Proof Assisted Symbolic Model Checking for B and Event-B

Sebastian Krings and Michael Leuschel

Institut für Informatik, Universität Düsseldorf**
Universitätsstr. 1, D-40225 Düsseldorf
{krings,leuschel}@cs.uni-duesseldorf.de

Abstract. We have implemented various symbolic model checking algorithms, like BMC, k-Induction and IC3 for B and Event-B. The high-level nature of B and Event-B accounts for complicated constraints arising in these symbolic analysis techniques. In this paper we suggest using static information stemming from proof obligations to simplify occurring constraints. We show how to include proof information in the aforementioned algorithms. Using different benchmarks we compare explicit state to symbolic model checking as well as techniques with and without proof assistance. In particular for models with large branching factor, e.g., due to complicated data values being manipulated, the symbolic techniques fare much better than explicit state model checking. The inclusion of proof information results in further clear performance improvements.

Keywords: B-Method, Event-B, Proof, Symbolic Model Checking

1 Introduction and Motivation

Model checking is one of the key techniques used in formal software development. Two variants are currently in use: explicit state model checking and symbolic model checking. In explicit state model checking, every state is computed, the invariant is verified and discovered successor states are queued to be analyzed themselves. Symbolic model checking on the other hand tries to represent the state space and possible paths through it by predicates representing multiple states at once. Instead of stepwise exploration of the state space graph, the model checking problem is encoded as a formula and given to a constraint solver.[1]

So far, existing model checkers for B and Event-B like ProB [21,20], Eboc [23], pyB [30] or TLC [31] (via [17]) rely on explicit state model checking. ProB features some symbolic techniques for error detection [14] and test-case generation [27], but not full-blown symbolic model checking. This is mostly due to the high-level nature of B and Event-B. Both the usage of higher-order constructs

** Part of this research has been initially sponsored by the EU funded FP7 project 287563 (ADVANCE).

[1] BDD-style model checking [10] is also called symbolic model checking. In recent work ProB has been integrated with LTSMin for such kind of model checking.

and the underlying non-determinism accounts for complicated constraints during symbolic model checking. Some complexity can be coped with by relying on SMT solvers [12] or SAT solvers [25], but this is not always the case.

In this paper we have implemented various symbolic model checking algorithms for B and then study various ways to use proof information to optimize them. The proof information is used to strengthen constraints and reduce the counterexample search space. For Event-B, the information stems from discharged proof obligations exported from Rodin [1]. For classical B, no automatic proof information is available at the moment within ProB.[2] However, we can recompute the information using ProB's proof capabilities as outlined in [18]. Essentially, for a B operation with before-after-predicate BA we search for a solution to

$$invariant(x) \land BA(x, x') \land \neg conjunct_of_invariant(x').$$

If ProB reports a contradiction, we know that the operation can not lead to a violation of the particular conjunct. In addition to the technique in [18], we developed a bridge to the Atelier B provers ml and pp.

All techniques used in this paper have been implemented both for classical B and Event-B. Both languages will be used in our empirical evaluation. For the sake of brevity we will only talk about Event-B events in the following sections instead of distinguishing events and operations.

We will introduce the model checking algorithms BMC, k-Induction and IC3 in Sections 2.1 to 2.3. For each of them we will show how to include proof information into the occurring constraints. Following, in Section 3, we will empirically compare symbolic model checking to explicit state model checking and model checking with and without proof assistance. Discussion and conclusions will be presented in Section 4.

2 Proof Assisted Symbolic Model Checking

When using the B method to develop a software or system, one often alternates between different phases. Among those are writing and adapting the specification, manual and automated proof efforts as well as model checking. Yet, the different steps are only loosely coupled when it comes to tool support.

In [4] the authors have shown how to augment explicit state model checking with proof information. In the following, we introduce three symbolic model checking techniques for B and incorporate proof information in a similar fashion.

We will use the running example in Fig. 1 to illustrate various concepts in our paper. First, let us introduce the notation we will be using. By x we will denote a vector of state variables. x' denotes the state variables in the successor state. A predicate p over the state variables x is denoted by $p(x)$. The same predicate over the successor state is written as $p(x')$. By *Events* we denote the set of events of a B machine. By *Inv* we denote its invariant.

[2] In theory, one could export proof information from Atelier B.

