Writing a Model Checker in 80 Days: Reusable Libraries and Custom Implementation

Jessica Petrasch[1], Jan-Hendrik Oepen[1], Sebastian Krings[1,2], and Moritz Gericke[1]

[1] Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{jessica.petrasch,jan-hendrik.oepen,moritz.gericke}@hhu.de
[2] Competence Center for Information Security
Niederrhein University of Applied Sciences
Webschulstr. 41, D-41065 Mönchengladbach
sebastian.krings@hs-niederrhein.de

Abstract. During a course on model checking we developed BMoth, a full-stack model checker for classical B, featuring both explicit-state and symbolic model checking. Given that we only had a single university term to finish the project, a particular focus was on reusing existing libraries to reduce implementation workload.
In the following, we report on a selection of reusable libraries, which can be combined into a prototypical model checker relatively easily. Additionally, we discuss where custom code depending on the specification language to be checked is needed and where further optimization can take place. To conclude, we compare to other model checkers for classical B.

Keywords: Model Checking, SMT Solving, Libraries, University Course

1 Introduction and Motivation

During a course on model checking we developed BMoth, a prototypical model checker for classical B [1]. BMoth is based on a translation to SMT-LIB, using the well-known SMT solver Z3 [12] as backend. Our model checker is mostly written in Java and a clean-room implementation not built upon other model checkers. This allows us to use BMoth as a secondary toolchain for other B method tools. BMoth's source code, benchmarks used in this article and further information can be found on `https://github.com/bmoth-mc`.

A typical university course on model checking features different techniques such as explicit-state and symbolic model checking as well as temporal model checking, e.g., using LTL formulas. All algorithms and techniques discussed in the lectures were supposed to be at least rudimentarily implemented by the students in the programming project, whereas parser and type checker were given by the tutors. To achieve the goal of developing the full model checker during a single university term, we focused on the integration of different publicly available general purpose libraries.

We believe that our selection of libraries and their composition can be useful to other developers and help developing model checkers for new languages. To do so, the following article

- introduces the B language and some of its key concepts,
- identifies several publicly available libraries that can be used to implement various parts of a typical model checker,
- highlights how these libraries can be integrated,
- exemplarily discusses language-dependent implementation on top, and
- evaluates BMoth by comparing to the state-of-the-art model checker ProB.

2 A Primer on the B-method

The formal specification language B and the corresponding development method called the B-method [1] follow the correct-by-construction approach. Their models consist of a set of machines. A machine consists of variable and type definitions together with a predicate describing the initial states. By defining machine operations, one is able to specify transitions between states. A machine operation has a unique name and consists of B substitutions [1] defining the machine state after its execution. An operation can have a precondition allowing or prohibiting execution based on the current state.

Transitions can be non-deterministic and might be nested. Furthermore, B features different constructs that are executed atomically to define operations, ranging from simple variable assignment to if-then-else and while loops as well as parallel execution. All of these are supported to be executed automatically.

To ensure correctness of a specification, the user can define machine invariants, i.e., safety properties that have to hold in every state. Inside predicates and expressions, one can make use of a multitude of high-level language constructs featuring arithmetics, set logic and set comprehensions, sequences, as well as existential and universal quantification.

Besides using data types explicitly provided by the B language, one can define custom types in the form of sets. A set is defined by a unique name and may be initialized by a finite enumeration of distinct elements. Sets not enumerated are called deferred sets which are assumed to be non-empty and finite.

Below, we focus on classical B [1], but our approach also works for Event-B [2] and could be extended to other state-based specification languages such as TLA$^+$ [21]. While a simpler language than B would certainly be a more appropriate target for a university course, we decided to stick with what is used at our chair. In particular, this allows us to compare to our existing tools and to transfer knowledge gained while developing BMoth.

3 Libraries and Tools Used

As stated in the introduction, we focused on integrating different reusable libraries, avoiding to reinvent the wheel as much as possible. Below, we introduce the libraries and tools we used and discuss how they contribute to our model checker. Figure 1 gives an overview of the libraries and where they are employed.

3.1 ANTLR

ANTLR [30] is a parser generator getting a grammar as input. In our case, we were able to closely follow a subset of the B language grammar given by Abrial [1]. See Table 1 for a list of supported and unsupported parts of classical B. ANTLR then automatically generates code, in our case Java classes, for lexing and parsing using accepting finite-state machines.

While parser and lexer are generated automatically, the resulting AST is often too concrete and has to be restructured into a more abstract syntax tree. This is done manually using rewriting rules on top of the generated classes. Details are given in subsection 4.1.

3.2 JGraphT

JGraphT [29] handles storage of the state space used for explicit-state and LTL model checking. As the state space is just a directed graph, using a general purpose graph library seems appropriate. Furthermore, this allowed us to use efficient (graph focused) algorithms provided by JGraphT for finding shortest paths between two nodes, e.g. an initialisation node and a node occurring after a few steps, and for finding strongly connected components in the state space.

3.3 Z3

We use the SMT solver Z3 [12] as a backend for checking the satisfiability and computing valuations of formulas. This is done both for computing states as well as for invariant checking. Furthermore, for LTL model checking, Z3 computes state transitions in a Büchi automaton. This will be discussed in subsection 4.3. We connect to Z3using its Java API.

