and mutation are applied as described below.

Crossover. For mixed chromosomes (with both binary and integer genes) and binary-only chromosomes, a *single-point crossover* is used. This randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two new offspring. For example, the strings 00000000 and 11111111 could be crossed over at the third locus to produce the two offspring 00011111 and 11100000. The crossover is applied to individuals

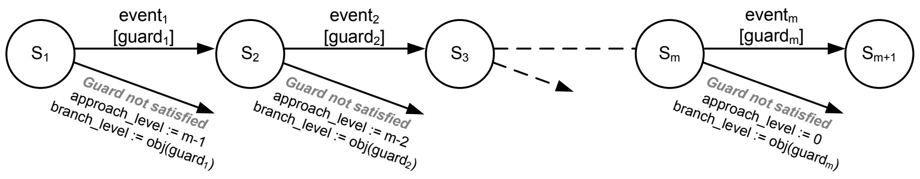

Fig. 2. Calculating the fitness function

selected at random, with a probability (rate) p_c. Depending on this rate, the next generation will contain the parents or the offspring.

For integer chromosomes, we used a *heuristic real value crossover*, inspired from [12], that showed to be the most efficient type of crossover for our problem. This uses the fitness of the individual for determining the direction of the search. For parents $x = (x_1, \dots, x_m)$, $y = (y_1, \dots, y_m)$, x fitter than y, one offspring $z = (z_1, ..., z_m)$ is generated, with z_i being the integer-rounded value of $\alpha \cdot (x_i - y_i) + x_i$, $\alpha \in (0, 1)$. Heuristic real value and single-point crossovers can be combined.

Mutation is used to introduce variation in the population by randomly changing genes in the chromosome. Mutation can occur at each bit position in a string with some probability p_m, usually very small [13]. For binary genes, the mutation operator randomly flips some bits in a chromosome. For example, the string 00000100 could be mutated in its second position to yield 01000100. For integer genes, the gene value is replaced by another integer value that is randomly chosen from the same interval.

4.3 Fitness function

The algorithm evaluates a candidate solution by executing each event with the values encoded in the chromosome's genes until the guard of the current event is not satisfied. The fitter individuals are the ones which enable more events from the given path. They are rewarded with a lower fitness value. The fitness function is calculated using a formula widely used in the search-based testing literature [11, 8], using two components. The first evaluates how close a chromosome is to executing the given path, by counting the events executed. The second measures how close is the first unsatisfied guard predicate to being true.

$$\mathit{fitness} := \mathit{approach\_level} + \mathit{normalized\_branch\_level}$$

The first component, *approach (approximation) level* is similar a metric in evolutionary structural test data generation [11]. This is calculated by $m - 1 - n$, where m is the length of the path to be executed and n is the number of events successfully executed until the first unsatisfied guard on the path, as in Fig. 2.

A fitness function formed only from the approach level has many plateaux (i.e. for each value $0, 1, \dots, m-1$) and it would not offer enough guidance to the search. Consequently, the second component, called *branch level*, was introduced.

Table 3. Tracey's objective functions for relational predicates and logical connectives. The value K, $K > 0$, refers to a constant which is always added if the term is not true.

Relational predicate or logical connective	Objective function *obj*
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
Boolean	if $TRUE$ then 0 else K
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
a xor b	$obj((a \wedge \neg b) \vee (\neg a \wedge b))$
$\neg a$	Negation is moved inwards and propagated over a

This computes, for the place where the actual path diverges from the required one, how close was the guard predicate to being true.

For numeric types, the *branch level* can be derived from the guards predicates using Tracey's objective functions as shown in Table 3 [17, 10]. The *branch level* is then mapped onto the interval [0,1) or normalized.

We extended the calculation of the *branch level* to applications which involve set theory based constraints as described below. The applications considered use basic types that can be mapped onto either an interval ($[p..q]$, $0 \leq p < q$,) or an enumeration of non-negative integers ($\{p_1, \ldots, p_n\}$, $n \geq 1$, $p_i \geq 0$, $1 \leq i \leq n$). Furthermore, the derived types use the $\cup$, $\cap$, $\setminus$ and $\times$ set operators. Then the objective function for the $a \in A$ and $a \notin A$ predicates can be derived using the transformations given at the top of Table 4. The formulae are then extended for the $\subseteq$ and $=$ set operators, as shown at the bottom of Table 4.