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN c := c+i END
END
```

Fig. 1: A simple, erroneous B machine

Definition 1. *For an event $evt \in Events$ let $BA_{evt}(x, x')$ denote the before-after-predicate connecting state variables in x to their successors in x'.*

Definition 2. *For a predicate $p = \bigwedge_{i \in I} p_i$ and event evt let $proven_{evt,p}$ denote a set of conjuncts p_i that are proven to hold after the execution of evt on a p-state, i.e., we have $proven_{evt,p} = \bigwedge_{i \in J} p_i$ for some $J \subseteq I$ such that*

$$\forall x, x'. \, p(x) \wedge BA_{evt}(x, x') \Rightarrow proven_{evt,p}(x').$$

Let $unproven_{evt,p} = \bigwedge_{i \in I \setminus J} p_i$ denote the complement of $proven_{evt,p}$, i. e., all the conjuncts of p that are not in $proven_{evt,p}$. We also define $proven_{evt} = proven_{evt,Inv}$ and $unproven_{evt} = unproven_{evt,Inv}$.

For the example in Fig. 1, we have $BA_{incby}(c, c') = \exists i \in 1..64 \wedge c' = c + i$. Furthermore, $proven_{incby}(c) = c \geq 0$; the invariant $c \geq 0$ is preserved by *incby*. This implies $unproven_{incby}(c) = c \leq m$.

In our current implementation, we have that $proven_{evt,p} = \bigwedge_{j \in J} p_j$ with $J \subseteq I$. This however is not a strict limitation. One could add other predicates discovered to be implied to $proven_{evt,p}$ so as to further strengthen the predicates given below.

Lemma 1. *A valid solution for Definition 2 is always $proven_{evt,p} = true$, meaning that nothing is proven for the event evt. At the other extreme, if all conjuncts of p are proven to hold after the execution of evt then $proven_{evt,p} = p$ and $unproven_{evt,p} = true$.*

Definition 3. *By T we refer to a monolithic transition predicate, i. e., the disjunction of all before-after-predicates: $T(x, x') = \bigvee_{e \in Events} BA_e(x, x')$. By I we denote the after predicate of the initialization; including the properties about the constants.*

For Fig. 1 we have $I(c) = m \in \{127, 255\} \wedge c = 0$.

In the following sections, we show how proof information can be embedded in the queries of bounded model checking (Section 2.1), k-Induction based model checking (Section 2.2) and IC3 (Section 2.3). An empirical evaluation of the algorithms and the influence of using proof information will be performed in Section 3.

2.1 BMC — Bounded Model Checking

BMC [5] has been suggested by Armin Biere et. al. in 1999 [6]. One of the main goals is to avoid the blowup and resulting slow down of BDDs-based model checking algorithms. This is achieved by replacing the BDDs by a SAT solver.

The basic idea is as follows: For an initial state relation I, a transition relation T and a property p and a bound k starting with $k = 0$, a sequence of propositional formulas is generated. Each of the formulas is satisfiable if and only if there exists a counterexample to the property with length $\leq k$. This can be expressed as:

$$BMC(p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

In order to include proof information we have to rewrite the predicate. First of all, if we increase k step-by-step as done in ProB's implementation of BMC, it is sufficient to check only the last state for a violation of p:

$$BMC(p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k)$$

For the example machine in Fig. 1, we have:

$$\begin{aligned}
BMC(Inv,0) = &\quad m \in \{127,255\} \wedge c_0 = 0 \wedge \neg(c_0 \geq 0 \wedge c_0 \leq m) &(1)\\
BMC(Inv,1) = &\quad m \in \{127,255\} \wedge c_0 = 0 \wedge \exists i.(i \in 1..64 \wedge c_1 = c_0 + i) \\
&\quad \wedge\neg(c_1 \geq 0 \wedge c_1 \leq m) &(2)\\
BMC(Inv,2) = &\quad m \in \{127,255\} \wedge c_0 = 0 \wedge \exists i.(i \in 1..64 \wedge c_1 = c_0 + i) \\
&\quad \wedge\exists i.(i \in 1..64 \wedge c_2 = c_1 + i) \wedge \neg(c_2 \geq 0 \wedge c_2 \leq m) &(3)
\end{aligned}$$