Of course, Z3 and its input language SMT-LIB do not support all high-level constructs available in B. In consequence, we have to translate B to SMT-LIB. More concrete, we translate into a logic supporting quantifiers, integer arithmetic and set theory. While for most language features the translation can be implemented using a simple AST walker on top of ANTLR, translation is more complicated for others. As an example, we will look at the translation of exponentiation in subsection 4.1, where we will discuss different translation alternatives and their performance impact.

3.4 UI libraries

We decided to create a graphical user interface for more flexibility (especially while testing), as it allows the user to edit machines directly in BMoth. For this GUI we used JavaFX in combination with MvvmFX [9]. In particular, we follow the MVVM pattern. To create an efficient editor for B models we used RichTextFX [28], which also allows custom syntax highlighting.

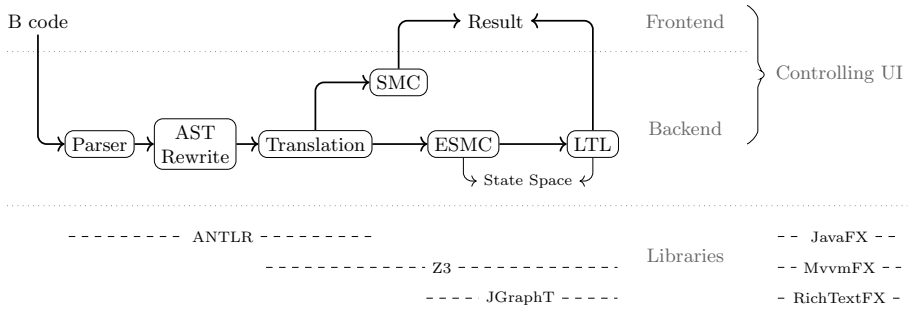

Fig. 1. Architecture overview and libraries used

3.5 Infrastructure

For source code management we worked with GitHub. Code quality and performance where surveyed and improved using the Travis CI [35] and SonarQube [34]. While not as crucial as using the right set of libraries, infrastructure had a tremendous impact on progress. While GitHub provided us with the means of remote synchronization and project management, Travis CI and SonarQube-helped us to detect bugs early and thus avoid spending time fixing them. Without the right infrastructure, our project would have been delayed way beyond the scope of a university term.

4 Implementation Details

In this section, we will discuss several parts of our implementation, focusing on the libraries used to realize different aspects of our model checker. In particular, we will discuss custom, e.g., language-dependent, implementation where needed.

4.1 Backend and Translation of B to Z3

A key question within the development process of BMoth was how to connect a suitable backend to the JVM-based parts of the project. Our approach consisted in using the Z3 solver, connected via the Z3 Java API for solving B formulas translated to SMT. In particular, we translate into a logic natively supporting sets and quantifiers.

Each input instance of BMoth contains a specification of a machine written in B and a property given as a B or LTL formula to be checked. As of course B and LTL cannot be directly processed by Z3, a translation process from B to an SMT-LIB constraint is needed. An overview of the full translation process is visualized in Figure 2.

If an input instance contains an LTL formula, it is normalized, reducing available features in order to simplify translation rules. LTL conversion is covered in

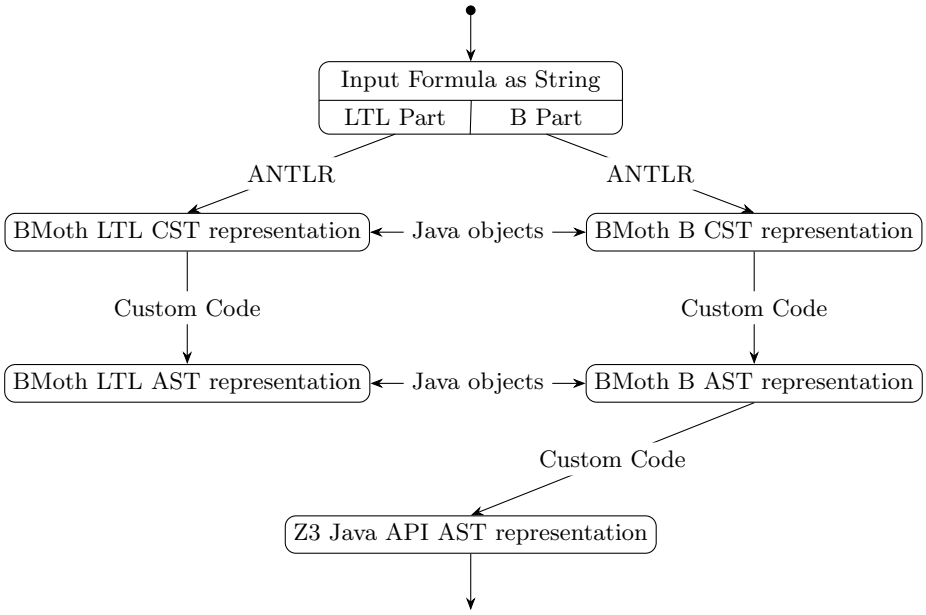

Fig. 2. Translation process from user input to SMT

more detail in subsection 4.3. An input instance may contain both LTL properties (given as separate formula) and B properties (as machine invariants or as separate formula), which will individually be checked by BMoth.