5 Experiments

We have implemented our approach as a plugin for the Eclipse-based Rodin platform for Event-B. The plugin is designed to automatically generate test data for given paths in the Event-B model. It can generate test data, i.e. the input parameters for the events on the given path, employing the fitness function described in Section 4.3. The execution of the events (including the initialization) was performed using the Event-B model simulation of the ProB plugin [9].

For the benchmark of 5 models mentioned in Section 3, we have considered a set of 18 random paths likely to be feasible, which covered all the events from the models. The paths length varied between 2 and 5 events (without counting the initialization event). The number of parameters on each path varied between: (a) 1 and 7 for numerical models; (b) 1 and 2 non-numerical parameters, such

Table 4. The extension of Tracey's objective functions to set operators

Predicate involving $\in$ for basic or derived sets	Objective function obj
$a \in [p, q]$	$obj((a \geq p) \wedge (a \leq q))$
$a \notin [p, q]$	$obj((a < p) \vee (a > q))$
$a \in \{p_1, \ldots, p_n\}$	$obj((a = p_1) \vee \ldots \vee (a = p_n))$
$a \notin \{p_1, \ldots, p_n\}$	$obj((a \neq p_1) \wedge \ldots \wedge (a \neq p_n))$
$a \in A \cup B$	$obj((a \in A) \vee (a \in B))$
$a \in A \cap B$	$obj((a \in A) \wedge (a \in B))$
$a \in A \setminus B$	$obj((a \in A) \wedge (a \notin B))$
$(a, b) \in (A, B)$	$obj((a \in A) \wedge (b \in B))$
Predicates for $\subseteq$ and $=$ operators	**Objective function obj**
$[p, q] \subseteq A$	$obj(\wedge_{i=p}^{q}(i \in A))$
$\{p_1, \ldots, p_n\} \subseteq A$	$obj((p_1 \in A) \wedge \ldots \wedge (p_n \in A))$
$[p, q] \nsubseteq A$	$obj(\vee_{i=p}^{q}(i \notin A))$
$\{p_1, \ldots, p_n\} \nsubseteq A$	$obj((p_1 \notin A) \vee \ldots \vee (p_n \notin A))$
$A = B$	$obj((A \subseteq B) \wedge (B \subseteq A))$
$A \neq B$	$obj((A \nsubseteq B) \vee (B \nsubseteq A))$

as $x \in \mathbb{P}(ITEMS)$ for set examples; (c) $2-4$ set parameters and $2-3$ numerical parameters for the mixed model. The codification used was: integer-valued for numerical parameters (the integer range was fixed to 2000) and bitmap for set parameters.

As recommended in [2], a search algorithm (GA in this case) should be compared with random search in order to check that the algorithm is not simply successful because the search problem is easy. Therefore, we tried to generate test data for the 18 selected paths mentioned above, denoted by $P1 - P18$, using the two methods: *search-based testing with genetic algorithms (GA)* and *random testing (RT)*. For each path and each test generation method, 30 runs were performed (this number was also recommended in [2]). A run is considered *successful* if it can produce input test data that can trigger the whole path, or equivalently, the fitness function associated has the value 0. The run ends when a solution was found or when the maximum number of generations was reached.

Using genetic algorithms, the amount of time needed to obtain test data for a path, i.e. the actual values of the parameters which trigger the path, varied between 1 second (for very simple paths, where the solution could be found from the first generation) and 60 seconds (for complex paths).

The genetic algorithm framework used for experimentation was the open source Java Genetic Algorithms Package (JGAP) [7]. The maximum number of generations for the genetic algorithm was set to 100 and the population size to 20. The selection operator employed was *BestChromosomesSelector*, an elitist operator, the mutation rate was $p_m = 1/10$ and the crossover was single-point (for non-numerical parameters) or heuristic crossover (for numerical ones), as presented in Section 4.2.

For random testing, the same library was used: instead of applying recombination or mutation, the population was randomly generated at each step, ensuring this way an equal treatment, i.e. an equal number of generations (or fitness function evaluations) for both methods, GA and RT. For each run, the generation when the solution was found was recorded and Table 5 presents the summarizing data: the success rate for each method (percent of successful runs from the 30 ones considered) and other descriptive statistics, e.g. the average (mean) number of generations, the median and the standard deviation.