ProB's constraint solver finds no solution for Eqs. (1) and (2), but does so for Eq. (3): $m = 127, c_0 = 0, c_1 = 64, c_2 = 128$. One can see that the constraint solver has instantiated the parameter of the event in such a way as to violate the invariant. ProB's classical model checker on the other hand "blindly" enumerates all 64 possible successor states. Using breadth-first search, the counterexample is found after having generated 325 states and 12420 transitions; taking $\sim 1.5s$ whereas BMC finds the counterexample with $k = 2$, i.e., after three calls to the constraint solver and $\sim 1s$. A depth-first search may generate a long counterexample of up to 127 steps, depending in which order the successors are processed. ProB in this case actually processes the successors with the larger i values first; leading to a counterexample of length 4 after generating 324 states and 323 transitions. The state space is shown in Fig. 2, the corresponding counterexample is shown in Fig. 3. The larger the branching-factor, the better BMC becomes as compared to explicit state model checking. When the number

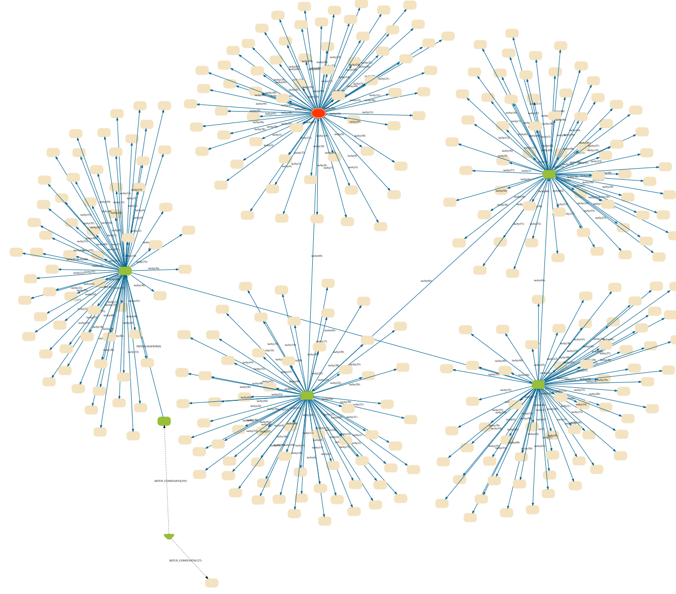

Fig. 2: State space of explicit state model checking

Fig. 3: Counterexample found by BMC

of possible parameter values becomes unbounded, e.g., supposing the incby event had no upper bound on i, BMC is often the only practical solution.[3]

Next, we extend the transition relation to either assert a property after every step or assert its negation:

Definition 4. *For a predicate p we define T_p and $T_p^{\neg}$ by*

$$T_p(x, x') = \bigvee_{e \in Events} (BA_e(x, x') \wedge p(x'))$$

$$T_p^{\neg}(x, x') = \bigvee_{e \in Events} (BA_e(x, x') \wedge proven_{evt,p}(x') \wedge \neg unproven_{evt,p}(x'))$$

For $k \geq 1$, the proven conjuncts of p can be used to strengthen the constraint:

[3] ProB gives the user the opportunity to set an upper-bound on the number of successor states per event for the explicit model checker; exhaustive model checking is then not possible but counterexamples can still be found.

$$BMC(p,k) = I(s_0) \wedge p(s_0) \wedge \bigwedge_{i=0}^{k-2} T_p(s_i, s_{i+1}) \wedge T_p^{\neg}(s_{k-1}, s_k) \tag{4}$$

For the example machine in Fig. 1, we have that for $k = 0$ the constraint remains unchanged, but for $k = 1$ and $k = 2$ we obtain:

$$\begin{aligned}
BMC(Inv, 0) = & \quad m \in \{127, 255\} \wedge c_0 = 0 \wedge \neg(c_0 \leq m) \\
BMC(Inv, 1) = & \quad m \in \{127, 255\} \wedge c_0 = 0 \\
& \wedge \exists i.(i \in 1..64 \wedge c_1 = c_0 + i) \wedge c_1 \geq 0 \wedge \neg(c_1 \leq m) \\
BMC(Inv, 2) = & \quad m \in \{127, 255\} \wedge c_0 = 0 \\
& \wedge \exists i.(i \in 1..64 \wedge c_1 = c_0 + i) \wedge c_1 \geq 0 \wedge c_1 \leq m \\
& \wedge \exists i.(i \in 1..64 \wedge c_2 = c_1 + i) \wedge c_2 \geq 0 \wedge \neg(c_2 \leq m)
\end{aligned}$$

Remember that $unproven_{evt,p}(s_k)$ evaluates to true if all conjuncts of p have been proven to hold after the execution of *evt*. Hence, for completely proven events $\neg unproven_{evt,p}(s_k)$ is false and the corresponding disjunct in $T_p^{\neg}$ is obviously unsatisfiable. However, we can not remove such completely proven events from the first $k-1$ steps as they might contribute to the path to a violation of p, using another final event.

Another BMC approach is to use the test-case generation algorithm from [27], using $\neg p$ as target predicate. In contrast to the BMC technique above, the transition predicate is not monolithic, and the algorithm builds up a tree of feasible paths. We have extended the algorithm from [27] to also use $\neg unproven_{evt,p}$ instead of $\neg p$, where *evt* is the last event of any given path. The algorithm optionally uses a static enabling analysis to filter out infeasible paths before calling the solver. In the remainder of the paper we refer to this algorithm as BMC*.

2.2 k-Induction

k-Induction [29] is a mixture of BMC and proof by induction. For the method to be complete, one has to avoid getting stuck in loops. Hence, the constraints are strengthened to avoid a state occurring twice on a given path. [4]

The base condition is encoded in Eq. (5); it is basically a BMC step and tries to find a counterexample of length k starting from the initialization. Like in Section 2.1 we assume that we gradually increase the value of k starting from 0, as shown in Algorithm 1. The inductive step, including the uniqueness of states, is expressed in Eq. (6), where *Axm* are the axioms on the constants of the model (e.g., $m \in \{127, 255\}$ in our running example).

[4] We could have added theses constraints $s_i \neq s_j$ also in Section 2.1.