In order to be evaluated, a B predicate is parsed to a concrete syntax tree using the lexer and parser generated by ANTLR. Afterwards, the CST is simplified to an AST using common compiler construction techniques [4]. Following, we implemented a syntax-directed translation to SMT-LIB using Z3's Java API.

While the syntax-directed conversion visits each node of the AST and replaces it with an equivalent node of Z3's AST, in some cases additional constraints are generated, e.g., for well-definedness. Any additional constraints are conjugated appropriately after translation.

The approach chosen allows direct conversion of nodes in many cases, namely where BMoth's B node has a semantically equivalent node in the Z3 AST and no additional constraints are required. Examples are, among others, $=/2$, $\leq/2$, $+/2$, $-/2$, $-/1$, $\in/2$, $\subseteq/2$, $\cup/2$ and $\top/0$. A counterexample would be the division $:/2$, which requires an additional constraint, stating that the divisor may not be zero, in B. A significant number of node types can be transformed by simple additions or rewrites. Examples are B's $\neq/2$, which corresponds to Z3's $\neg(=/2)$, or $\subset/2$, which can be split into $\wedge(\neq/2, \subseteq/2)$.

The AST rewrites are easily implemented on top of the AST provided by ANTLR. Listing 1 shows the code used to rewrite $expr \in \mathbb{N}$ into $expr \geq 0$. Essentially, we just have to check whether the node to be rewritten has the $\in$

Listing 1. AST rewriting

```
if (node.getOperator() == ELEMENT_OF) {
    ExprNode left = node.getExpressionNodes().get(0);
    ExprNode right = node.getExpressionNodes().get(1);
    if (right instanceof OperatorNode
        && ((OperatorNode) right).getOperator() == NATURAL) {
        return new OperatorWithArgsNode(..., GREATER_EQUAL,
            Arrays.asList(left, new NumberNode(..., ZERO)));
        }
}
return node;
```

operator and whether the right argument is $\mathbb{N}$. If so, we return a new AST node with the target predicate, which is inserted into the AST replacing the old node.

Another interesting example illustrating the flexibility of our compositional approach is the exponentiation expression $**/2$ in BMoth: Rather than using Z3's integrated exponentiation, we can experiment with alternative encoding, e.g., using recursive functions. By doing so, we can force the solver to use distinct ways of evaluation. The potential of recursive definitions for Z3 via AST manipulation and the advantages of a flexible approach to work around weaknesses of the backend are described in detail later on.

Just like formulas, BMoth parses machines into an abstract syntax tree representation before further conversion. Afterwards, before-after-predicates are computed for all operations and Z3 representations of all occurring predicates are precomputed and stored.

Note that there is no closed representation of B's while loops in terms of a before-after-predicate. The definition given by Abrial [1] only states that the loop condition has to evaluate to false in the after state and that the invariant should evaluate to true. The effect of the loop however has to be computed using a fixpoint operation. In case of BMoth, this means that we cannot model check machines using the current evaluation architecture. However, one could think of a small-step semantic for B, in which the body of a while loop is computed individually until the loop condition eventually evaluates to false. This would enable explicit-state model checking using our architecture. For the symbolic model checking algorithms, other approaches are needed and should be part of future research both in BMoth and ProB.

Once all additional predicates are computed, all model checking algorithms solely rely on the SMT-LIB translation of predicates, while the B representation is only used for textual representation and user feedback.

As can be seen in Table 1, the list of language features implemented in BMoth only covers a subset of B. Of those not yet implemented, some can be considered variants of implemented features that they can be rewritten to to be expressed. For instance, relations can be expressed via set theory constraints, while the

Table 1. Overview of implemented and unimplemented features

	implemented	not implemented
comparison predicates	$=$, $<$, $>$, $\leq$, $\geq$, $\neq$	
arithmetics	$+$, *, -, /, %, $\hat{}$	
basic logical predicates	$\wedge$, $\vee$, true, false, $\forall$, $\exists$, $\Rightarrow$	
set predicates	$\in$, $\notin$, $\subseteq$, $\not\subseteq$, $\subset$, $\not\subset$, $\times$	
basic sets	deferred and enumerated sets, $\emptyset$, intervals, $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_+$, INT, NAT, NAT1, BOOL	
set operators	$\setminus$, $\cup$, $\cap$, $\bigcup$, domain, range, set enumeration, min, max	$\mathfrak{P}$, finite subsets, $\bigcap$, cardinality, set summation, set product
tuples/sequences	couples, enumerated sequences, empty sequence	
sequence operators	front, last, first	set of finite subsequences, permutations, concat, prepend, append, size, reverse, take, drop, tail, strings
relations	inverse	forward composition, backward composition, identity, relational image, override, domain/range restriction/subtraction
functions	function call	partial functions (and less general variants), lambda abstraction

missing kinds of functions can be constructed in a more general way as relations with additional constraints (consisting of already implemented features).

