Embedding SMT-LIB into B for Interactive Proof and Constraint Solving

Sebastian Krings[0000−0001−6712−9798] and Michael Leuschel[0000−0002−4595−1518]

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

Abstract. The SMT-LIB language and the B language are both based on predicate logic and have some common operators. However, B supports data types not available in SMT-LIB and vice versa. In this article we suggest a straightforward translation from SMT-LIB to B. Using this translation, SMT-LIB can be analyzed by tools developed for the B method. We show how Atelier B can be used for automatic and interactive proof of SMT-LIB problems. Furthermore, we incorporated our translation into the model checker ProB and applied it to several benchmarks taken from the SMT-LIB repository. In contrast to most SMT solvers, ProB relies on finite domain constraint propagation, with support for infinite domains by keeping track of the exhaustiveness of domain variable enumerations. Our goal was to see whether this kind of approach is beneficial for SMT solving.

1 Introduction and Motivation

B [1] and Event-B [2] are formal software development methods following the correct-by-construction approach. They are supported by a range of tools such as Atelier B [16], the integrated development environment Rodin [3] or the animator and model checker ProB [34]. Most B users rely on a mixture of automatic and interactive proof as well as model checking during development.

The SMT-LIB language and its logics [7] and the B language have several similarities. Both are based on predicate logic, they support the same arithmetic operators and quantifiers. Furthermore, both are strongly typed languages. However, there are considerable differences as well. For instance, B supports several data types not (fully) available in SMT-LIB, such as (finite) sets and sequences. For SMT-LIB, these only exist as a proposal [40]. However, some solvers already provide partial support, e.g., CVC4 [5] supports finite sets [13]. SMT-LIB and certain SMT solvers on the other hand are able to cope with real arithmetic, while B only supports integer arithmetic.

With this article, we bridge the gap between B and the SMT-LIB language, embedding SMT-LIB in the B language. This allows reasoning over both SMT-LIB constraints as well as SMT solving algorithms, using the B method and its tool chain, e.g., one can specify any algorithm working on SMT-LIB constraints using the B language and prove its correctness using Atelier B.

Listing 1: Boolean Example in SMT-LIB

```
(set-logic QF_UF)
(declare-fun p () Bool)
(assert (and p (not p)))
(check-sat)
```

Additionally, as the semantics specified for SMT-LIB are preserved during the translation, one can analyze properties of given SMT-LIB constraints in B. This could be used to perform a meta-level analysis of SMT-LIB.

2 Introductory Examples

First, let us look at two simple examples illustrating the general translation scheme: In Listing 1 we encode the assertion $p \wedge \neg p$, which is obviously unsatisfiable. With the first line, we state that the solver should use the logic QF_UF of quantifier free uninterpreted functions. We do not need to translate this fact to B, since B does not distinguish between different logics. The second line introduces a constant symbol p that is of type boolean. As an SMT constant can have just one value, we translate it to a B constant with the same name.

The type can then be specified using the `PROPERTIES` section of a B machine that contains predicates that have to be true for the constants. In this case, we assert that $p \in BOOL$, where $BOOL$ is the set $\{TRUE, FALSE\}$.

The assertion $p \wedge \neg p$ cannot be written in B as trivially as one might expect, because B does not support the use of booleans as predicates. Hence, the assertion has to be translated as $p \wedge \neg p \Leftrightarrow p = TRUE \wedge \neg p = TRUE$. The complete B machine can be found in Listing 2. Once loaded, ProB detects the unsatisfiability and reports that the properties are inconsistent.

The same basic idea can be used for integer arithmetic as shown in Listings 3 and 4. Here, we solve the indeterminate equation system $6x + 12y + 3z = 30 \wedge 3x + 6y + 3z = 12$. This time, ProB is able to find a valuation for the constants and satisfiability can be reported.

Note that `INTEGER` is the B type for the integers. As with SMT-LIB, the B set of integers represents mathematical integers. The B method tools in general, and ProB's solver in particular, can handle arbitrarily large integers.[1] In contrast to SMT-LIB, B's arithmetic operators do not support an arbitrary number of operands. Hence, `(+ x y z)` has to be translated into `x + y + z`.

[1] The CLP(FD) library employed by ProB may generate overflows, which ProB tries to catch and provide alternative treatment for. In case this is not possible, ProB reports “unknown”.