```
Data: Property P
Result: true iff P holds
1 procedure boolean k-induction(P)
2     k := 0
3     while true do
4         if Base(P, k) satisfiable then return false
5         elsif Step(P, k) unsatisfiable then return true
6         else k := k + 1 end
7     end
```

Algorithm 1: k-Induction

$$Base(p, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k) \quad (5)$$

$$Step(p, k) = Axm \wedge \bigwedge_{0 \leq i < j \leq k} s_i \neq s_j \wedge \bigwedge_{i=0}^{k} T_(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} p(s_i) \wedge \neg p(s_{k+1}) \quad (6)$$

For $k = 0$ $Step(Inv, k)$ corresponds to trying to find counterexamples to the B invariant preservation proof obligations. In a similar fashion, $Base(Inv, 0)$ corresponds to finding initial states which violate the invariant. Hence, if $Base(p, 0)$ and $Step(T, p, 0)$ are unsatisfiable, we have found an inductive proof of the property p. However, the difference with B's approach to proving invariants does appear when $Step(T, p, 0)$ is satisfiable, i.e., there exists a state which satisfies p and a successor state violates p. The k-induction method tries to construct a real counterexample, starting from a valid initial state, not from *any* state satisfying p. Hence, the value of k is now increased and we try to find a real counterexample of length $k + 1$ using the BMC constraint Eq. (5).

Compared to BMC, k-Induction has the advantage of including an explicit termination condition. Suppose for example we take for p the predicate $c \geq -2 \wedge c \neq -1$ for Fig. 1. In this case BMC will never terminate, as for every value of k no counterexample can be found. k-Induction, however, can already stop with $k = 1$, as $Step(Inv, 1)$ is unsatisfiable. This is an interesting result, given that the state space of the model is infinite. The constraints are shown in Fig. 4 and are unsatisfiable except for $Step(Inv, 0)$. $Step(Inv, 0)$ corresponds to the B proof obligation for the event, checking that p is inductive. Hence, the B proof method is not able to prove that $c \geq -2 \wedge c \neq -1$ always holds. (A user would need to find an inductive invariant such as $c \geq 0$ implying the property. The IC3 algorithm in the next section will do just that automatically.)

The *Base* constraint is equal to the one in BMC: $Base(p, k) = BMC(p, k)$. Hence, we can include proof information in the same fashion and simply reuse the optimized constraint Eq. (4) from Section 2.1. For the inductive step, we can again use $T_p^{\neg}$ for the last step, to only look for violations of *unproven* parts of p. Following Algorithm 1, we also know that all intermediate states must

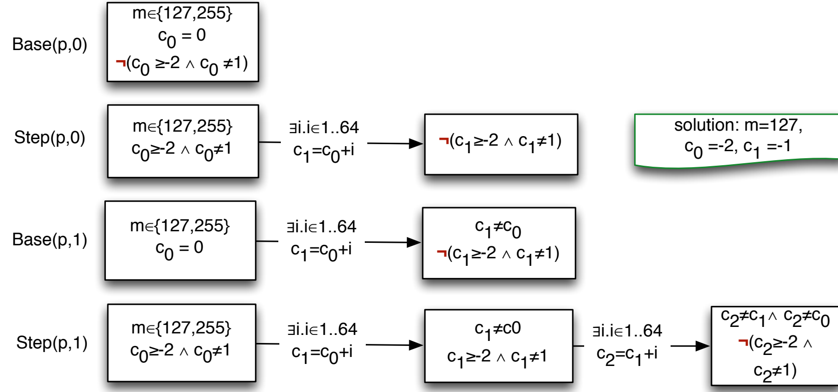

Fig. 4: Steps of k-Induction Algorithm 1 for Fig. 1 with property $c \geq -2 \wedge c \neq -1$

satisfy the property p; this we can encode using T_p, leading to the definition in Eq. (7). As in BMC, we can not remove before-after-predicates from the first steps. Constraints are simplified but the search space is not reduced.

$$Step(p,k) = Axm \wedge p(s_0) \wedge \bigwedge_{0 \leq i < j \leq k} s_i \neq s_j \wedge \bigwedge_{i=0}^{k-1} T_p(s_i, s_{i+1}) \wedge T_p^{\neg}(s_k, s_{k+1}) \tag{7}$$

2.3 IC3

In contrast to BMC presented in Section 2.1 and k-Induction presented in Section 2.2 the IC3 algorithm does not use an unwinding $(\bigwedge_{i=0}^{k} T(s_i, s_{i+1}))$ of the transition system. Instead, only single step queries are performed.

In order to verify a system, IC3 tries to *automatically* find an inductive invariant implying the property in question. To do so, it keeps a list of *frames* F_i over-approximating the set of states reachable in $\leq i$ steps. Counterexamples reachable in one or two steps are handled as a special case, as shown in line 2 of Algorithm 2. Afterwards, for each level k IC3 tries to find a property violation in a single step, i. e., a solution to $F_k \wedge T \wedge \neg p$.

If no solution exists, k is incremented and a new frame holding p is added. Otherwise, IC3 tries to show that the faulty state is in fact not reachable from the initialization. This is done by incrementally strengthening frames until F_k becomes strong enough to prevent the property violation from occurring. A partial outline is shown in procedure *strengthen* in Algorithm 2. The *Counterexample* exception is thrown by *inductivelyGeneralize* if generalization fails and the counterexample can not be proven spurious.