What remains among those features not implemented so far, but yet has to be taken care of in an efficient way, is the cardinality of sets. In principle, it could be dealt with by finding a bijection to an integer interval. However, that yields an expensive solution, which often does not perform well [19].

Custom Encodings The system design and selection of libraries we presented facilitates experimenting with different encodings and other optimization techniques. To give an example, we discuss one of the problems we encountered developing BMoth: Due to the undecidability of the underlying logic, Z3 performed comparably poor when confronted with a combination of quantifiers and non-linear arithmetic (in particular exponentiation). While both individually could often be solved, test cases combining both usually resulted in the constraints being unsolvable for Z3. We investigated different possible solutions:

- use other backends, possibly in parallel with Z3,
- changing the way BMoth handles quantified formulas, e.g., try unrolling them where possible,
- changing the way BMoth handles exponentiation, and

– search for ways to alter Z3's behavior by configuration of Z3.

Since the undecidability limits Z3's support for non-linear integer arithmetic combined with quantifiers, relying on a different backend might have helped. However, additional solvers would have been a rather severe change in design, quite impossible to perform within our time restrictions. We thus decided to evaluate the other approaches first.

In comparison, examining various encodings of exponentiation was a comparatively lightweight solution and thus appeared more promising. Our alternative approach to Z3's internal exponentiation consists in translating the exponentiation nodes of the internal B AST to a recursive function. More concretely, exponentiation is realized using the well-known square-and-multiply method[3].

Square-and-multiply is introduced in Z3 by defining a function and recursively assigning a value to certain invocations by means of universal quantification. In particular, the special cases x^0 and x^1 are used as termination cases of the recursion. The recursive cases implement the square-and-multiply pattern.

Of course this alternative encoding does not help Z3 proving formulas involving exponentiation in every case, but some of the problematic combinations ceased posing an obstacle to checking satisfiability and finding a valuation. For certain models, the approach was able to compensate for Z3's inability to solve combinations of quantifiers and non-linear arithmetic, showing a notable advantage of the flexible architecture used in BMoth.

4.2 Explicit-State Model Checker

BMoth's explicit-state model checker (ESMC) works on a state space, a directed graph made up of the possible states of the system. These states are successively checked for invariant violations. To build this state space we use Z3. For each new state found by evaluating the conjunction of an already discovered state and the before-after-predicate, a new vertex is added to JGraphT's representation of the state space.

The basic explicit-state model checking algorithm [11] is simple: Determine the initial states by finding all solutions to the initial state predicate using Z3, put them in a queue, check each one for invariant violations, add all successors to the queue and repeat the last two steps until the queue is empty. In case of a violation the found counterexample is returned and presented to the user together with the path to the offending state. If desired, we could return the shortest path (so far) using the implementation of Dijkstra's algorithm in JGraphT. If we encounter undecidability because of used quantifiers, ESMC may run into a timeout. It then returns a partial state space and the result *Unknown*.

To improve the efficiency of the implementation BMoth aggregates different before-after-predicates disjunctively for finding successors. Further optimization in the future could include the aggregation of states.

[3] Square-and-multiply significantly reduces the number of multiplications, thus improving performance when computing on given values as done for computing successor states.

Listing 2. State space cycle detection using JGraphT

```
if (lgba != null) {
    labelStateSpace(); // assigns LGBA nodes to state nodes
    List<List<State>> cycles = new
        TarjanSimpleCycles<>(stateSpace).findSimpleCycles();
    for (List<State> cycle : cycles) {
        for (State state : cycle) {
            if (lgba.isAcceptingSet(state.getBuechiNodes())) {
                return LTLCounterExample(state);
            }
        }
    }
}
return verified(stateSpace);
```

4.3 LTL Model Checker

So far, we only discussed how to implement checking of safety properties, i.e., the absence of faulty states. However, to be considered a fully fledged model checker, BMoth has to also process liveness properties.

As LTL is more common in the B community, we decided to implement an LTL rather than a CTL model checker. To do so, we follow an automaton-based approach, i.e., the formula in question is negated and transformed into a *Labeled generalized Büchi automaton* (LGBA) [7] using the algorithm suggested by Gerth et al. [16]. Afterwards, BMoth searches for a counterexample by combining the automaton with the model's state space. As BMoth executes the LTL Model Checking subsequent to ESMC, we can reuse the same state space.

For BMoth, LTL formulae over a set AP of atomic propositions are formed according to the basic grammar

$$\phi ::= true \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{X}\phi \mid \phi\, \mathcal{U}\, \phi$$

for $a \in AP$, together with the LTL operators $\Diamond$ (finally), $\Box$ (globally), $\mathcal{W}$ (weak-until) and $\mathcal{R}$ (release), which are all derived from $\mathcal{X}$ and $\mathcal{U}$. As of now, BMoth does not support past-LTL. However, this could be added using rewriting rules (increasing the size of the formulas to check [22]).

