A Lazy Unbounded Model Checker for EVENT-B [1]

Paulo J. Matos[1], Bernd Fischer[1], and João Marques-Silva[2]

[1] Electronics and Computer Science, University of Southampton
{pocm,b.fischer}@ecs.soton.ac.uk
[2] School of Computer Science and Informatics, University College Dublin
jpms@ucd.ie

Abstract. Formal specification languages are traditionally supported by theorem provers, but recently model checkers have proven to be useful tools. In this paper we present Eboc, an explicit state model checker for EVENT-B. Eboc is based on lazy techniques that allow it to fairly perform an exhaustive state space search without bounding the size of the sets used in the specification. We describe the implementation of Eboc and provide a preliminary comparison with ProB, an existing bounded model checker for CLASSICAL B.

1 Introduction

Model checking has been the focus of many research papers in recent years, with successes in both hardware and software development. In formal methods, languages are usually supported by theorem provers but model checking has recently been investigated as well and model checkers have been developed for languages like Z [1–3], CSP [4], or CLASSICAL B [5]. This paper addresses the problem of model checking EVENT-B.

The B-method, originally devised by J.-R. Abrial [5], is a theory and methodology for the formal development of computer systems. It is used by industries in a range of critical domains, most notably railway control [6]. EVENT-B [7], an evolution of the CLASSICAL B, focuses on the formal development of discrete systems based on refinement. An EVENT-B specification consists of machines and contexts. A machine defines a state and several events which repeatedly update the state by means of update rules or *actions*, and so provide dynamics to the system. Contexts, which are seen by machines, provide static data to the model. Proof obligations ensure the correctness of the model and its dynamics [8], by for example assuring that invariants remain true after each event's actions.

In this paper, we describe Eboc, an explicit state model checker for EVENT-B. Like other model checkers, Eboc does not discharge proof obligations, but simulates the execution of the model, searching each state for an invariant violation. It thus complements theorem proving, which leaves users in the lurch if it fails to discharge a proof obligation, as the users cannot tell whether the proof

[1] This work is partially funded by EU project Coconut FP7-ICT-217069 and by EPSRC Grant EP/E012973/1.

```
MACHINE Simple
  VARIABLES x
  INVARIANTS
    x ≠ 5
  INITIALISATION ≙
    x := 0
  EVENT Simple ≙
    ANY y WHERE y ∈ ℤ THEN x := y END
END
```

Fig. 1. Simple machine causing a false claim in ProB's bounded model checking

obligation is provable in principle or not. In the latter case, Eboc will (eventually) find a counterexample, describing which state violates which invariant and how it can be reached from the initial state. If the state space is finite, Eboc can even search the complete state space and decisively show whether any invariant is violated. Even if the state space is infinite and the proof obligation is provable Eboc can search through enough of this space to give the user enough confidence to proceed and try to discharge it manually.

Eboc's state space exploration is driven by a scheduler that expands all states and searches for a violation of an invariant in the original (i.e., unbounded) state space. Eboc traverses this potentially infinite state space by lazily enumerating the values for any given variable such that no value is ever repeated and the space is fairly covered. The way Eboc handles the problem of infinite search space is fundamentally different than for example ProB [9], an existing model checker for CLASSICAL B. ProB bounds the state space to be explored by bounding the size of the deferred sets, the integers, the number of initial states computed, and the number of enablings for each state. This up-front bounding however, is of course problematic when the invariant is only violated for values outside the bounds. Consider for example the (deliberately simple) machine shown in Figure 1. Obviously, the invariant is violated if $y = 5$ is chosen. However, ProB's default lower and upper bounds for the integers are -1 and 4, respectively, so the guard will return no violation of the invariant. In this simple example it is of course possible to inspect the model, see that a higher bound is required and overwrite ProB's defaults, but with larger models this will generally become harder, and eventually setting the right bound becomes a trial-and-error issue. The problem is aggravated by the fact that if ProB returns no error, it is unclear whether this is because the model is consistent with the invariants or because the bounds are too restrictive. Moreover, more obscure problems might be due to bounding the number of initial states computed by ProB or due to any other bound imposed by ProB before the search starts. Our approach avoids these bounds, thereby solving this problem. The solution involves the combination of a lazy exploration of the values in the domain with a priority system that avoids the search being deadlocked in a single infinite stream of values. This allows us to fairly explore even infinite domains, or more precisely, given finite time, fairly

chosen finite subsets of unbounded size. In this context, laziness means that all the nodes in the search space are considered however, they are only computed when required for processing.

In Section 2 we present some background concepts important for the rest of the paper, in Section 3 we will present in detail the model checker for Event-B. Section 4 will focus on the architecture of the system and Section 5 will provide a preliminary comparison between our model checker and ProB with a discussion of the results. The paper finishes with an overview of related work in Section 6 and conclusions in Section 7.

2 Event-B Essentials

Event-B is a modelling notation and method for formal development of discrete systems based on refinement [10] which evolved from the B-Method [5]. Here, we give a brief overview of Event-B; for details see [7]. Since our model checker focuses on model checking the validity of invariants, we will ignore the concepts and constructs of Event-B that are irrelevant to this end. In particular, we will ignore the concept of refinement, which makes the specification of large and complex systems more tractable by gradually adding more details to an abstract base model. However, we are not constraining the amount of models that can be model checked: since model checking generally focuses on a specific refinement level, it is possible to *remove* the more abstract levels by flattening the model into the required level. Moreover, statuses and witnesses will not be discussed either since they are associated with the refinement of events. Similarly, we will not discuss theorems which are associated with contexts and machines, because they do not influence the execution of an Event-B machine.

An Event-B specification consists of machines which specify the behavioral properties of the model, and contexts which axiomatically provide static aspects of the model. Event-B's mathematical language is based on first-order logic and set theory (as in classical B) and its syntax, type inference rules, and legibility rules are defined in [11].