Listing 2: Boolean Example in B

```
MACHINE BooleanExample
CONSTANTS p
PROPERTIES
 p:BOOL & p=TRUE & not(p=TRUE)
END
```

Listing 3: Integer Example in SMT-LIB

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (= (+ (* 6 x) (* 2 y) (* 12 z)) 30))
(assert (= (+ (* 3 x) (* 6 y) (* 3 z)) 12))
(check-sat)
```

3 Translating SMT-LIB to B

Because B provides more involved operators than SMT-LIB, translating SMT-LIB into B is mostly straightforward. New SMT-LIB non-parametric sorts (i.e. new base types) declared by the user are mapped to B *deferred* sets. Deferred sets introduce new base type to B as well, each containing an arbitrary (and possibly infinite) but fixed non-zero number of elements. If a new sort is parametric (i.e. a recursive type) further sets are generated for each concrete combination of parameters used in the input.

For instance, the SMT-LIB declaration `(declare-sort NewSort 0)` corresponds to a B deferred set `NewSort`. In contrast, a parametric sort declared by `(declare-sort Pair 2)` is not included in the B machine. Instead, its instantiations such as `(Pair Int Bool)` are introduced as sets containing pairs taken from `Int` × `Bool`. Nested pairs are used for higher number of arguments.

Listing 4: Integer Example in B