As discussed for basic B predicates, we use ANTLR for parsing and rewriting of LTL formulae. The algorithm introduced by Gerth et al. [16] can only process the negation normal form, which just allows the operators $\wedge, \vee, \mathcal{X}, \mathcal{U}, \mathcal{R}$ and $\neg$. Furthermore negations may only appear directly in front of a predicate. In consequence, the input formula has to be normalized, again following the patterns introduced for the normalization of a regular B predicate.

BMoth searches for counterexamples differently than proposed by Gerth et al. [16]. Instead of checking the product automaton of state space and LGBA for

emptiness, BMoth finds loops in the state space that are accepted in the LGBA. This is realized by assigning each state of the state space the LGBA nodes the model can be in in this state. Initial state space states are assigned the initial LGBA nodes. Next, successors are assigned LGBA nodes recursively. Based on the assigned LGBA nodes of the current state we determine which LGBA nodes can be reached. For every successor in the state space we check whether the labels of all those reachable LGBA nodes do not contradict the propositions supposed to be true in the successor state. If Z3 states that labels and propositions match, that LGBA node is assigned to the successor. State space traversal is realized using the accessors provided for graph nodes by JGraphT.

As shown in Listing 2, BMoth uses *Tarjan's strongly connected components algorithm* provided by JGraphT to find loops in the state space created during ESMC. For each loop in the state space, BMoth checks whether there is a node from each accepting set of the LGBA assigned to the states in the loop. If so, a counterexample is found and the model does not satisfy the LTL property.

4.4 Symbolic Model Checker

Selecting states from the state space one by one can be very inefficient, particularly for systems with a large number of transitions. To counteract this inefficiency symbolic model checkers analyze multiple states at once. BMoth includes two symbolic model checking algorithms: BMC and k-induction.

Bounded Model Checker A bounded model checker looks for counterexamples that can be reached after a number k of steps, which implies that a counterexample that occurs later than after k steps will not be found unless the system is checked again with an increased k [5]. BMoth's BMC works the same way: sequentially increasing the bound from 1 to k, all transitions as well as the negated invariant are added to the constraint in question. The query is given to Z3 to check the satisfiability. If a valuation is found, the model is not correct and the path to the counterexample is extracted from the model and returned. To increase efficiency, we add a constraint enforcing states to be distinct, ensuring that new states are discovered with every increment of the bound. In future, this could be used to detect if the system has been checked exhaustively, i.e., if no new states can be computed.

Implementing BMC on top of our library selection is simple. The complete implementation is given in Listing 3. As you can see, the amount of custom code is quite small. Essentially, we combine Z3 (calls to the solver object) with the results of the AST transformations mentioned earlier. Most notably, `init`, `transition` and `negatedInvariant` return three predicates translated from the B machine to Z3's AST: the initial state predicate, the conjunction of the before-after-predicates and the negated invariant. Finally, `stateFromModel` accesses the valuation found by Z3 and extracts the values of the state variables.

Listing 3. Implementation of BMC

```
for (int k = 0; k < maxSteps; k++) {
    // get a clean solver
    solver.reset();

    // INIT(V0)
    solver.add(init());

    // CONJUNCTION i from 1 to k T(Vi-1, Vi)
    for (int i = 1; i <= k; i++) {
        solver.add(transition(i - 1, i));
    }

    // not INV(Vk)
    solver.add(negatedInvariant(k));

    // i from 1 to k, j from i + 1 to k (Vi != Vj)
    solver.add(distinctVectors(k));

    Status check = solver.check();
    if (check == Status.SATISFIABLE) {
        // counter example found!
        State ceState = stateFromModel(solver.getModel(), k);
        return createCounterExampleFound(k, ceState, null);
        }
}

// no counter example found after maxStep steps
return createExceededMaxSteps(maxSteps);
```

***K*-Induction Model Checker** BMC can only refute the correctness of a system by finding a counterexample in the first k steps. However, the absence of such a counterexample cannot be proven, i.e., BMC can never verify a system as correct unless there is some upper bound known for k [10].

This limitation can be avoided by employing induction in the model checker, leading to an algorithm called k-induction [33]. Essentially, k-induction uses mathematical induction to conclude the entire absence of an invariant violation from the absence of an invariant violation in the first k steps. The implementation uses an additional constraint and otherwise is similar to the one of BMC presented above.

5 Empirical Evaluation

To detect flaws and strengths of our implementation we compared it to ProB [23,25,24] in various benchmarks, mostly using examples provided with ProB. In addition

to ProB's explicit-state model checker, we compare to ProB's symbolic model checker introduced by Krings et al. [18,20]. Benchmarks were executed on a laptop running Ubuntu 16.04 and featuring a 2.4 GHz i7 CPU and 8 GB of RAM. Most of our benchmarks are comparably small. As BMoth does not support all of B (see subsection 4.1), there are not that many larger machines available. The used B machines can be assorted into the following classes:

- **Counters** Counters are suitable to investigate qualities like speed and the capability of a model checker to detect invariant violations in a potentially large (but linear) state space. Even though symbolic model checkers (especially BMC) often have trouble verifying these models because of the possibly tremendous length of paths to counterexamples, we were also curious to find out how long it would take them to examine the k first steps. Presented machines of this class are *SimpleCounter, DoubleAndHalve, FaultyCounter, GetToOneMillion* and *IncTwoDecTwo*.
- **Puzzles** Solving a puzzle that was translated to a B machine is done by finding an invariant violation in the initial state. As this happens very fast for the *SendMoreMoney* machine, model checking it to find a solution of the puzzle demonstrates how fast the model checkers kick off without actually having much to do.
- **Laws** *ArithmeticLaws* and *BooleanLaws* are two more extensive machines consisting of invariants stating basic laws of arithmetics and Boolean algebra.
- **Miscellaneous** Apart from these special machines we included simple models to get more results for basic machines containing violations. Presented here are *SetVarToConstant* (which assigns the given value of a constant to a state variable and thus violates the invariant) and *DecOneOfTwo*.