Statistical tests should be realized to support the comparison of GA and RT runs. In our experiments we have used two statistical tests: the parametric t-test and the non-parametric Mann-Whitney U-test. The null hypothesis (H_0) is thus formulated as follows: *There is no difference in efficiency (the number of generations needed to find a solution) between GA and RT.* The alternative hypothesis (H_a) follows: there is a difference between the two approaches, GA and RT. The two tests measure different aspects: the t-test measures the difference in mean values (the null hypothesis is $H_0 : \mu_1 = \mu_2$), whereas the Mann-Whitney U-test measures their difference in medians ($H_0' : \theta_1 = \theta_2$), i.e. whether the observations in one data sample are more likely to be larger than observations in the other sample.

The test results and the p-values obtained are given in Table 5. In the columns t-test and U-test, the sign '+' stands for rejecting the null hypothesis (consequently, there is a statistically significant difference between GA and RT results), while the '$-$' indicates that the null hypothesis cannot be rejected at the significance level considered, $\alpha = 0.01$. The p-value computed by the statistical test is also provided, excepting the case when it can not be computed, e.g. when both approaches were able to find a solution from the first generation for all the runs (paths $P16, P17$), where '†' stands for not computed.

Some standardized effect size measures were also used and they are given in the last two columns: the Vargha and Delaney's A statistic, the Cohen's D coefficient. The Vargha and Delaney's A statistic [2] is a performance measure, used to quantify the probability that GA yield 'better values' than RT. In our case, 'better values' means lower number of generations needed to obtain a solution.

The Vargha and Delaney's statistics is given in the column 'A'. For simple paths, where RT and GA provide the solution in the same number of generations, the effect size is 0.5. For more complex paths, a value of 0.82 means that we would obtain better results in 82% of the time with GA (they guide the search to success in a lower number of generations). It is worth noting that *GA clearly outperformed RT for* 14 *out of* 18 *paths considered*, and the difference in terms of success rate, average (or median) number of generations was significant.

The last column of Table 5 presents the Cohen's D coefficient, which is computed as the absolute difference between two means, divided by a pooled standard deviation of the data [2, 15]. According to [15], Cohen has proposed the following 'D values' as criteria for identifying the magnitude of an effect size:

Table 5. Success rates, results of the statistical tests and effect size measures

Path	**Meth.**	**Success rate**	**Avg. gen.**	**Median**	**Std. dev.**	**t-test p-val**	**U-test p-val**	**A**	**D**
P1	GA	100.0%	18.2	12.0	18.8	+	+	1.00	5.85
P1	RT	3.3%	99.0	100.0	5.3	< 0.001	< 0.001		
P2	GA	100.0%	14.9	11.0	12.5	+	+	1.00	6.45
P2	RT	3.3%	97.6	100.0	13.1	< 0.001	< 0.001		
P3	GA	96.7%	22.7	14.5	19.8	+	+	0.98	4.61
P3	RT	10.0%	96.9	100.0	11.1	< 0.001	< 0.001		
P4	GA	100.0%	12.4	8.0	11.6	+	+	0.82	1.17
P4	RT	96.7%	35.0	32.5	24.8	< 0.001	< 0.001		
P5	GA	66.7%	53.3	44.5	38.7	+	+	0.83	1.71
P5	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P6	GA	100.0%	23.5	21.0	9.2	+	+	1.00	11.73
P6	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P7	GA	100.0%	12.7	12.0	4.5	+	+	1.00	16.72
P7	RT	6.7%	98.5	100.0	5.7	< 0.001	< 0.001		
P8	GA	100.0%	16.9	17.0	4.4	+	+	1.00	26.59
P8	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P9	GA	100.0%	13.7	13.0	2.4	+	+	1.00	12.46
P9	RT	3.3%	98.3	100.0	9.3	< 0.001	< 0.001		
P10	GA	100.0%	30.9	31.5	6.3	+	+	1.00	15.51
P10	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P11	GA	96.7%	20.0	13.0	22.6	+	+	0.98	4.64
P11	RT	3.3%	98.6	100.0	7.9	< 0.001	< 0.001		
P12	GA	100.0%	13.5	13.0	2.8	+	+	1.00	43.66
P12	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P13	GA	100.0%	11.9	11.5	2.2	+	+	1.00	56.56
P13	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P14	GA	100.0%	1.3	1.0	1.6	−	−	0.47	0.29
P14	RT	100.0%	1.0	1.0	0.0	0.28	0.49		
P15	GA	100.0%	1.0	1.0	0.0	−	−	0.50	†
P15	RT	100.0%	1.0	1.0	0.0	†	1.00		
P16	GA	100.0%	1.0	1.0	0.0	−	−	0.50	†
P16	RT	100.0%	1.0	1.0	0.0	†	1.00		
P17	GA	90.0%	16.9	5.5	29.9	+	+	0.91	1.79
P17	RT	53.3%	72.2	83.5	31.7	< 0.001	< 0.001		
P18	GA	100.0%	1.8	1.0	2.4	−	−	0.53	0.17
P18	RT	100.0%	1.5	1.0	0.8	0.53	0.63		

a) small effect size: $D \in (0.2, 0.5)$, b) medium effect size $D \in [0.5, 0.8)$, c) large effect size $D \in [0.8, \infty)$. According to this classification, it can be easily noticed that the *difference between the results obtained with GA versus RT correspond for most paths (*14 *out of* 18*) to a large effect size.*