```
MACHINE IntegerExample
CONSTANTS x, y, z
PROPERTIES
 x:INTEGER & y:INTEGER & z:INTEGER &
 6*x + 12*y + 3*z = 30 &
 3*x +  6*y + 3*z = 12
END
```

New function symbols are translated to B constants. The SMT types are thereby mapped to B types, e.g., the `Int` sort is mapped to B's `INTEGER` set. For functions declared using `declare-fun`, we create constants defined as total functions between the parameter types and the result type.

Certain SMT-LIB operators have a direct equivalence in B. With the exception of division, the arithmetic operators can be translated directly. Furthermore, like SMT-LIB, the B language and tools support universal and existential quantification natively.

There is no absolute value function in B. However it can be translated as $|x| = \max(\{-x, x\})$. Integer division and modulo in SMT-LIB are defined following the Euclidean definition by Boute [10], while B uses a floored division [30].[2] Thus in B $-8/3 = -2$ while in SMT-LIB it is -3. Furthermore, in B, $x \mod y$ is only defined if x is non-negative and y is positive. In contrast, the Euclidean definition permits both cases. We express (in Section 3.3) SMT-LIB's division and modulo by rewriting it to B's floored division. Another difference is that in SMT-LIB all functions are total, meaning that $2/0$ is an (unknown) integer value. So, in SMT-LIB $x/0 = 2$ is a satisfiable formula, while in B it is ill defined. Currently, our B translation of $x/0 = 2$ is not well-defined in B [4],and Atelier-B will generate a well-definedness proof obligation that cannot be discharged.ProB will also raise errors, followed by reporting unknown for the constraint.[3]

Aside from the translations above, certain constructs available in SMT-LIB are not available in B and require more involved translations:

– In B, booleans are values and cannot be used as predicates. We showed how to overcome this limitation in the introductory examples. Listing 2 shows how to turn booleans into predicates simply by comparing them to `TRUE`.
– There is no if-then-else for expressions or predicates in B. The B if-then-else may only be used in substitutions (aka statements). Thus, the if-then-else from SMT-LIB has to be rewritten as we explain in Section 3.1.
– Analogously, B features a let substitution but no let predicate. We rewrite let as shown in Section 3.2.

Additionally, B has no dedicated data types for arrays or bit vectors. While we experimented with different translations, we found no representation performing well for both constraint solving and proof. Further research in this area is needed and part of our future work.

Another limitation is that there are no real numbers in B [1]. There is experimental support for floats and reals in Atelier B since version 4.1 [16]. However, proof support is far from being useful and there is currently no support in ProB. Hence, we currently do not take them into account in our translation.

[2] More precisely, the definition of division in B [1] is $n/m = \min(\{x \mid x \in \mathbb{N} \land n < m * succ(x)\})$.

[3] We may improve our translation to remedy this, but our assumption is that most SMT-LIB examples are well-defined according to B.

3.1 If-Then-Else

As mentioned above, the if-then-else construct in B can only be used in substitutions, not in predicates or expressions. However, the core theory of the SMT-LIB language supports the `ite` function that returns its second or third argument depending on the truth value of the first one. To mimic the behavior of `ite` we can reuse a translation developed for a translation from TLA$^+$ to B [27].

In order to translate `(ite` P E_1 E_2`)` we first have to look at the type of E_1 and E_2. If both are predicates, the if-then-else can be expressed in terms of implications, that is, `(ite` P E_1 E_2`)` is translated to $(P \Rightarrow E_1) \wedge (\neg P \Rightarrow E_2)$. Implication is available in B, so the predicate above can be used as a replacement for `ite` in case the arguments are all predicates.

When the `ite` construct is used as an expression rather than a predicate, the translation is more complex. We use the translation suggested by [27]: For both branches of the `if` we create a lambda function that maps 1 to E_1 or E_2 respectively. Afterwards, the union of the two lambda relations is computed:

$$(\lambda t \cdot (t = 1 | E_1)) \cup (\lambda t \cdot (t = 1 | E_2)).$$

This relation has two elements both mapping 1 to a result. In order to mimic the behavior of if-then-else, we now have to assure that one of the lambda relations is empty depending on the value of P:

$$(\lambda t \cdot (t = 1 \wedge P | E_1)) \cup (\lambda t \cdot (t = 1 \wedge \neg P | E_2)).$$

Now, either P or $\neg P$ is false, making the respective lambda expression to be the empty set (and avoiding the evaluation of the corresponding expression E_i). The other lambda expression maps 1 to the result of `(ite` P E_1 E_2`)`. Hence, we just have to apply the relation to 1 to extract the result:

$$(ite\ P\ E_1\ E_2) == (\lambda t \cdot (t = 1 \wedge P | E_1)) \cup (\lambda t \cdot (t = 1 \wedge \neg P | E_2))(1).$$

Observe that B strictly distinguishes between boolean values and predicates. However, there are operators to convert between the two. We considered other encodings of if-then-else, such as using a constant function. However, they often did not harmonize with the inner workings of B and its tools. The encoding presented above exhibits the best performance so far.

Take for example the SMT-LIB formula, where we suppose x to be a natural number: $(ite\ (not\ (=\ x\ 0))\ (div\ 10\ x)\ (div\ 10\ (-\ x\ 1)))$. Our translation is

$$(\lambda t \cdot (t = 1 \wedge x \neq 0 | 10/x)) \cup (\lambda t \cdot (t = 1 \wedge \neg(x \neq 0) | 10/(x-1)))(1)$$

One may think we could translate this into a simpler B expression:

$$\{TRUE \mapsto 10/x, FALSE \mapsto 10/(x-1)\}(bool(x \neq 0))$$