Figure 2 presents a simple Event-B model which consists of a single machine and a single context. The context *Colors* consists of the deferred set Colors whose elements are left undefined, three constants and an axiom. The interpretation of the constants is given by the Axioms; in this case, the axioms specify that the deferred set consists of three different values represented by the identifiers red, green, and blue, respectively. The machine *Example* has two state variables x and y, which are initialized to 0 and red, respectively. It can see the *Colors* context, hence all the definitions of the context can be referred to in the machine. The machine defines an event e with two parameters xx and yy, a set of predicates referred to as *guards*, and a set of update rules referred to as *actions*. Events are guarded atomic actions that drive the execution of the model. Once all the variables are initialized, all events are checked for enabledness. An event is enabled if there is an assignment to its parameters that satisfies the guards in the current state. An enabled event is then chosen non-deterministically to

```
MACHINE Example
  SEES
    Colors
  VARIABLES
    x  y
  INVARIANTS
    x ∈ ℤ ∧ y ∈ Colors
    x = 2 ⇒ y ≠ red
  INITIALISATION ≙
    x, y := 0, red
  EVENT e ≙
    ANY
      xx  yy
    WHERE
      xx ∈ ℤ
      yy ∈ Colors
      yy ≠ y
    THEN
      x := x + xx
      y := yy
    END
END
```

```
CONTEXT Colors
  SETS
    Colors

  CONSTANTS
    red  green  blue

  AXIOMS
    partition(Colors, {red}, {green}, {blue})
END
```

Fig. 2. Example EVENT-B specification

be triggered and, once triggered, values are chosen non-deterministically for its parameters and the state is updated according to its actions. In the initial state $x = 0$, $y =$ red, e is enabled. Since this is the only event of the machine, it is triggered and if for example $xx = 1$ and $yy =$ green are chosen as values for the parameters, the new state becomes $x = 1$, $y =$ green. Note that $xx = 1$ and $yy =$ red would not be a valid choice for the parameters as this would violate the guard $yy \neq y$. Once a new state is generated, the process continues until no event is enabled, in which case it is said the model reached a deadlock state.

The initialization of the state variables is given by a set of actions which can take three forms:

$$\mathbf{v} := \mathbf{E}(\mathbf{c}) \tag{1}$$

$$v :\in S(\mathbf{c}) \tag{2}$$

$$\mathbf{v} :| \; P(\mathbf{v}', \mathbf{c}) \tag{3}$$

Here, $\mathbf{E}(\mathbf{c})$ represents a set of expressions over the seen constants, $S(\mathbf{c})$ is a single set expression over the seen constants and $P(\mathbf{v}', \mathbf{c})$ is a before-after predicate, where $\mathbf{v}'$ is the set of variables $\mathbf{v}$ in the after-state. The first form is a deterministic assignment form which assigns the value of each expression on the right-hand side to the set of variables in the left-hand side in order. The second form is a non-deterministic assignment form which assigns to the variable on the left-hand

side one of the elements of the set that result from the evaluation of $S(\mathbf{c})$. The third form, which is the most general form, assigns a value non-deterministically to the variables $\mathbf{v}$ that satisfy the predicate $P(\mathbf{v}', \mathbf{c})$. Event actions can, in general, also take different forms. However, we will assume without loss of generality that all the non-determinism is represented in the guards and that event actions always take the deterministic form. The following is thus the canonical form used for events:

$$\textsc{Event}\ \ \mathrm{e}\ \ \widehat{=}\ \ \textsc{any}\ \ \mathbf{x}\ \ \textsc{where}\ \ P(\mathbf{v}, \mathbf{x}, \mathbf{c})\ \ \textsc{then}\ \ \mathbf{v} := \mathbf{E}(\mathbf{v}, \mathbf{x}, \mathbf{c})$$

Here, $\mathbf{x}$ represents the event's parameters whose scope are the the event's body and whose value are constrained by the guard $P(\mathbf{v}, \mathbf{x}, \mathbf{c})$. The guard enables the event and generates a new state through the application of the actions $\mathbf{v} := \mathbf{E}(\mathbf{v}, \mathbf{x}, \mathbf{c})$, which are analogous to the form shown in Equation (1).

Proof obligations play an important role in Event-B and their purpose is two-fold. On one hand, they show that a model is sound with respect to some behavioral semantics. On the other hand, they serve to verify properties of the model [8].

In this paper, we will focus on checking invariants. Invariants are the predicates that must be satisfied in all states of the model. Event-B ensures that a machine is consistent by constructing proof obligations that formalize the intuition that the machine preserves the invariant. Each event thus induces a proof obligation of the form:

$$\mathbf{I}(\mathbf{v}) \wedge \mathbf{P_e}(\mathbf{v}, \mathbf{x}, \mathbf{c}) \Rightarrow \mathbf{I}(\mathbf{v}') \tag{4}$$

Intuitively this means that if the invariants and the guard for event e hold in the current state, then the invariants also hold in the post-state. If this is proven for each event, then the machine is consistent. Eboc focuses on checking that these invariants are never violated. Eboc checks this by simulating the model, generating and exploring a state space by repeatedly applying all possible events that are enabled in the given state.

3 Explicit State Model Checking of Event-B

In contrast to ProB, Eboc performs a lazy, unbounded, explicit state search of an Event-B model. Before going into the details of how Eboc model checks an Event-B model, we present a brief discussion on what it means to do explicit state model checking of an Event-B model. Consider a slightly simplified version of event e of Figure 2:

$$\textsc{Event}\ \ \mathrm{e1}\ \ \widehat{=}\ \ \textsc{when}\ \ \top\ \ \textsc{then}\ \ x := x + 1$$

where x is always incremented. Consider also a second event:

$$\textsc{Event}\ \ \mathrm{e2}\ \ \widehat{=}\ \ \textsc{any}\ \ zz\ \ \textsc{where}\ \ zz \in \mathrm{Colors} \wedge x \bmod 2 = 1\ \ \textsc{then}\ \ x, y := x + 1, zz$$

which increments x and non-deterministically chooses a new color for y if x is odd. Figure 3 shows the search tree for the machine of Figure 2 but with the