```
    Data: Property p, Transition predicate T, Initial state predicate I
    Result: true iff p holds
 1  procedure boolean ic3(p, T, I)
 2      if sat(I(s_0) ∧ ¬p(s_0)) ∨ sat(I(s_0) ∧ T(s_0, s_1) ∧ ¬p(s_1)) then return false
        end
 3      F_0 := I, clauses(F_0) := ∅
 4      F_1 := p, clauses(F_1) := ∅
 5      k := 1
 6      while true do
 7          if not strengthen(k, P, T) then return false end
 8          propagate_clauses(k)
 9          if ∃i ∈ [1, k]: clauses(F_i) = clauses(F_{i+1}) then return true end
10          k := k + 1
11      end
```

Algorithm 2: IC3: Main Loop

```
 1  procedure boolean strengthen(k, p, T)
 2      try
 3          while sat(F_k(s) ∧ T(s, s') ∧ ¬p(s')) do
 4              s := the predecessor extracted from the witness
 5              n := inductivelyGeneralize(s, k − 2, k)
 6              pushGeneralization((n + 1, s), k)
 7          end
 8          return true
 9      catch Counterexample
10          return false
```

Algorithm 3: IC3: Strengthen

Afterwards, a new counterexample might be found and IC3 will start to iterate between finding counterexamples and strengthening frames. If strengthening the frames eventually fails, a counterexample to the property is found. Otherwise, an inductive invariant has been found.

In the following, we will only go into details of IC3 wherever proof information can be incorporated. For a complete overview, see Bradley's original paper [9] or the one by Een, et. al. in [13]. Algorithms 2 and 3 follow the implementation of [9].

The first change to incorporate proof support takes place in the main loop of IC3. When implemented as suggested by Bradley in [9], IC3 features a special case for 0-step and 1-step reachability of a property violation as explained above. This is shown in line 2 of Algorithm 2. The query on line 2 can be changed in the same way we did for BMC and k-Induction. After splitting the transition relation and adding proof information we obtain:

$$sat(I(s_0) \land \neg p(s_0)) \lor sat(I(s_0) \land T_p^{\neg}(s_0, s_1))$$

Table 1: Runtimes (in seconds) and Speedup (in percent)

Model	MC		BMC		BMC*		k-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes
Models with Invariant Violations										
LargeBranching	-	-	1.18	0.99 (16.1%)	0.97	0.98 (-1.03%)	-	1.1 (∞)	1.0	1.0 (0.0%)
Search	-	-	-	-	-	-	-	-	-	-
SearchEvents	-	-	1.12	1.12 (0.0%)	1.08	1.05 (2.78%)	-	-	1.06	1.07 (-0.94%)
TravelAgency	1.11	1.09 (1.8%)	-	-	32.18	21.64 (32.75%)	-	-	-	-
CountersWrong	0.83	0.83 (0.0%)	0.9	0.96 (-6.67%)	0.94	0.88 (6.38%)	0.95	0.99 (-4.21%)	0.97	0.96 (1.03%)
Correct Models										
Coloring	-	-	-	-	-	-	-	1.22 (∞)	1.44	1.44 (0.0%)
Counters	-	-	-	-	-	-	1.0	0.92 (8.0%)	0.87	0.87 (0.0%)
f_m0	0.82	0.88 (-7.32%)	-	-	-	-	-	-	0.84	0.83 (1.19%)
f_m1	0.81	0.8 (1.23%)	-	-	-	-	-	-	0.9	0.89 (1.11%)
R0_GearDoor	0.78	0.79 (-1.28%)	-	-	-	-	0.96	0.86 (10.42%)	0.94	0.91 (3.19%)
R1_Valve	0.92	0.92 (0.0%)	-	-	-	-	7.24	0.85 (88.26%)	1.01	0.96 (4.95%)
R2_Outputs	1.84	1.78 (3.26%)	-	-	-	-	0.84	0.9 (-7.14%)	0.89	0.9 (-1.12%)
R3_Sensors	3.06	2.85 (6.86%)	-	-	-	-	-	0.93 (∞)	1.21	0.96 (20.66%)
R4_Handle	33.19	27.61 (16.81%)	-	-	-	-	-	-	-	-

The key point where adding proof assistance improves the performance however is inside the *strengthen* procedure of IC3. The original version is given in Algorithm 3. Inside, the algorithm tries to find a state included in F_k that has a successor violating the property. With the usual transformation, $F_k(s) \wedge T(s, s') \wedge \neg p(s')$ is transformed into $F_k(s) \wedge T_p^{\neg}(s, s')$.

In addition to simplifying the query itself, we can provide the sub routines *inductivelyGeneralize* and *pushGeneralization* with the event that lead to the violating state. This enables simplifying the respective predicates considerably.

In IC3, adding proof information has more benefits than just simplifying the occurring constraints. Due to the one-step nature of queries, constraint solving can be skipped altogether if $unproven_{evt,p} = true$. As no paths are built up explicitly, fully proven events have to be considered only during strengthening. They can safely be omitted during the counterexample search. Thus, including proof information leads to a reduction of the search space.

3 Empirical Results

The four algorithms described above have been implemented and are available in the nightly builds of ProB[5]. For the empirical evaluation we want to focus on two questions:

- Does the usage of proof information considerably improve the performance of symbolic model checking algorithms for B and Event-B?

[5] Available at `http://stups.hhu.de/ProB`. Information on how to use the new algorithms can be found on the ProB wiki: For the BMC* algorithm see `http://stups.hhu.de/ProB/Bounded_Model_Checking`. The other algorithms are documented at `http://stups.hhu.de/ProB/Symbolic_Model_Checking`.

– Can symbolic model checking algorithms compete with explicit state model checking (MC) as done by ProB?