Results are given in Table 2. The average runtimes of explicit-state model checking (ESMC), bounded model checking (BMC) and k-induction applied to our examples are shown in Figure 3 for both BMoth and ProB. Only the executions leading to a clear result are considered, whereas those ending in a timeout or an exceeding of the maximum steps were left out of the calculation.

Summarizing, BMoth is not as fast as ProB. This is unsurprising, as we did focus on ease of implementation and code reuse rather than performance. Furthermore, ProB has been in development for more than 10 years and is thus more mature than our prototypical implementation. While the comparisons generally show that ProB's model checking algorithms are faster than our counterparts, especially for more extensive models, there are some interesting outcomes when looking at single results in Table 2.

5.1 Startup

For machines with small state spaces or counterexamples reachable within few steps, our implementation performed better than ProB. In contrast, for larger machines (e.g. see ESMC's results for *FaultyCounter* and *FiniteInfiniteMachine* in Table 2), BMoth took longer than ProB. We assume that BMoth's startup

Table 2. Runtimes in minutes and results - number of steps performed in parentheses

Machine	Value	ESMC		BMC		k-induction	
		BMoth	ProB	BMoth	ProB	BMoth	ProB
Simple Counter	Runtime	00:00.68	00:01.72	00:00.96	00:07.11	00:01.10	00:01.72
	Result	Verified (6)	Verified	Exceeded (20)	Timeout (13)	Verified (6)	Verified (0)
Faulty Counter	Runtime	01:18.81	00:02.91	00:01.13	00:05.77	00:01.27	07:27.97
	Result	CE found (5001)	CE found (5001)	Exceeded (20)	Timeout (9)	Exceeded (20)	Timeout (9)
Double And Halve	Runtime	00:00.64	00:01.65	00:03.31	00:08.37	00:05.74	00:03.58
	Result	Verified (9)	Verified	Exceeded (20)	Timeout (17)	Verified (14)	Verified (11)
Arith-metic Laws	Runtime	00:00.82	00:07.10	00:00.78	00:05.31	00:00.81	00:01.94
	Result	Unknown	Verified	Unknown	Timeout (6)	Unknown	Timeout (0)
Boolean Laws	Runtime	00:00.72	00:01.71	00:01.04	00:12.43	00:00.72	00:01.73
	Result	Verified (8)	Verified	Exceeded (20)	Exceeded (24)	Verified (0)	Verified (0)
GetTo One Million	Runtime	-	02:58.50	00:00.89	00:07.14	00:01.14	08:49.15
	Result	-	CE found (999999)	Exceeded (20)	Timeout (16)	Exceeded (20)	Timeout (16)
IncTwo DecTwo	Runtime	11:00.31	00:02.33	00:01.35	00:04.28	00:01.40	02:55.55
	Result	CE found (19819)	CE found (203)	Exceeded (20)	Timeout (6)	Exceeded (20)	Timeout (6)
SetVar ToCon-stant	Runtime	00:00.62	00:01.66	00:00.61	00:01.67	00:00.62	00:01.69
	Result	CE found (2)	CE found (2)	CE found (1)	CE found (1)	CE found (1)	CE found (1)
DecOne OfTwo	Runtime	00:00.64	00:01.67	00:00.64	00:01.70	00:00.63	00:01.74
	Result	CE found (2)	CE found (1)	CE found (1)	CE found (1)	CE found (1)	CE found (1)
Send More Money	Runtime	00:00.66	00:01.72	00:00.66	00:01.72	00:00.64	00:01.71
	Result	CE found (1)	CE found (0)	CE found (0)	CE found (0)	CE found (0)	CE found (0)

Fig. 3. Average runtime of model checking algorithms

is faster, while the actual model checking is quicker in ProB. The poor performance of BMoth's ESMC is mainly caused by the computation of successor states via Z3. The fast startup might be owing to a short parsing time. The average shares of parser and model checker in the total runtime of BMoth for some examples in Table 2 are depicted in Figure 4. ProB only reports the time consumed overall, we thus cannot compare the parsers individually.

5.2 Solver Limits

The backends of both tools were overstrained by different machines respectively when performing BMC and k-induction. Occasionally, ProB aborted the model check due to a timeout caused by an error that is described as "Solvers too weak for predicate". In these cases, BMoth could conduct the model check further and execute more steps (e.g. see *ArithmeticLaws* in Table 2).