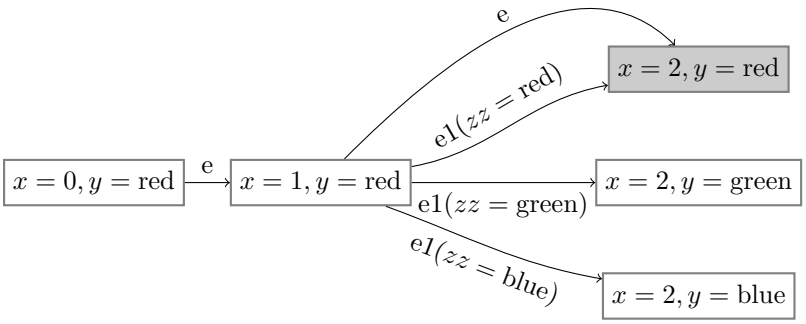

Fig. 3. Example of a search tree on an Event-B model.

events e and e1 above. The initial state is created and from it, all the enabled events are expanded: initially only event e because x is 0. In the resulting state both events are enabled however, four transitions occur to three different states. One transition is due to event e and the remaining three are due to the non-deterministic assignment of zz to one of the possible colors. Note what would have happened if there was a non-deterministic assignment to an integer variable, instead of a variable whose type is finite. The branching from this state would have been infinite. Current model checkers opt to bound these infinite sets so they can perform a search on the state space. However, our approach is to leave the infinite sets unbounded and lazily unroll them during search. Therefore, Eboc has the advantage of finding violations others may not find because their domains may be bounded to too restrictive values. This means that nothing is bound a priori and the search space (potentially infinite) is possibly exhaustively searched for a counterexample. However, this space is only unfolded when required. Therefore a counterexample, if it exists, is eventually found. The downside is that the model checker will not stop if the space is infinite and the model is correct. That is why we allow the user of Eboc to input a bound on the number nodes (which represent states in the system) to explore. In this case, Eboc assures the user that within that space no counterexample exists. The remainder of the section presents our approach in more detail.

3.1 Enumerations

In order to have an exhaustive search of an infinite state space, we need to have a methodology to list for every variable in the model all the values they can take, which are possibly infinite, so we need to lazily enumerate them. Whenever we need to choose non-deterministically a value for a given variable we take a value from a lazy stream of possible values (which depend on the type of the variable) and if the value does not fit the constraints the variable is subject to, we backtrack and try another. To this end, we discuss enumerations.

Consider again the simple example in Figure 1. Event Simple is always enabled, and can be triggered an infinite amount of times from a given state and for different values of y. From the initial state $x = 0$, each time we trigger event Simple we take a new y from the lazy stream of values $[0, -1, 1, -2, 2, \ldots]$, which generate the states $x = 0$, $x = -1$, $x = 1$, ... respectively. If instead the event was of the form:

EVENT Simple1 $\widehat{=}$ ANY y z WHERE $y \in \mathbb{Z} \wedge z \in$ Colors ...

where it can be assumed that Colors is the set defined in Figure 2, then we would need to lazily enumerate all values in $\mathbb{Z} \times Colors$, assign each of the values to y and z respectively and evaluate the state, thereby generating an infinite amount of states. The lazy stream this time would look like $[0 \mapsto \text{red}, 0 \mapsto \text{green}, 1 \mapsto \text{red}, 0 \mapsto \text{blue}, \ldots]$, where $x \mapsto y$ is EVENT-B's representation of the pair (x, y).

In general, an enumeration of a set S is a surjection f from $\mathbb{N}$ onto S. This definition allows that two different natural numbers have the same image under f, which is something we do not want for efficiency reasons, so f is, in our case, also injective.

EVENT-B has a flat type system that includes as basic types user defined sets, the booleans, the integers and cartesian product along with the powerset as type constructors. For each of these types we define enumerations that allow us to lazily step through each possible value induced by the type. However, we had to make sure these enumerations are not only injective but also fair. There are two dimensions to fairness:

1. it has to explore all the possible values the variable can have. For example, for an integer variable, it would not be fair to first explore all the positive number and then all the negative numbers because since the positive numbers are infinite the negative numbers would never be explored, even if given infinite time;
2. and, given a list of variables whose values we need to enumerate, we need to alternate the variable we change the value for next. For example, when enumerating all the values of a list of two integer variables, we cannot first enumerate all the values for the first and then, increase the second, enumerate all the values for the first and so on. If the first variable has an infinite domain, we end up never increasing the second variable. We need to alternate the variables to modify.

How each of the presented enumerations is fair will become clear during their presentation. To enumerate all possible values that a variable of given type can have, we will generate them recursively and them compose them. Assume a variable x has type $\mathbb{P}(\mathbb{Z} \times \text{BOOL})$ and the following enumerations:

- an enumeration h for powersets from $\mathbb{N}$ to $\mathbb{P}(\mathbb{N})$;
- an enumeration g for cartesian products from $\mathbb{N}$ to $\mathbb{N} \times \mathbb{N}$;
- an enumeration f_1 for integers from $\mathbb{N}$ to $\mathbb{Z}$, and;
- an enumeration f_2 for booleans from $\mathbb{N}$ to BOOL.

To obtain an enumeration for $\mathbb{P}(\mathbb{Z} \times \text{BOOL})$, we need to compose the above enumerations in the following way: given an n, we apply h to obtain a set of naturals $\{h_0, \ldots, h_s\}$. To every element of this set we apply g resulting in $\{g_{00} \mapsto g_{01}, \ldots, g_{s0} \mapsto g_{s1}\}$. Then we can apply f_1 to all the first elements of each pair and f_2 to all the second elements of each pair obtaining a set in $\mathbb{P}(\mathbb{Z} \times \text{BOOL})$: $\{f_{11} \mapsto f_{21}, \ldots, f_{1s} \mapsto f_{2s}\}$. This is a powerful method because it only requires us to define a few types of enumerations which by composition allows up to obtain enumerations for any possible EVENT-B type. The following enumerations will be defined:

- An enumeration for carrier sets S, mapping a subset of $\mathbb{N}$ to S;
- an enumeration for the integers, mapping $\mathbb{N}$ to $\mathbb{Z}$;
- an enumeration for the powerset, mapping $\mathbb{N}$ to $\mathbb{P}(\mathbb{N})$, and;
- an enumeration for fixed size lists, mapping $\mathbb{N}$ to $\mathbb{N} \times \ldots \times \mathbb{N}$. We require this enumeration to provide a lazy stream of values for a list of variables, for example, the list of local variables in an event. Furthermore, this enumeration for size 2 provides an enumeration of pairs.