We apply both the algorithms introduced in Section 2 as well as ProB's explicit state model checker (MC) to a selection of models, including artificial and real benchmarks. We use the explicit state model checker with and without proof support as outlined in [4]. The following models were used:

– *LargeBranching*, a crafted benchmark featuring a counterexample reachable in two steps. However, the initialization has numerous outgoing edges. Discovering the counterexample thus heavily relies on picking the right transitions to follow. The model is included to show that the symbolic algorithms are not influenced by this fact.
– *Search*, a classical B model of a binary search algorithm. *SearchEvents* models the same algorithm, but is written in Event-B style with simpler events. While this leads to simpler constraints, it increases the number of conjuncts due to the increased number of events.
– *TravelAgency*, a classical B model of a travel agency system storing and managing car and room rentals. The model includes an invariant violation.
– *Coloring*, a model of a graph coloring algorithm by Andriamiarina and Méry. In this particular model, the algorithm works on a concrete graph of 40 nodes.
– *f_m0* and *f_m1*, two hybrid models taken from [2].
– *Counters(Wrong)*, two artificial benchmarks featuring two independent counters, one of them bounded and one counting up infinitely. Both models feature an infinite state space. *CountersWrong* has a finite counterexample.
– *R0_Gear_Door*, *R1_Valve*, *R2_Outputs*, *R3_Sensors* and *R4_Handle* are the first refinement levels of our model [15] for the ABZ 2014 landing gear case study [8].

All benchmarks were run on a MacBook Pro featuring a 2.6 GHz i7 CPU and 8 GB of RAM. We did not run anything in parallel in order to avoid issues due to hyper-threading or scheduling. For each benchmark, a number of different results can occur:

– *verified*, i.e., the model could be model checked exhaustively without an invariant violation being detected.
– *counterexample found*, i.e., a state violating the invariant was found in the model and a trace to it has been computed.
– *incomplete*, i.e., no invariant violation has been found but model checking was not exhaustive. This could be due to timeouts or due to ProB being unable to solve occurring constraints. Currently, we do not try to recover. In case of BMC or k-Induction one could for instance try to increase k anyway.

The results are given in Table 1 showing the runtimes on successful benchmarks as well as the speedup achieved by using proof information.

The state space of the *Search* model is too large to be traversed by ProB's explicit state model checker. Unfortunately, the involved substitutions result in complex constraints that cannot be checked by the symbolic algorithms. The

effect is increased by the unwinding of the transition system, as complicated constraints start to occur multiple times.

The *SearchEvents* model features simpler substitutions and is thus more suited for symbolic analysis. Using proof information, all symbolic algorithms are able to find the counterexample. Without proof information, k-Induction is not able to check the model anymore. *LargeBranching* paints a similar picture.

The *TravelAgency* model on the other hand has a relatively small state space and can easily be verified using MC. However, it features involved constructs like sequences resulting in complicated constraints. BMC* is the only symbolic technique to find the counterexample, albeit taking much longer than MC.

The *Coloring* model is quite big and can not be checked exhaustively by MC in the given time. Only IC3 and k-Induction with proof information are able to do so. For IC3 this is due to its focus on one step reachability in combination with the model being correct: Only a small amount of counterexample candidates are discovered by IC3 and are immediately detected as spurious.

Abrial's hybrid models can be verified by MC and IC3. Here, constraints become considerably more involved with each unwinding of the transition relation done in BMC and k-Induction. IC3 is again able to verify the model thanks to its local search for counterexamples.

The infinite counters show one of the key limitations of explicit state model checking. Once a state space is infinite, exhaustive analysis is obviously impossible. For the correct model, BMC reaches its iteration limit without detecting an error. Both k-Induction and IC3 are able to analyze the models.

The landing gear model shows that the benefit of using proof information increases with the complexity of the model. As can be seen in Table 1 computation times go down once proof information is used. As for *SearchEvents* and *LargeBranching*, for *R3_Handle* k-Induction can only successfully be used if proof information is considered. For the first refinement steps, IC3 is quicker than explicit state model checking with ProB. However, once the fourth refinement level is reached, none of the symbolic algorithms can handle the occurring constraints anymore.