However, it has the problem that in order to determine the value of the expression, one needs to compute the value of the subexpression $\{TRUE \mapsto 10/x, FALSE \mapsto 10/(x-1)\}$. Thus, when x=0, we still need to evaluate $10/x$, generating a well-definedness error in B. Similarly, when x=1 we still need to evaluate $10/(x-1)$, again causing a well-definedness error.

3.2 Let

Similar to the if-then-else, B only has a let substitution and no let for expressions or predicates. SMT-LIB on the other hand includes a let construct available for both expressions and predicates.

According to the SMT-LIB standard [7] a let can always be resolved by inlining its definitions. This step may have to include renaming in order to avoid scoping errors due to capturing by quantifiers.

Regarding performance, inlining may lead to duplicated computation during solving. To some extent this could be countered by ProB's common subexpression detection. However, this would be equal to re-introducing the let internally. In order to avoid duplicating computation we suggest a translation comparable to the one of the if-then-else.

First, let us consider the case of a let where t is a predicate. In this case we rewrite $(let\ ((x_1\ t_1)\ \ldots\ (x_n\ t_n))\ t)$ to

$$\exists x_1, \ldots, x_n \cdot (\bigwedge_{i=1}^{n} x_i = t_i) \wedge t$$

which can be written in B without further translation.

Replacing a let where t is an expression can not be done as easily. As in Section 3.1 we create a function that is called on a fixed value. We translate $(let\ ((x_1\ t_1)\ \ldots\ (x_n\ t_n))\ t)$ to

$$\{k, v \mid k = 1 \wedge \exists x_1, \ldots, x_n \cdot v = t \wedge \bigwedge_{i=1}^{n} x_i = t_i\}(1).$$

The set comprehension contains only one element: The pair $(1, v)$ where v is equal to t, the expression copied from inside the let binder. We call this function on 1 to extract v.

As an example, consider $(let\ ((x_1\ 1)\ (x_2\ 2))\ (+\ x_1\ x_2))$. We translate this to $\{k, v \mid k = 1 \wedge \exists x_1, x_2 \cdot v = x_1 + x_2 \wedge x_1 = 1 \wedge x_2 = 2\}(1)$. The existential quantification can be removed by inlining the definition of the quantified variables, simplifying the comprehension to $\{k, v \mid k = 1 \wedge v = 1 + 2\}(1)$, which represents the partial function $1 \mapsto 1 + 2$. The function is applied to 1 in order to extract the desired result, i. e., $\{k, v \mid k = 1 \wedge v = 1 + 2\}(1) = \{(1 \mapsto 3)\}(1) = 3$.

3.3 Formal Definition of Translation

The translation is implemented as an AST walker carrying around a type environment. We will define how AST nodes are translated by gradually defining a translation function τ, mapping SMT-LIB to B. Just as above, we will use mathematical notation instead of B's ASCII syntax. However, the mathematical representation can be expressed in B without further translation.

Sorts are mapped to B types as discussed above. Existential and universal quantifiers are directly available in B. Let and if-then-else are translated as stated above. For the SMT-LIB Core theory, we translate as follows:

$$\begin{aligned}
\tau(true) &= \mathit{TRUE} & \tau(false) &= \mathit{FALSE}\\
\tau((\texttt{=>}\ x_1\ x_2)) &= (\tau(x_1) \Rightarrow \tau(x_2)) & \tau((and\ x_1\ x_2)) &= (\tau(x_1) \wedge \tau(x_2))\\
\tau((or\ x_1\ x_2)) &= (\tau(x_1) \vee \tau(x_2)) & \tau((=\ x_1\ x_2)) &= (\tau(x_1) = \tau(x_2))\\
\tau((distinct\ x_1\ x_2)) &= (\tau(x_1) \neq \tau(x_2)) & &\\
\tau((xor\ x_1\ x_2)) &= ((\neg\tau(x_1) \wedge \tau(x_2)) \ \ \vee \ \ (\tau(x_1) \wedge \neg\tau(x_2))) & &
\end{aligned}$$

For the SMT-LIB Ints theory, we translate as follows:

$$\begin{aligned}
\tau(\mathit{NUMERAL}) &= \mathit{NUMERAL} & \tau(-x) &= -\tau(x)\\
\tau((-\ x_1\ x_2)) &= (\tau(x_1) - \tau(x_2)) & \tau((+\ x_1\ x_2)) &= (\tau(x_1) + \tau(x_2))\\
\tau((*\ x_1\ x_2)) &= (\tau(x_1) * \tau(x_2)) & \tau((abs\ x)) &= (\max(\{-\tau(x), \tau(x)\}))\\
\tau((\texttt{<=}\ x_1\ x_2)) &= (\tau(x_1) \leq \tau(x_2)) & \tau((<\ x_1\ x_2)) &= (\tau(x_1) < \tau(x_2))\\
\tau((\texttt{>=}\ x_1\ x_2)) &= (\tau(x_1) \geq \tau(x_2)) & \tau((>\ x_1\ x_2)) &= (\tau(x_1) > \tau(x_2))
\end{aligned}$$

Division and modulo are rewritten as discussed above, i. e., we express the Euclidean definitions in terms of B's floored division.

$$\begin{aligned}
&\tau((div\ x_1\ x_2)) = \tau(fdiv\ (-\ x_1\ (ite\ (<x_1 0)\ (ite\ (<\ x_2\ 0)\ (-\ 0\ 1\ x_2)\ (-\ x_2\ 1))\ 0))\ x_2)\\
&\tau((fdiv\ x_1\ x_2)) = (\tau(x_1)\ /\ \tau(x_2))\\
&\tau((mod\ x_1\ x_2)) = \tau((-\ x_1\ (*\ x_2\ (div\ x_1\ x_2))))
\end{aligned}$$

4 Interactive Proof Using Atelier B

In the following section, we illustrate how the B method and its tools could be used for the development and validation of SMT algorithms or SMT optimization rules. As an example, we will verify a translation rule used in CVC4 in case the `-rewrite-divk` command line option has been provided.

The source code is given in Listing 5, taken from CVC4's repository[4]. The given snippet implements a rewriting rule for integer division. Numerator and denominator are given as variables `num` and `den`. For better understandability, we shortened some of the technical detail of CVC4's internal workings, as they do not contribute to the rewriting rule itself.

If the denominator is constant, the division is replaced by a new constant called `intVar`. Depending on the denominator, `intVar` is axiomatically defined using one of two additional constraints: If the denominator is larger than zero, $den * intVar <= num \wedge num < den * (intVar + 1)$ is asserted. Otherwise, $den * intVar <= num \wedge num < den * (intVar - 1)$ is asserted.

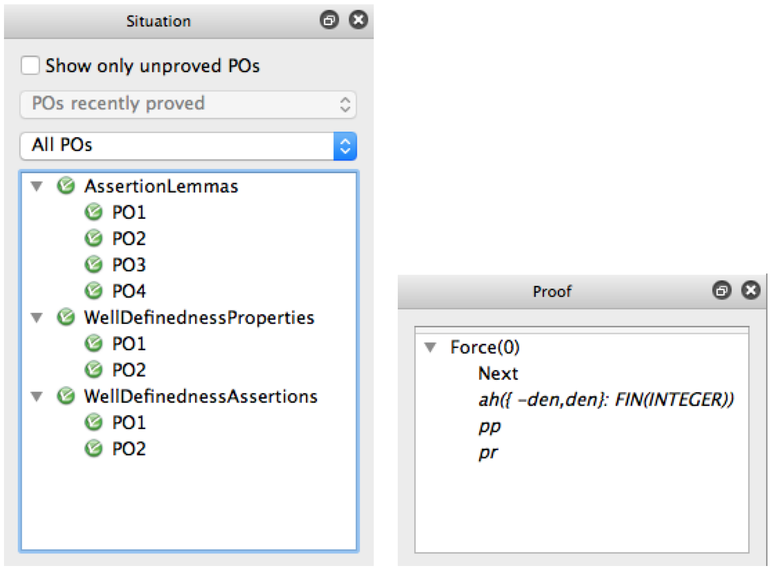
[4] At commit a6bd02c5c442b806b5e01fed40ab9d1017e42bc3, see `https://github.com/CVC4/CVC4/blob/a6bd02c5c442b806b5e01fed40ab9d1017e42bc3/src/theory/arith/theory_arith_private.cpp#L1231` for the full file and context.

Listing 5: CVC4 Source Code Snippet