The simplest enumeration is the one defined for user defined sets. Considering a set $S = \{s_0, s_1, \ldots, s_n\}$, the enumeration is a function that maps the first $n+1$ natural numbers to each of the elements of the set S. An enumeration for the set Colors defined in the context shown in Figure 2 would be $\{0 \mapsto \text{red}, 1 \mapsto \text{green}, 2 \mapsto \text{blue}\}$.

The enumeration of the integers is given by a function f, where $f(x) = -(x+1)/2$ if x is even and $f(x) = x/2$ otherwise. This generates an enumeration that jumps between the positive and negative numbers, without giving precedent to the positive or the negative numbers making it a fair enumeration for the whole set of integers.

Enumerating pairs is the same as enumerating lists of size 2. To enumerate lists, we enumerate first all of those whose elements sum 0, then all whose elements sum 1, and so on. This generates a diagonal perspective on the enumeration. Figure 4, on the left, represents diagrammatically how the enumeration proceeds for pairs of naturals. In the case of pairs, the only pair summing 0 is $0 \mapsto 0$, then all comes all of those summing 1: $0 \mapsto 1$ and $1 \mapsto 0$, and so on. Consider the enumeration of Colors $\times \mathbb{Z}$, where Colors is defined in Figure 2, in this case since Colors is finite, the enumeration will not generate pairs whose first element is bigger than 2, therefore having a diagrammatic representation as the one shown in the right of Figure 4.

Even though sometimes it is easy to find an explicit form as a function for an enumeration (as in the case of the integers), it is not so easy for more complex structures like lists or sets, so we will not pursue such representation and instead we will in these cases focus on how to go from one value to the next. The process of generating all lists $(enum\ sz\ s)$ which have a specific sum can be thought of recursively as two cases:

1. $(enum\ \mathbf{1}\ s) \mathrel{\widehat{=}} (\mathbf{list}\ (\mathbf{list}\ s))$
2. $(enum\ sz\ s) \mathrel{\widehat{=}} ((\mathbf{cons}\ i\ (enum\ (-\ sz\ \mathbf{1})\ (-\ s\ i))) \ldots)$

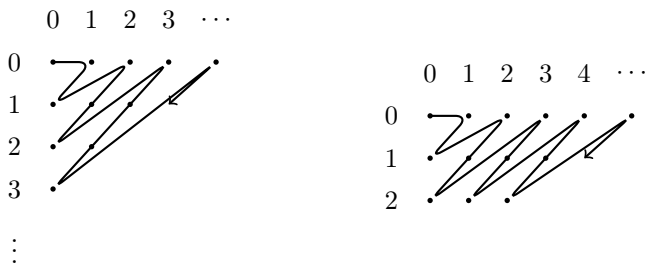

Fig. 4. Diagrammatic representation of enumeration of pairs in $\mathbb{N} \times \mathbb{N}$ (on the left) and $\{0, 1, 2\} \times \mathbb{N}$ (on the right)

card(s_i)	
0	s_0
1	$s_1, s_2, s_4, s_5, s_8, s_9, s_{11}, s_{12}$
2	s_3, s_6, s_{10}, s_{13}
3	s_7, s_{14}
4	s_{15}

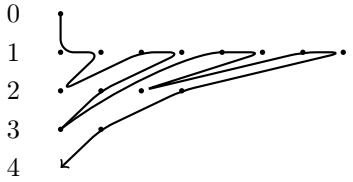

Fig. 5. Diagrammatic representation of set enumeration with $step = 2$

The first case is the base case that returns a list of all the lists of size 1 and a given sum. The second case builds all the lists of size sz and sum s by noting that the problem can be reduced by building on the lists one element smaller thus generating a recursive solution to the problem.

The enumeration for sets is analogous to lists, with the constraint that two elements in a set cannot be the same and that we need to generate sets of different sizes in a fair order. We have a parameter which we called *step* that defines how many sets of size n we have to generate until we generate a set of size $n + 1$. As such, we can then lazily enumerate all the sets fairly by starting with the empty set and increasing their size. Figure 5 explains how the *step* parameter works, where $f(i) = s_i$, and f is a powerset enumeration.

3.2 Model Checking

In what follows we explain the search method used to find an invariant violation in Eboc through an example and discussing some important details in the end. Consider again the machine Example in Figure 2 and the part of the generated search tree in Figure 6.

In Figure 6, each of the shaded rectangles represent a state and the white rectangles with rounded corners represent choice points.