While this did not lead to more results, as BMoth's BMC and k-induction still always reached the specified maximum number of steps to be executed without a falsification or verification, it shows that the backend of ProB has issues with certain machines that BMoth can handle. We assume that these differences are either due to different approaches to encoding or due to the synchronization of different backends in ProB. A more thorough analysis should be performed in future and might aid the development of both BMoth and ProB.

The other way around, we found machines troubling BMoth but not ProB. The explicit-state model checking could not be completed because of limitations of Z3, e.g. incomplete quantifiers when using exponentiation.

Some results are due to the behavior of ProB's k-induction in presence of timeouts. While BMC stops as soon as it runs into a timeout, k-induction continues, hoping for an inductivity to be proven eventually. This implies a long runtime (e.g. see *FaultyCounter* and *GetToOneMillion* in Table 2, where the

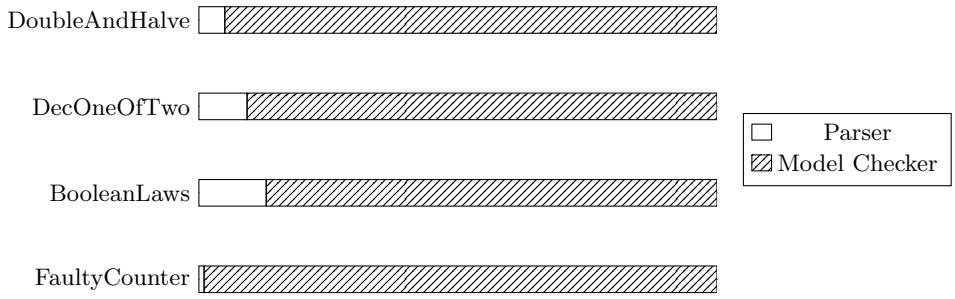

Fig. 4. Average shares of parser and model checker in the runtime of BMoth's ESMC

timeout was encountered after just a few steps, but the algorithm continued for minutes reaching a timeout again and again) and leads to the favorable comparison for BMoth in Figure 3.

Table 2 also contains the example *ArithmeticLaws* for which our model checker, contrary to ProB, could not come to a result. However, Figure 5 shows a machine that could be proven right by BMoth, but not by ProB. BMoth immediately realizes that the guard for the operation *op*, x < y & y < x, can not be satisfied, making the operation infeasible. It hence comes to the conclusion that v is never changed and particularly not set to 5, and that the machine is correct. ProB's Prolog backend on the other hand searches for an assignment for x and y that will satisfy the predicate until it runs into a timeout.

Altogether, the runtime results for BMoth were very satisfactory, given that we had no time for fine-tuning and further experimentation. Often enough our implementation was about as fast or even faster than ProB for small models.

```
MACHINE BMothWins
VARIABLES
  v
INVARIANT
  not(v = 5)
INITIALISATION
  v := 1
OPERATIONS
 op = ANY y,x WHERE x < y & y < x THEN v := 5 END
```

Fig. 5. Model where BMoth outperforms ProB

6 Related Work

Several model checkers for classical B and Event-B have been developed: ProB [23] features both explicit-state [25,24] and symbolic model checking [18,20]. Symbolic model checking in ProB uses the same algorithms implemented in BMoth. In contrast to BMoth, ProB supports different backends, such as SICStus Prolog's CLP(FD) library [8], Z3 [19] or SAT via Kodkod [32]. Furthermore, ProB's LTL model checker [31] uses a tableau based approach [26] rather than Büchi automata. ProB supports fairness constraints in LTL, which are currently unsupported in BMoth but could be implemented as done by Dobrikov et al. [14].

pyB [37] is another clean-room implementation of an explicit-state model checker for B. Originally, pyB was designed as a second toolchain to verify results generated by ProB. Furthermore, Yang et al. implemented JeB [38], an animation framework for Event-B written in JavaScript. While it does not yet reach the level of maturity of ProB, JeB certainly shows that JavaScript can be a viable alternative to Prolog, which is used in ProB's kernel. The research done in JeB could influence BMoth's further development, especially regarding alternative backends to overcome the limitations outlined in subsection 4.1.

Model checking aside, B and Event-B have been translated to SMT-LIB for other purposes. A translation from classical B to SMT-LIB has been suggested by Krings et al. [19] mostly aiming at (interactive) animation of B models, but also supporting proof. In addition, the authors also introduce a backend combining SMT solvers and constraint logic programming in the style of the CLP(FD) libraries of SICStus and SWI Prolog [36,8]. This approach could help overcome limitations of BMoth.

Event-B has been translated to SMT-LIB via the SMT plug-in [15,13] for Rodin [3]. The plugin is used inside Rodin's proving perspective. As B and TLA$^+$ [21] feature considerable parallels, the translation of TLA$^+$ [27] to SMT-LIB also influenced the translation approach employed in BMoth.

7 Discussion and Conclusion

In summary, we presented a selection of libraries and their combination into a model checker for classical B. Even though BMoth remains a student project and is merely a prototypical implementation, it shows that reinventing the wheel is as unnecessary for model checkers as it is in general software development.