```
// Input numerator "num" and denominator "den"
// shortened technical detail, e.g., creation of intVar
if(den.isConst()) {
  if(den != 0) {
    if(den > 0) {
       d_containing.d_out->lemma(nm->mkNode(kind::AND,
           nm->mkNode(kind::LEQ,
               nm->mkNode(kind::MULT, den, intVar), num),
           nm->mkNode(kind::LT, num,
               nm->mkNode(kind::MULT, den,
                   nm->mkNode(kind::PLUS, intVar,
                       nm->mkConst(Rational(1)))))));
    } else {
       d_containing.d_out->lemma(nm->mkNode(kind::AND,
           nm->mkNode(kind::LEQ,
               nm->mkNode(kind::MULT, den, intVar), num),
           nm->mkNode(kind::LT, num,
               nm->mkNode(kind::MULT, den,
                   nm->mkNode(kind::PLUS, intVar,
                       nm->mkConst(Rational(-1)))))));
    }
  }
  return intVar;
}
```

Listing 6: B Encoding

```
DEFINITIONS
    smt_abs(x) == max({-x,x});
    smt_div_mod(m,n,q,r) ==
        (n /= 0 => m = n*q + r & 0 <= r & r <= smt_abs(n) - 1);
    smt_div(x,y,divres) ==
        #(modres).(modres:INTEGER
                  & smt_div_mod(x,y,divres,modres));
    smt_mod(x,y,modres) ==
        #(divres).(divres:INTEGER
                  & smt_div_mod(x,y,divres,modres))
ASSERTIONS
    !(num,den,intVar).(num:INTEGER & den:INTEGER
        & intVar:INTEGER &
        smt_div(num,den,intVar) =>
          (den > 0 =>
              den*intVar <= num & num < den*(intVar + 1)) &
          (den <= 0 =>
              den*intVar <= num & num < den*(intVar - 1)))
```



Situation

Show only unproved POs

POs recently proved

All POs

- AssertionLemmas
  - PO1
  - PO2
  - PO3
  - PO4
- WellDefinednessProperties
  - PO1
  - PO2
- WellDefinednessAssertions
  - PO1
  - PO2

Proof