The simulation starts by setting up a choice point for the initial states. A choice point represents a suspension of an assignment, which might possibly have infinite results. This happens whenever a choice is required, as for example in

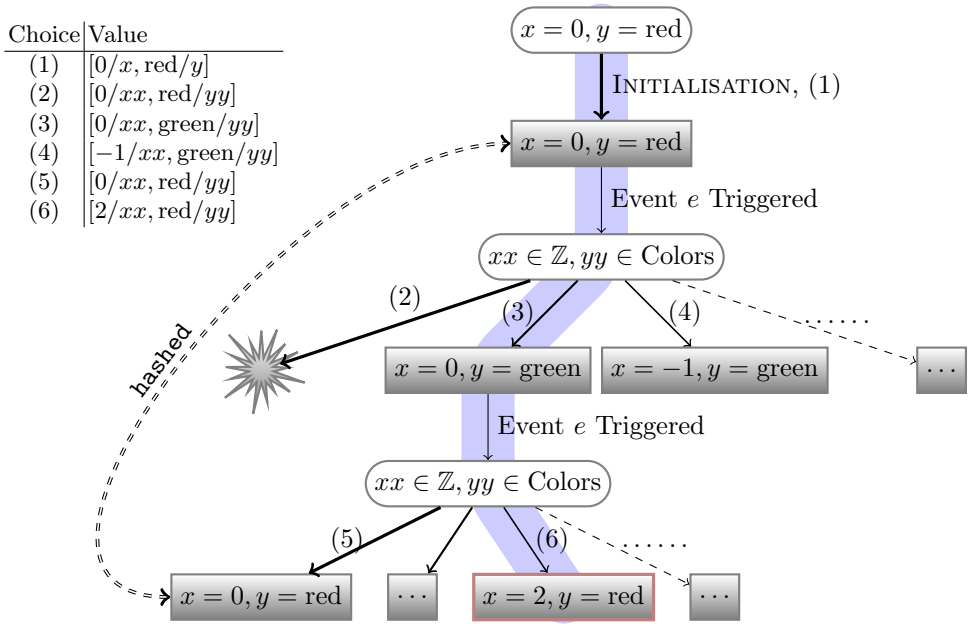

Fig. 6. Search space of the model shown in Figure 2

the case of choosing a value for the parameters of an event or in the use of quantifiers.

In this case there is only one initial state since the initialization is deterministic, $x = 0$, $y =$ red, and therefore the choice point does not branch. Then, all events generate a suspension which represents a choice point for all the parameters. Event e generates a suspension for the choice of two parameters: $xx \in \mathbb{Z}$, $yy \in$ Colors. As it can be seen in 6 we are representing states with rectangles with a gradient background and choice points with rectangles with rounded corners and white background. The choice points are where the lazy enumeration happens. In this case the scheduler will enumerate values of the for $\mathbb{Z} \times$ Colors. The first enumeration $xx = 0$, $yy =$ red generates no state since it violates one of the guards: $yy \neq y$. Note the thickness of the lines out of choice points, representing the priority with which a state is generated from a given choice. The next enumeration is: $xx = 0$, $yy =$ green generating the state $x = 0$, $y =$ green. This state generates again a suspension for the triggering of event e. At this point it is important to note the relevance of priorities in the search. The choice point has still infinitely many states to generate, but their priorities decrease as it generates more and more states from this. The second choice point has a infinitely more states to generate but the first state has again the highest possible priority and that will be the one that will be generated. The first enumeration is $xx \in \mathbb{Z}$, $yy \in$ red generating the state $x = 0$. $y =$ red but since states are hashed, the state is promptly discarded. From this point, both choice points will

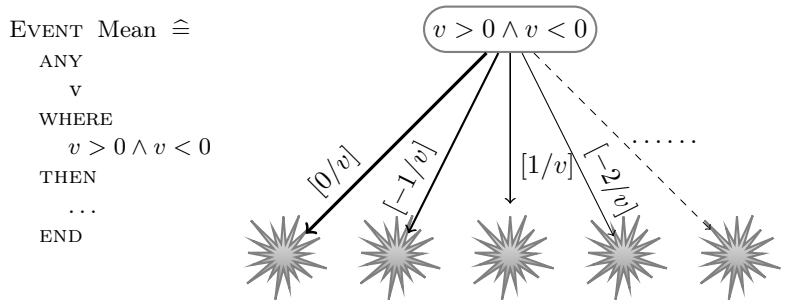

Fig. 7. Event and tree presenting an impossible guard to satisfy

generate new states whose order will depend on their priorities. Once the second choice point tries the enumeration $x = 2$, $y =$ red, generating the state $x = 2$, $y =$ red the invariant evaluator, signals a violation and the process stops returning the trace: INITIALISATION, Event $e(xx = 0, yy =$ green), ($x = 0$, $y =$ green), Event $e(xx = 2, yy =$ red), ($x = 2$, $y =$ red). This is exactly the path shaded in Figure 6 and represents the path to the state violating the invariant.

This lazy method assures that all the space will be searched and if the model is finite the process will stop. If the model is infinite, the user either aborts the search after some time, or sets up the number of states that should be searched. This method generates a search tree that mixes depth with breadth first search and focusses the attention on the values which have the highest priority of generating a state which violates the invariant. For example, the enumeration of the integers starts at zero but it is possible to change the enumeration to allow it to start at any other point by adding to each of the values of the enumeration a specified offset. The priorities, which can be thought of as probabilities, are generated from a normal distribution. Note, that they are not exactly probabilities because their values range from 0 to 1, and the sum of all the priorities from a choice point does not sum 1.

Consider the event shown in Figure 7. This event has an impossible guard to satisfy which generates a choice point during the search that will never succeed in generating a state. The only reason why the search does not stop here in an infinite loop is because choice points generate branches with decreasing priority. After a while, depending on what the scheduler has on queue, the search will focus on some other part of the tree. This does not mean this choice point will be forgotten, but it will not be tried out as often as the rest. This is so that cases like $v > 1000000$ have a chance of ever generating a state.