6 Final discussion

Bottom line. In this paper, we have presented an approach based on genetic algorithms that allows generating test data for event paths in the Event-B framework. One distinguishing feature of Event-B is its set-theoretic foundation, meaning that in Event-B models, numerical variables are used together with non-numerical types based on sets. To address this, we extended the fitness functions available in the search-based testing literature to set types. Moreover, the encoding of the sought solutions included mixed chromosomes containing both numerical and non-numerical types. Finally, we followed standard statistical guidelines [2] to demonstrate the efficiency and effectiveness of our implementation on a diversified benchmark inspired by discussions with the industry.

Related work. The only approach of test generation for Event-B models is based on explicit model-checking [19] with ProB [9], which suffers from the classical state space explosion problem. There is also related work on applying search-based techniques to EFSMs [8, 5, 20]. Differently from these, we address a different modeling language and tackle non-numerical types. However, we can certainly extend our work with ideas from these papers, e.g. regarding feasible path generation, or from previous work on test generation from B models (the precursor of Event-B language, even though B is not an event-based language) [14], [18, ch.3].

Future work. Since the goal is to develop a test method that scales for industrial Event-B models, we have performed a survey of 29 publicly available Event-B models posted by the DEPLOY academic and industrial partners[3]. Beside the large size of industrial models, there are a couple of other dimensions still to be addressed. For instance, Event-B uses a rich set of operations as well as complex data based on set relations, sets of sets or partial functions. In principle, these can be mapped to sets and use the proposed methods but this may not scale, so the fitness functions and encodings might need to be further specialized for these operators. Moreover, industrial models are usually decomposed in order to mitigate modeling complexity, which means that we have to extend our methods to work for modular and component-based models.

Acknowledgment This work was partially supported by project DEPLOY (EC-grant no. 214158) and Romanian Research Grant CNCS-UEFISCDI no. 7/05.08.2010 at the University of Pitesti. We thank V. Kozyura and S. Wieczorek from SAP Research for interesting models and inspiring discussions.

[3] *http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html*

References

1. Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.
2. Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. ICSE'11*, to appear.
3. Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools, Object Technology.* Addison-Wesley, 1999.
4. Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.
5. Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Estimating the feasibility of transition paths in extended finite state machines. *Autom. Softw. Eng.*, 17(1):33–56, 2010.
6. David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *FOGA*, pages 69–93, 1990.
7. Klaus Meffert et al. JGAP - Java Genetic Algorithms and Genetic Programming Package. *http://jgap.sf.net.* Last visited, March 2011.
8. Raluca Lefticaru and Florentin Ipate. Functional search-based testing from state machines. In *Proc. ICST'08*, pages 525–528. IEEE Computer Society, 2008.
9. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, 2008.
10. Phil McMinn. Search-based software test data generation: A survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.
11. Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1013–1020. ACM, 2005.
12. Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.).* Springer-Verlag, London, UK, 1996.
13. Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1998.
14. Manoranjan Satpathy, Michael Butler, Michael Leuschel, and S. Ramesh. Automatic testing from formal specifications. In *Proc. TAP'07*, volume 4454 of *LNCS*, pages 95–113. Springer, 2007.
15. David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman & Hall/CRC, 4 edition, 2007.
16. Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Towards search-based testing for Event-B models. In *To appear in Proc. of 4th Workshop on Search-Based Software Testing (SBST'11).* IEEE, 2011.
17. Nigel James Tracey. *A Search-based Automated Test-Data Generation Framework for Safety-Critical Software.* PhD thesis, University of York, 2000.
18. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
19. Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Proc. TESTCOM'09*, volume 5826 of *LNCS*, pages 179–194. Springer, 2009.
20. Thaise Yano, Eliane Martins, and Fabiano L. de Sousa. Generating feasible test paths from an executable model using a multi-objective approach. In *Proc. ICSTW'10*, pages 236–239. IEEE Computer Society, 2010.