```
Force(0)
    Next
    ah({ -den,den}: FIN(INTEGER))
    pp
    pr
```

(a) List Of Proof Obligations (b) Successful Proof Tree

Fig. 1: Atelier B Screenshots

We want to use B and Atelier B to verify if the additional assertions are implied by the definition of modulo and division in SMT-LIB. Proving the reverse direction could be done in the same way.

In order to prove the rule correct, we translate it to B using the translation defined in Section 3. The result can be found in Listing 6. The given B machine defines the absolute value, division and modulo following our translation scheme.

The property we want to prove is then provided as an assertion using B's universal quantifier, written as `!`: For all combinations of integers `num,den,intVar`, corresponding to the variables in CVC4's code, we want to prove that if `intVar` is the result of the division of `num` by `den` it follows that the axioms added by CVC4 are indeed true. The left hand side of the implications represents the two paths that can be taken in the if statement in Listing 5.

Atelier B type checks the machine and automatically generates eight proof obligations. Four of them are concerned with well-definedness and four with the actual assertion. Figure 1a shows the list of proof obligations. The ones named "PO1" to "PO4" in the group "AssertionLemmas" are concerned with the proof of our propagation rule.

The well-definedness proof obligations stem from the usage of the maximum operator in the definition of `smt_abs`. It is only well-defined if there exists a maximum element. Hence, we have to prove that the set $\{-den, den\} \cap \mathbb{N} \in \mathbb{F}(\mathbb{N})$, where $\mathbb{F}(\mathbb{N})$ is the set of all finite subsets of $\mathbb{N}$.

The proof obligation can be discharged by interactively adding the additional hypothesis $\{-den, den\} \in \mathbb{F}(\mathbb{Z})$. Once added, Atelier B's automatic provers can process further and discharge the proof obligation. Figure 1b shows the proof tactic used to perform the proof.

The remaining proof obligations are used to show the validity of the assertion in different steps. All share a set of common hypotheses that might be used in the proof:

– $num \in \mathbb{Z} \wedge den \in \mathbb{Z} \wedge intVar \in \mathbb{Z}$
– $\exists m.(m \in \mathbb{Z} \wedge (den \neq 0 \Rightarrow num = den * intVar + m \wedge 0 \leq m \wedge m \leq \max(\{-den, den\}) - 1))$

The generated obligations are:

PO1. We have to prove that using the hypotheses given above and the additional one $0 + 1 <= den$ it follows that $den * intVar \leq num$.
PO2. Proves that from the same preconditions $num + 1 \leq den * (intVar + 1)$ follows.
PO3. With the additional hypotheses $den + 1 \leq 0$, we have to prove that $den * intVar \leq num$ is true.
PO4. Using the same hypotheses $num+1 \leq den*(intVar+1)$ has to be shown.

As you can see the proof obligations mimic the case distinction done in Listing 5. The remaining proofs can be performed automatically this time.

In summary, we have used both interactive and automatic proof methods offered by Atelier B to show that the implemented rewriting rule is compliant with SMT-LIB's local definitions of the operators used.

5 Constraint Solving using ProB

The ProB kernel can be viewed as a constraint-solver for the basic datatypes of B and the various operators on it. It supports booleans, integers, user-defined base types, pairs, records and inductively: sets, relations, functions and sequences. These datatypes and operations are embedded inside B predicates, which can make use of the usual logical connectives ($\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg$) and typed universal ($\forall x.P \Rightarrow Q$) and existential ($\exists x.P \wedge Q$) quantification. For integers and finite base types ProB uses the finite domain library CLP(FD) [14], whereby each B variable is associated with an interval of possible values. Sets and relations are represented as Prolog terms, with special representations for large known sets and for infinite sets defined by predicates. ProB tries to delay enumeration of values and gives priority to deterministic computations. To cater for large sets and relations, ProB prioritizes operations that are deterministic *and* are guaranteed to produce data values in an efficient representation.

The underlying mechanism is thus deterministic propagation of partial information about variables, interleaved with controlled enumeration of possible values. This is fundamentally different to the DPLL(T) [26] based SMT solvers and the DPLL [17] or CDCL [35] based SAT solvers.

Using SMT solvers and ProB for cooperative constraint solving and proof has been suggested by us [33], where we made the observation that constraint logic programming based solvers and SMT solvers show different strength and weaknesses. On the one hand, SMT solvers are usually faster when it comes to detecting unsatisfiability. They can handle variables with infinite domains, e.g. integers, more easily. On the other hand, constraint logic programming based solvers can often effectively handle intertwined constraints over bounded variables. Furthermore, they always generate a model for a satisfiable formula. In case of constraints specified in B, combining constraint logic programming and SMT solving in ProB has been proven successful [33].