EVENT-B supports definitions of constants whose value is constrained by a set of predicates, known as axioms. The model checking process handles them in the example same way as parameters. Constants are left undefined in the beginning and their value is only searched for once we notice an event needs their value. In this case, the choice point of the event will contain a choice for all the constants defined by the model constrained by all the axioms of the model

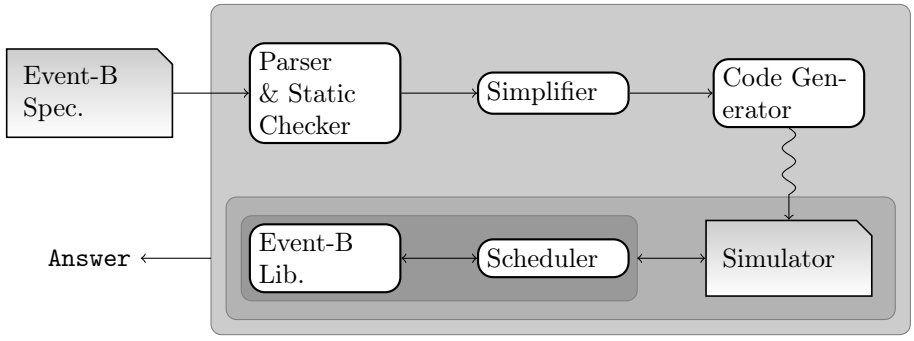

Fig. 8. Eboc system architecture

and from that point onwards the scheduler will not worry about them anymore since below that branch their value is already assigned. This is highly inefficient: a value is assigned to all the constants once one of them is referenced, however, it should only be required to assign a value to the constants referenced in the event and those that require a value because they share the same constraint. For example, if an event uses a constant x constrained by the axioms $x < y$, then we need to assign a value to x and y but no other constant that might exist. We hope to address this in the future to improve the efficiency of Eboc on models dealing with a lot of constants.

4 System Architecture

Eboc is fully implemented in PLT-Scheme [12] and was designed to be an easily extensible tool to work with Event-B models. Even though we implemented a model checker in the backend, it would be very easy to integrate other types of tools. We will focus this section on the system architecture and some implementation details which are important to understand how Eboc performs the lazy model checking.

Eboc is a command line tool that in its simplest form receives a file describing an Event-B specification and a natural number (the number of states to verify) and returns either that the verification was successful, or a trace to a state that violates one of the invariants. Figure 8 shows the system architecture. An Event-B specification is received as an argument and passed into the parser and static checker. The parser is responsible for generating an abstract syntax tree and the static checker besides assigning a type to each node through type inference, check that the tree is legible. The type inference and legibility rules for Event-B can be found in [11]. After the static check has been performed the simplifier does a series of simplifications to the model in order to simplify the simulation without affecting its performance. The code generator outputs the simulator code specific to the simulation of the input model which is linked to the scheduler and the Event-B library to produce the final answer.

4.1 Scheduling

The scheduling is performed by a function which given a search structure and the number of states to explore explores the state space of the model until either a violation is found or the number of states to explore has been reached.

The search structure

(**define-struct** *search* (*ns igen evgen props*))

has four elements, the number of states to explore, the initialization suspension, a list of suspension generators representing the events and a list of procedures that represent the properties that each state needs to verify.

Figure 9 shows a simplification of the scheduling algorithm. The functions *pq:enq!*, *pq:next-el*, and *pq:empty?* act on a global priority queue that contains suspensions and their respective priorities. These functions enqueue a suspension with a given priority, dequeue and remove the element with the highest priority from the queue and check if the queue is empty respectively. The scheduling function begins by enqueueing the initialization suspension which corresponds to the suspension that will generate all possible initialization states. Then it enters a loop that only stops on one of three conditions:

1. Either the number of violations found until now is not zero, in which case it returns the violations;
2. the number of states to explore as been reached, in which case it returns that no violations where found, or;
3. the queue is empty, which means that no states are left to explore and the whole state space has been explored.

Otherwise the procedures enters the **else** clause of the **cond** in the loop. Here we get the element with the highest priority in the queue. If this element is empty, meaning that no states are left to generate from this suspension, then we loop, otherwise we proceed by getting the new state the suspension has to generate (*new-state*) and the priority of the next state of the suspension (*new-prt*). Since the element was not empty, it is enqueued with the new priority and the code that follows handles the new state. If the new state has already been explored, then we loop and forget the new state, otherwise we increment the number of explored states and verify which properties have been violated in this state (*vprops*). If the number of violated properties is non-zero we loop with the violated properties to return to the user. If there are no violated properties in this state we hash the new state and enqueue all event suspensions that are not empty, meaning they generate some state and then we loop.

4.2 Code Generation and Simulation

Eboc makes use of units [13] in order to plugin automatically generated code. Code that simulates this model is generated, linked with the scheduler and the EVENT-B library, plugged into the main system and executed. Once the simulator terminates, its code is discarded and an answer is provided to the user.

```
(define (begin-search! n s)
  (pq:enq! (search-igen s) ((search-igen s) 'prt))
  (let loop ((violations '()))
    (cond ((not (null? violations)) violations)
          ((> (search-ns s) n) '()) ; State bound achieved
          ((pq:empty?) '())         ; State space fully explored
          (else
           (let* ((element (pq:next-el!))) ;; Returns Removes high priority element
             (if (element 'empty?)
                 (loop '())
                 (let ((new-state (element 'stt)) (new-prt (element 'prt)))
                   (pq:enq! element new-prt)
                   (if (hashed? new-state)
                       (loop '())
                       (let ((vprops (foldl (λ (p acum)
                                              (if (p new-state)
                                                  acum
                                                  (cons p acum)))
                                            '()
                                            (search-props s))))
                         (inc1! (set-search-ns! s))
                         (if (not (null? vprops))
                             (loop vprops)
                             (begin
                               (hash! new-state)
                               (for-each (λ (proc)
                                           (let* ((gen (proc new-state)))
                                             (when (not (gen 'empty?))
                                               (pq:enq! gen (gen 'prt)))))
                                         (search-evgen s))
                               (loop '()))))))))))))))))))))))))
```

Fig. 9. Simplified algorithm of the scheduler in pseudo-Scheme code

The most important thing to consider is *what are suspensions?* Suspensions are closures over some variables that dictate how the next state is generated. Consider the following non-deterministic event NDet:

EVENT NDet $\widehat{=}$ ANY x WHEN $x > 0$ THEN $y := y + x$ END

The code generated for this particular event is shown in Figure 10. Each event generates a pair: guard/action procedures and it is the guard procedure that handles all the complexity related to non-determinism. The actions are always deterministic (which is not a restriction as described in section 2). The variable *enum* is a function that generates a lazy stream of values whose types are listed in the argument for *type-list-enumerator*, in this case *INT*. The closure receives a message which is then handled as appropriate. If a state is requested through the message 'stt, then the local state is generated, the guard is evaluated and if the guard evaluates to true, the action is then invoked returning a new state. A pair of procedures, as shown in Figure 10, is generated per each event besides initialization code that sets up the search structure discussed in section 4.1 and the initial call to the *begin-search!* procedure. All this code is wrapped around a unit, which can be thought of as a pluggable module, compiled on the fly and linked to the main Eboc components.

```
(define (NDet-guard state)
  (let* ((enum (type-list-enumerator '(INT)))
         (next-enum (enum))
         (next-prt (enum 'prt)))
    (λ (msg)
      (case msg
        ((empty?) (not next-enum))
        ((stt)
         (let ((local-state (map cons '(var:x) next-enum)))
           (begin0
             (if (eval-predicate '(> var:x 0) state local-state)
                 (NDet-action state local-state)
                 #f)
             (set! next-enum (enum))
             (set! next-prt (enum 'prt)))))
        ((prt) next-prt)))))

(define (NDet-action state local-state)
  (foldl (λ (assign-pair acum)
           (state-update acum
                         (car assign-pair)
                         (eval-expression (cdr assign-pair) state local-state)))
         state
         (list (cons 'var:y '(+ var:y var:x)))))
```

Fig. 10. Code generated for deterministic event NDet.

4.3 Event-B Library

The Event-B deals with the evaluation of expressions and predicates. It mainly implements the operations of Event-B and provides two function: *eval-predicate* and *eval-expression* which evaluate a predicate or an expression respectively in a given state. Note that during the code generation each event guard and action contains symbolic expressions that represent Event-B predicates and expressions. All of this is done through code generation and the simulation is performed on a symbolic expression representation of Event-B.

5 Experiments

In this section we will report some preliminary experiments with Eboc. We will present four different models, show some results of their execution in ProB and Eboc and comment on the results.

All experiments were run on a Pentium-D 3.2GHz, with 2Gb of memory under a 64bit Gentoo Linux operating system with a timeout of 1200 seconds. Eboc was ran from the console and ProB was executed from its GUI. The measured time is shown always in seconds and in ProB reflects the time from the button to start the model checking is pushed until the experiment is completed (either because a violation is found or because the number of state to explore has been reached. For the remainder of this section, by *run* we mean a single execution of the model checkers with a given bound. Unless noted otherwise all the ProB runs were executed using the default settings of ProB-1.3.0-rc3 (compiled for 64bit). More

	100 States		1000 States		10000 States		100000 States	
	Eboc	ProB	Eboc	ProB	Eboc	ProB	Eboc	ProB
Bakery1	3	1	4	2	13	11	141	317
Jukebox	3	1	4	59	10	217	108	>1200
Huffman	3	1	5	3	49	40	820	>1200
Consts	3	1	4	3	23	42	26	>1200

Table 1. Experimental results of running Eboc and ProB on four different models (runtime is shown in seconds).

over, ProB was set to only verify the invariants during model checking (which differs from the default option which includes also the check for deadlocks).

The first model is the Bakery1 model, which is distributed with ProB. It is a B model that we converted to Event-B syntax so that we could model check it with Eboc. This is a simple deterministic model where the invariants are not violated. The first row of Table 1 shows the timings for the execution of Eboc and ProB for different number of states to explore.

The Bakery1 model has four integer variables whose value is updated by six different events through simple arithmetic operations. One interesting point is that even though ProB is very fast for small number of states it gets slower as more and more states are explored until the point that it gets slower than Eboc. Given that this model has no invariant violation, none of the model checkers reported a violation.

The following model is the Jukebox and its experimentation table is shown in the second row of Table 1. The Jukebox is a model from a book about classical B [14], which is also distributed by ProB as an example. Once again we converted the model from classical B to Event-B so that we could use it with Eboc. The Jukebox machine sees a context that declares a deferred set and a constant, all of the machine events are non-deterministic and the update rules are set expressions.

In this example, after the first case ProB asked to increase the bound on the number of computed initializations because otherwise it would have no more states to explore. So, for all the bounds higher than 100, ProB default setting of computing 4 initializations was changed to 100. This is the reason why ProB got slower than Eboc for the remaining tests. Again, since there no violation of invariants in the model, none was reported.

Third row of Table 1 shows the experiments regarding the Huffman model by John Colley [15]. The model simulates the encoding and decoding of an infinite string of vowels from a fixed huffman tree. The model, which has 14 events, is non-deterministic and declares an enumerated set among several variables. The variables are sets or integers and most of the update rules deal with set expressions.

Eboc deals very well with these models and scaled very well. However ProB after a certain number of nodes have been explored the performance deteriorates

very quickly. No invariant is violated and none of the model checker report a violation.

The Consts model is an artificial model created by us (ref. appendix A) to explore the handling of the constants when under a lot of non-determinism and which has a violation very far from the initial of the search. Last row of Table 1 shows the results for this experiment.

For the first 3 runs, neither of the model checkers found a violation and had similar performance (even though ProB already took three times more than Eboc on its third run). However, on the fourth run, Eboc found a violation after 26 seconds and ProB ran past the timeout without returning any violation. This is a case where ProB would not find the bug due to its default bounds and where Eboc had no problem finding the bound if given enough freedom to search the state space.

In conclusion, ProB is a very mature model checker with a wide range of options and model checking techniques. ProB seems to be extremelly fast for a small number of states (< 1000) but then its performance detiorates quickly. Unfortunately, at this point we did not any find real world examples that violate its invariant and where ProB is unable to detect it due to its restrictive bounds.

6 Related Work

Traditionally formal languages are supported by automated theorem provers with the notable exception of Alloy [16] which is supported by KodKod [17], a model finder.

However, other formal languages, more notably CLASSICAL B, Z and CSP, have already included a model checker to their available tools, but none has attempted to perform lazy unbounded model checking. Since the languages EVENT-B and CLASSICAL B are closely related we will concentrate our discussion in the B model checker, ProB.