Of course, we had to lower our sights regarding completeness and performance. Due to the short development period, BMoth does not support all features of classical B. However, as B is among the most high-level specification languages, we suppose a simpler language could have been implemented exhaustively. To gain more benchmarks, existing B specifications could be rewritten to constructs that BMoth supports. Regarding performance, our evaluations show that BMoth lacks the years of fine-tuning that went into ProB [23].

In consequence, we should reconsider BMoth's backend. While we are pleased with Z3's API and performance, its input language lacks the expressiveness

needed to translate B. This has been pointed out by Krings and Leuschel [19] and is a major area of work in the B community. We want to explore further backends regarding ease of integration and performance. In particular, following the spirit of code reuse, projects such as JavaSMT [17] and Why3 [6] can be the way to go. A backend-agnostic library, connecting to multiple solvers at once, would fit nicely. However, since we translate into Z33's set-theory for now, both approaches would not be a drop-in replacement. Additionally, we could integrate an interpreter for B and use it to compute transitions on given values, i.e., where no constraint solving is needed.

While BMoth cannot compete with ProB, it shows that developing a prototype for an experimental language can be done quickly, by an inexperienced development team. This does not only help language experimentation, but also might be useful to the development of other model checkers, as a prototype can serve as playground for techniques not easily implemented in an old codebase.

References

1. J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.
4. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.
5. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. In *Advances in Computers, Volume 58*. Academic Press, 2003.
6. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. `https://hal.inria.fr/hal-00790310`.
7. J. R. Büchi. Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic. In *Logic, Methodology and Philosophy of Science*, pages 1–11. Elsevier, 1966.
8. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, pages 191–206, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
9. A. Casall and M. Mauky. mvvmFX. `https://github.com/sialcasa/mvvmFX`.
10. E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and Complexity of Bounded Model Checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 85–96. Springer Berlin Heidelberg, 2004.
11. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
12. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

13. D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ*, volume 7316 of *LNCS*, pages 194–207. Springer, 2012.
14. I. Dobrikov, M. Leuschel, and D. Plagge. LTL Model Checking under Fairness in ProB . In R. De Nicola and E. Kühn, editors, *Software Engineering and Formal Methods*, pages 204–211, Cham, 2016. Springer International Publishing.
15. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 94, Part 2(0):130–143, 2014.
16. R. Gerth, D. Dolech, D. Peled, M. Vardi, and P. Wolper. Simple On-the-Fly Automatic Verification of Linear Temporal Logic. In P. Dembiński and M. Średniawa, editors, *Proceedings PSTV*, IFIPAICT, pages 3–18. Springer, 1996.
17. E. G. Karpenkov, K. Friedberger, and D. Beyer. JavaSMT: A Unified Interface for SMT Solvers in Java. In S. Blazy and M. Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 139–148, Cham, 2016. Springer.
18. S. Krings and M. Leuschel. Proof Assisted Symbolic Model Checking for B and Event-B. In *Proceedings ABZ*, volume 9675 of *LNCS*. Springer, 2016.
19. S. Krings and M. Leuschel. SMT Solvers for Validation of B and Event-B models. In *Proceedings iFM*, volume 9681 of *LNCS*. Springer, 2016.
20. S. Krings and M. Leuschel. Proof assisted bounded and unbounded symbolic model checking of software and system models. *Science of Computer Programming*, 2017.
21. L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.
22. F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal Logic with Forgettable Past. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 383–392, Washington, DC, USA, 2002. IEEE Computer Society.
23. M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, and D. Plagge. From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In J.-L. Boulanger, editor, *Formal Methods Applied to Complex Systems: Implementation of the B Method*, chapter 14, pages 427–446. Wiley ISTE, Hoboken, NJ, 2014.
24. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME'03*, LNCS 2805, pages 855–874. Springer, 2003.
25. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, 2008.
26. O. Lichtenstein and A. Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 97–107, New York, NY, USA, 1985. ACM.
27. S. Merz and H. Vanzetto. Encoding TLA$^+$ into Many-Sorted First-Order Logic. In *Proceedings ABZ*, volume 9675 of *LNCS*, pages 54–69. Springer, 2016.
28. T. Mikula. RichTextFX. `https://github.com/FXMisc/RichTextFX`.
29. B. Naveh. JGraphT. `http://jgrapht.org/`.
30. T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.
31. D. Plagge and M. Leuschel. Seven at One Stroke: LTL Model Checking for High-level Specifications in B, Z, CSP, and More. *Int. J. Softw. Tools Technol. Transf.*, 12(1):9–21, Jan. 2010.
32. 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*, pages 372–386, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

33. M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design*, pages 127–144, 2000.
34. SonarSource SA. SonarQube. `https://www.sonarqube.org/`.
35. Travis CI GmbH. Travis CI. `https://travis-ci.org/`.
36. M. Triska. The Finite Domain Constraint Solver of SWI-Prolog. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming*, pages 307–316, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
37. J. Witulski and M. Leuschel. Checking Computations of Formal Method Tools - A Secondary Toolchain for ProB. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, pages 93–105, 2014.
38. F. Yang, J. Jacquot, and J. Souquières. JeB: Safe Simulation of Event-B Models in JavaScript. In *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 1*, pages 571–576, 2013.