Regarding speedup, we can report from $\sim 7\%$ (*CountersWrong* with BMC) up to $\sim 88\%$ (*R1_Valve* with k-Induction). For most of the models, incorporating proof information leads to a speedup. Using IC3, our approach leads to a performance decline for some models. We suspect it is because adding additional constraints is not necessarily beneficial for a constraint solver.

Summarizing, we can answer the two questions stated at the beginning:

- The inclusion of proof information into the symbolic model checking algorithms does improve the performance most of the time. Furthermore, some models can only be checked if proof information is used.
- For some, albeit small, models symbolic techniques can compete with explicit state model checking. Symbolic model checkers allow to verify infinite state spaces which are beyond the scope of ProB's classical model checker.
- Among the symbolic techniques, BMC* was the best for erroneous models, while IC3 was best for correct models.

– However, existing solvers for B and Event-B are still to weak to handle the constraints occurring in larger or more involved models. This currently hinders symbolic model checking efforts.

4 Discussion, Related Work and Conclusion

In [4] the authors presented a similar integration of proof information into explicit state model checking algorithms. As is the case with our implementation, the authors report a speedup by not checking invariants known to be true. In contrast to our approach, the use of proof information never slowed down the model checking process.

Compared with [4], we have added a way to construct proof information within ProB itself, using a bridge to the Atelier B provers and using ProB's proving capabilities [18]. Of course this takes time and does not always pay off.

In [4], as with BMC and k-Induction, the search space itself is never reduced. Search space reduction through using proof techniques is considered in [28] and [24]. For model checking CTL and LTL properties, proof information can be used as well. In [26] the model checker SMV is coupled with theorem proving techniques. In a similar fashion, [3] combines the Alloy Analyzer with the Athena theorem prover.

Instead of using theorem provers to support model checking, one can use model checkers for theorem proving. We have done so using ProB [22,18].

Our evaluation shows that using symbolic model checking techniques for B and Event-B models is beneficial: Several counterexamples could only be detected by the symbolic algorithms. Furthermore, some models could be model checked exhaustively. As already outlined in [14], symbolic techniques prove to be a valuable addition to explicit techniques. The techniques are actually also applicable to TLA^+, via ProB's translation from TLA^+ to B [16][6].

They key weakness of employing symbolic model checking techniques lies within the expressiveness of B and Event-B. Even though constraint solvers and SMT solvers have increased their efficiency by a huge margin, the constraints occurring during symbolic model checking of high-level languages like B are still too involved. Among other abstraction techniques, integrating static (proof) information into the constraints is one way to help. It brings down computation times and sometimes enables successful validation. We are also working on strengthening the underlying constraint solver, by integrating SMT solvers such as Z3 [19]. Still, more improvements need to be achieved until full symbolic verification of B and Event-B models becomes viable.

Regarding the different model checking algorithms, especially IC3 seems promising. In contrast to the other two algorithms, its focus on one step reachability keeps occurring constraints easier. This makes it more suited for symbolic model checking of high-level languages like B and Event-B. Additionally, the integration of proof information can lead to a reduced search space. As IC3 has

[6] For further information regarding TLA^+ support in ProB have a look at `http://stups.hhu.de/ProB/TLA`.

originally been developed for hardware model checking, it is not trivial to lift it to the software world. To do so, we would like to investigate IC3 for B together with abstraction techniques as introduced in [11] or [7].

Another direction of future work could be to generate missing proof obligations from the model checking run. Analyzing predicates that lead to a timeout one could find problematic properties and try to prove them externally or in an independent run. Once the constraint solver gets stuck we could ask an external solver[7] to proof or disprove further invariants. Afterwords, one could extend the set of properties under consideration.