ProB [9] is an animator, constraint-based checker and temporal bounded model checker for CLASSICAL B developed in SICStus Prolog. We will focus on its use as a bounded model checker. ProB requires the input of several types of bounds: bound on the size of the set of integers, bound on the number of computed initializations, and bound on the number of computed enablings (along with a timeout for computing them). Even though ProB provides default values for each of these, in practice there might be models whose faults lie outside the state space set by these bounds forcing the user to tweak them so that a faulty state can be reached. ProB as a model checker tries to find whether a machine violates its invariant by finding a sequence of operations that, starting from the initial state of the machine, navigates the machine into a state in which the invariant is violated. The exploration is done using an adaptation of the A* algorithm with cycle detection, and can be tuned to perform in the extreme cases as either a depth-first or breadth-first search. By default every node had 25% chance of being treated in a depth-first manner. ProB has been adapted over the years to check goals written in Linear Temporal Logic (LTL), and to

model check Z, CSP and Promela [18]. ProB integrates symmetry reduction [19] and more recently, introduced support for Event-B.

On the subject of Event-B model checking, besides ProB we know about an attempt to use SAL, KodKod and BDDs to model check Event-B [20] however, at the time of writing there is no software available to experiment with.

7 Conclusions

In this paper we presented a new model checker for Event-B based on a lazy strategy to explore the state space of the models in an unbounded way. We focused our discussion around the problem we were trying to solve: *how to perform explicit state model checking and yet avoid bounding our domains?* We presented techniques to fairly enumerate the space of values of Event-B expressions, the model checking algorithm that makes use of these enumerations to lazily explore the state space and the Eboc system architecture details.

The work proposed in this paper is based on lazy streams coupled with a priority scheme and seems to work well in theory as well as in practice, even though there are still improvements to be made to the implementation in order to improve the efficiency of Eboc. Another important step is in finding complex case studies that demonstrate the importance of this approach and that Eboc is successful in finding invariants violations in these case studies, which would be otherwise impossible using bounded approaches.

A Consts Model

Machine consts

Sees consts

Variables x

Event e1 $\widehat{=}$
ANY
xx
WHERE
$\top$
THEN
$x := x + xx$
END

Event e2 $\widehat{=}$
ANY
xx
WHERE
$xx > c2$
THEN
$x := xx + x + c1$
END

Event e3 $\widehat{=}$
WHEN
$\top$
THEN
$x := c1 + c2 + x$
END

Initialisation $\widehat{=}$
$x := 0$

Invariants
$c1 = 2 \Rightarrow x \neq 150$
$(x \geq -5000) \wedge (x \leq 5000)$

End

CONTEXT consts
  CONSTANTS $c1$ $c2$
  AXIOMS
    $c1 < c2$
END

References

1. Smith, G., Wildman, L.: Model checking Z specifications using SAL. In *Proc. 4th Intl. Conf. of B and Z Users*, *LNCS 3455*, pp. 85–103. Springer, 2005
2. Derrick, J., North, S., Simons, T.: Issues in implementing a model checker for Z. In *Proc. 8th Intl. Conf. Formal Engineering Methods*, *LNCS 4260*, pp. 678–696. Springer, 2006
3. Derrick, J., North, S., Simons, A.J.H.: Z2SAL - building a model checker for Z. In *Proc. 1st Intl. Conf. Abstract State Machines, B and Z*, *LNCS 5238*, pp. 280–293. Springer, 2008
4. Hoare, C.A.: Communicating Sequential Processes. Prentice Hall, 1986
5. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, 1996
6. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: Météor: A successful application of B in a large project. In *Proc. of World Congress on Formal Methods*, *LNCS 1708*, pp. 369–387. Springer, 1999
7. Abrial, J.R.: Modeling in Event-B: Systems and Software Engineering. To be published by Cambridge University Press, 2009
8. Hallerstede, S.: On the purpose of Event-B proof obligations. In *Proc. 1st Intl. Conference on Abstract State Machines, B and Z*, *LNCS 5238*, pp. 125–138. Springer, 2008
9. Leuschel, M., Butler, M.: ProB: A model checker for B. In *Proc. Intl. Symposium of Formal Methods Europe*, *LNCS 2805*, pp. 855–874. Springer, 2003
10. Hallerstede, S.: Justifications for the Event-B modelling notation. In *Proc. 7th Intl. Conf. of B Users*, *LNCS 4355*, pp. 49–63. Springer, 2006
11. Métayer, C., Voisin, L.: The EVENT-B mathematical language. http://wiki.event-b.org/index.php/Event-B_Mathematical_Language, March 2009
12. Flatt, M., et al.: Reference: PLT Scheme. Reference Manual PLT-TR2009-reference-v4.2, PLT Scheme Inc., June 2009
13. Flatt, M., Felleisen, M.: Units: Cool modules for hot languages. In *Proc. Conf. on Programming Language Design and Implementation*, *SIGPLAN Notices, 33(5)*, pp. 236–248. ACM, 1998
14. Schneider, S.: The B-method — an introduction. Palgrave Macmillan, 2001
15. Colley, J.: An Investigation into using Event-B for sub-system development in a SystemC TLM flow. *Private Communication*, July 2007
16. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, 2006
17. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In *Proc. 13th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS 4424*, pp. 632–647. Springer, 2007
18. Leuschel, M., Plagge, D.: Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. Technical Report STUPS/2007/02, Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, 2007

19. Spermann, C., Leuschel, M.: ProB gets nauty: Effective symmetry reduction for B and Z models. In *Proc. 2nd Intl. Symposium on Theoretical Aspects of Software Engineering*, pp. 15–22. IEEE, 2008
20. Plagge, D., Leuschel, M., Lopatkin, I., Iliasov, A., Romanovsky, A.: SAL, Kodkod, and BDDs for validation of B models. lessons and outlook. In *Proc. 4th Workshop on Automated Formal Methods*, June 2009