We compare ProB to Z3 [18] and CVC4 [5] on benchmark files taken from the SMT-LIB benchmark repository. We used development snapshot versions of all three solvers.[5] We limit the benchmarks to non-incremental ones taken from the logics including quantifiers and integer arithmetic: QF_IDL, (QF_)LIA, (QF_)NIA. Logics starting with QF are quantifier-free. The different and possibly combined abbreviations for logics are LIA for linear integer arithmetic and NIA for non-linear integer arithmetic. IDL represents integer difference logic, i. e., expressions of the form $r = o_1 - o_2$.

In addition to comparing provers, we also compare several options of ProB's constraint solver. In particular, we compare vanilla ProB to

- A version using random instead of linear enumeration of CLP(FD) domains,
- A version using common sub-expression elimination (CSE), and
- A version featuring an extended rule set of CHR [25] rules used to infer certain facts CLP(FD) is unable to infer on its own.

All benchmarks were run on StarExec [39]. The machines used feature an Intel Xeon E5–2609 Quad-Core CPU running at 2.4 GHz and 256 GB of RAM. Red Hat Enterprise Linux Workstation 6.3 was used as the operating system.

Regarding time and memory limits, we used the same values used in the SMT Competition [6]. The timeout was set to 1500 seconds (25 minutes) walltime and CPU time for all solvers. Solvers were enforced to use 100 GB of memory or less.

Table 1 lists the total number of benchmarks detected satisfiable or unsatisfiable by the different solvers. As was to be expected, ProB is outperformed by Z3 and CVC4. Z3 clearly outperforms ProB both when it comes to runtimes and number of solved benchmarks. CVC4 is faster than ProB but solves less benchmarks. However, this is mostly due to the fact that the version of CVC4 used did not fully support non-linear integer arithmetic yet.

ProB being outperformed is especially shown by the high number of benchmarks ProB has to report "unknown" on. In most cases, this is due to infinite sets that would need to be enumerated exhaustively in order to solve the constraint or to prove it unsatisfiable. In certain cases, ProB is able to detect that any further attempt is futile and gives up reporting "unknown".

Furthermore, the table shows that ProB performs better on satisfiable benchmarks. Again, this is due to the CLP(FD) based solving kernel. Using constraint programming, it is easier to find a valuation than to detect unsatisfiability. Especially for infinite domains, the latter might even be impossible.

[5] The development version of ProB is available at `http://www3.hhu.de/stups/prob/Download`.

Table 1: Benchmarks: Median Runtimes and Deviation (in s)

solver	configuration	# successful tests	⌀ runtime sat	⌀ runtime unsat
		Successful Tests per Solver		
ProB	Vanilla	9081	0.84 (0.04)	1.38 (0.85)
ProB	Random	9081	0.84 (0.04)	1.39 (0.85)
ProB	CSE	9074	0.84 (0.04)	1.38 (0.84)
ProB	CHR	9065	0.87 (0.04)	1.32 (0.73)
Z3	-	16322	0.01 (0.01)	0.56 (0.56)
CVC4	-	7757	3.5 (5.16)	1.29 (1.29)
		Common Successful Tests		
ProB	Vanilla	1328	1.65 (0.83)	1.23 (0.61)
ProB	Random	1328	1.64 (0.82)	1.24 (0.63)
ProB	CSE	1328	1.64 (0.8)	1.22 (0.6)
ProB	CHR	1328	2.61 (2.42)	1.29 (0.68)
Z3	-	1328	0.12 (0.16)	0.02 (0.02)
CVC4	-	1328	0.57 (0.81)	0.05 (0.05)

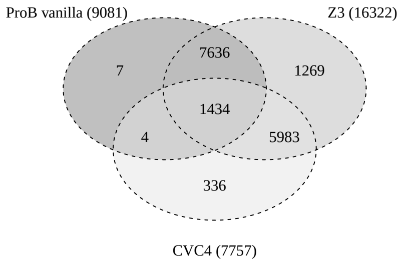
Figure 2a shows the number of test cases solved by each of the solvers individually as well as the number of test cases multiple solvers were able to solve. We only consider the "vanilla" configuration of ProB, in order not to compare CVC4 and Z3 with a portfolio of ProB-based solvers. As can be seen, CVC4 and Z3 each contribute some test cases they alone were able to solve. Furthermore, there is a large class of test cases both can solve.

Interestingly, the diagram shows an (albeit small) number of benchmarks can only be solved by ProB. We suspect that this is due to the different technologies used by ProB and Z3 / CVC4, as benchmarks that are only solved by ProB can easily be classified: they are satisfiable and involve non-linear constraints, variables are highly intertwined and the SMT solvers CVC4 run out of time rather than memory. The efficiency of CLP(FD) for non-linear arithmetic is underlined by the fact that ProB won the NIA division of the 2016 SMT competition[6].

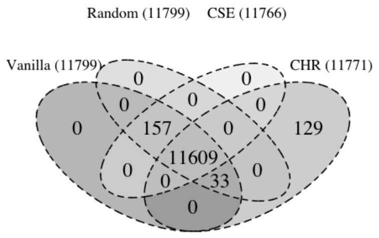
In addition to the result, we measured the runtimes of the different solvers. Table 1 shows how long the solvers took to produce different results. In general, ProB is much faster for satisfiable benchmarks than for unsatisfiable ones. This is to be expected from a tool that has mainly been used to find models of formulas. In contrast, CVC4 and Z3 runtimes do not differ much between satisfiable and unsatisfiable benchmarks.

For better comparability, the table also shows the median runtimes and median absolute deviation on the commonly solved benchmarks. ProB is one to two orders of magnitude slower than the dedicated SMT solvers. However, ProB's run time includes the time spent in the translation phase.

[6] See `http://smtcomp.sourceforge.net/2016/results-NIA.shtml`



(a) Solvers



(b) ProB Options

Fig. 2: Performance Comparison

Table 2: Performance Comparison: Overall Results

solver	configuration	sat	unsat	unknown	timeout	memory out
ProB	Vanilla	8625	456	1444	7228	0
ProB	Random	8625	456	1444	7228	0
ProB	CSE	8625	449	1444	7235	0
ProB	CHR	8627	438	1138	7550	0
Z3	-	12472	3850	412	1019	0
CVC4	-	4152	3605	9324	672	0

Summarizing, we suspect that, especially for detection of satisfiability, a CLP(FD) based approach can be a useful addition to DPLL(T) based algorithms if used in a solver portfolio. Since we introduced a certain overhead by the translation, a direct implementation should add to the gain.