In summary, we have implemented four symbolic model checking algorithms for B and Event-B and have shown how to integrate proof information to improve the algorithms' performance. Our evaluation shows that bounded model checking can effectively find counterexamples in models with very large branching factors and that IC3 is capable of automatically proving models with infinite state spaces correct. Further research is, however, needed to scale up the symbolic techniques to models with more involved events.

Acknowledgements We would like to thank the reviewers of ABZ'2016 for their useful feedback. We also thank Aymerick Savary for comments and ideas, in particular relating to BMC and test-case generation.

References

1. J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.
2. J.-R. Abrial, W. Su, and H. Zhu. Formalizing hybrid systems with Event-B. In *Proceedings ABZ'12*, LNCS 7316, pages 178–193. Springer, 2012.
3. K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. In R. Berghammer, B. Möller, and G. Struth, editors, *Relational and Kleene-Algebraic Methods in Computer Science*, volume 3051 of *LNCS*, pages 21–33. Springer, 2004.
4. J. Bendisposto and M. Leuschel. Proof assisted model checking for B. In *International Conference on Formal Engineering Methods*, ICFEM '09, pages 504–520, Berlin, Heidelberg, 2009. Springer.
5. A. Biere. Bounded model checking. In *Handbook of Satisfiability*, pages 457–481. 2009.
6. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
7. J. Birgmeier, A. Bradley, and G. Weissenbacher. Counterexample to induction-guided abstraction-refinement (ctigar). In A. Biere and R. Bloem, editors, *Computer Aided Verification*, volume 8559 of *LNCS*, pages 831–848. Springer, 2014.
8. F. Boniol and V. Wiels. The landing gear system case study. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 1–18. Springer, 2014.

[7] Like the Atelier B provers or the SMT solvers for Rodin.

9. A. R. Bradley. Sat-based model checking without unrolling. In R. Jhala and D. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
10. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
11. A. Cimatti and A. Griggio. Software model checking via IC3. In *Computer Aided Verification*, pages 277–293, 2012.
12. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 94, Part 2:130 – 143, 2014.
13. N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 125–134, Austin, TX, 2011. FMCAD Inc.
14. S. Hallerstede and M. Leuschel. Constraint-based deadlock checking of high-level specifications. *Theory and Practice of Logic Programming*, 11(4–5):767–782, 2011.
15. D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. Validation of the abz landing gear system using prob. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, CCIS 433, pages 66–79. Springer, 2014.
16. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM'2012*, LNCS 7321, pages 24–38. Springer, 2012.
17. D. Hansen and M. Leuschel. Translating B to TLA+ for validation with TLC. In *Proceedings ABZ'14*, LNCS 8477, pages 40–55, 2014.
18. S. Krings, J. Bendisposto, and M. Leuschel. From Failure to Proof: The ProB Disprover for B and Event-B. In *Proceedings SEFM'2015*, LNCS 9276. Springer, 2015.
19. S. Krings and M. Leuschel. SMT Solvers for Validation of B and Event-B models. In *Proceedings iFM'2016*, LNCS. Springer, 2016. To appear.
20. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME'03*, LNCS 2805, pages 855–874. Springer, 2003.
21. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, Feb. 2008.
22. O. Ligot, J. Bendisposto, and M. Leuschel. Debugging Event-B Models using the ProB Disprover Plug-in. *Proceedings AFADL'07*, Juni 2007.
23. P. Matos, B. Fischer, and J. Marques-Silva. A lazy unbounded model checker for Event-B. In K. Breitman and A. Cavalcanti, editors, *Formal Methods and Software Engineering*, volume 5885 of *LNCS*, pages 485–503. Springer, 2009.
24. O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In E. Brinksma, W. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 1–16. Springer, 1995.
25. D. Plagge and M. Leuschel. Validating B,Z and TLA+ using ProB and Kodkod. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *LNCS*, pages 372–386. Springer, 2012.
26. A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 82–97, London, UK, UK, 2001. Springer.
27. A. Savary, M. Frappier, and M. Leuschel. Model-based robustness testing in Event-B using mutation. In R. Calinescu and B. Rumpe, editors, *Proceedings SEFM'15*, LNCS 9276, pages 132–147, 2015.

28. N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR'00: Concurrency Theory*, number 1877 in LNCS, pages 1–16, State College, PA, aug 2000. Springer.
29. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In J. Hunt, WarrenA. and S. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 127–144. Springer, 2000.
30. J. Witulski and M. Leuschel. Checking computations of formal method tools - a secondary toolchain for prob. In *Proceedings of the 1st Workshop on Formal-IDE*, EPTCS 149. Electronic Proceedings in Theoretical Computer Science, 2014.
31. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA^{+} specifications. In L. Pierre and T. Kropf, editors, *Proceedings CHARME'99*, LNCS 1703, pages 54–66. Springer-Verlag, 1999.