In addition to the benchmarks above, we evaluated how the different options the ProB kernel can be tweaked with influence its performance on SMT-LIB benchmarks. Figure 2b shows the results.

Of the options mentioned above, only the "CHR" option has a positive influence on the number of solved test cases. However, this influence is neither positive nor negative. On the one hand, the CHR rules help to identify unsatisfiable predicates. There is a significant number of test cases that can only be solved with the option enabled. On the other hand, evaluating propagation rules takes time. There is a set of (mostly satisfiable) benchmarks that used to be solvable within the timeout, but is not solvable with the extended rule set.

6 Related Work

Of course there are other solvers supporting SMT-LIB available. Most prominently, these are Boolector [12], CVC4 [5], veriT [11], Yices [22] and Z3 [18].

Using model checkers to solve SMT-LIB formulas has been suggested at least for certain SMT-LIB formulas. A translation from SMT-LIB formulas featuring bit vectors to SMV has been suggested [24], using the NuSMV [15] symbolic model checker as solver. In contrast to this approach, we translate into the input language of an explicit state model checker rather than a symbolic model checker. However, as the B models generated by our translation do not include state transitions, this does not cause a considerable difference in behavior or performance. Furthermore, our translation supports more SMT-LIB constructs.

Constraint Logic Programming has been applied to SMT-LIB constraints in different projects [28,29,21]. Howe and King present a SAT solver in Prolog as a programming pearl [28] and extended it to SMT [29]. The solver is quite small and easy to understand and extend, showing how certain Prolog techniques lead to an efficient implementation. An SMT solver implemented in CHR [25] rather than pure Prolog has been presented as well [21]. It features an input language comparable to SMT-LIB, however, it can not be applied to SMT-LIB directly.

A translation in the opposite direction, i.e., from Event-B proof obligations to SMT-LIB has been developed [19,23]. Sadly, it proves hard to fully encode B's and Event-B's axioms of set theory. The authors of [19,23] thus suggest to break down set theory and arithmetic into first-order formulas using uninterpreted functions for membership, etc. The resulting constraint is enriched by certain set theoretic axioms conveying background knowledge about Event-B to the SMT solvers. As a result, encoded formulas only approximate the Event-B semantics, but work well in practice.

A comparable approach has been followed in the Isabelle community as well. Blanchette et al. [9] translate Isabelle/HOL to first-order relational logic. Afterwards Kodkod is used, translating to SAT formulas solved using a SAT solver. The Sledgehammer tool to discharge proof obligations in interactive proofs in Isabelle can be connected to SMT solvers as well [8]. The authors report a considerable increase in automatically discharged proof obligations.

Using first-order theorem provers for SMT solving has been considered by Vampire [37]. Recent improvements include the introduction of the FOOL logic, an extension of many-sorted first-order logic by a first-class boolean sort [32]. FOOL is supported by Vampire [31] and includes if-then-else and let and thus simplifies translating SMT-LIB into TPLP, the common input language of first-order theorem provers. In comparison to our approach, our CLP(FD) backend focusses more on model finding for satisfiable formulas rather than proof.

A higher level approach towards embedding B and Event-B into SMT-LIB has also been evaluated [33]. Instead of relying on a first-order encoding, we directly employ Z3's own set theory solver. Thus, we only have to supply axiomatic definitions of operators unavailable in SMT-LIB, e.g., cardinality. Empirical comparison shows that both approaches have their merits and none of them is strictly superior [33]. In particular, Z3 was good at detecting inconsistent predicates but not good at finding solutions (aka models). This finding is also confirmed in one of our projects [38], where we experimented with various encodings and solvers for university timetabling.

7 Future Work

In the future, we would like to further investigate ProB's performance in comparison with SMT solvers to gain a deeper understanding on kinds of constraints and how they relate to the performance of CLP(FD) and DPLL(T). In order to perform a more in-depth comparison, we would like to extend our translation and ProB's constraint solving kernel to support incremental solving. This would enable us to use larger industrial case studies for our evaluation.

We would also like to further evaluate the effectiveness of our translation, considering different representations of SMT-LIB expressions in B. While we have done so for smaller collections of benchmarks, we would like to compare encodings more thoroughly throughout the whole SMT-LIB collection. In particular, we would like to improve our translation for bit vectors and arrays. Another aspect to be considered later is the time we allow the solvers to run. It would be interesting to check if the timeout influences the solvers differently.

Furthermore, we want to evaluate how a combination of ProB with the Atelier B provers performs as a solver portfolio. However, as classical rule based provers, the Atelier B provers are not designed to report models for satisfiable formulas. An efficient integration with ProB into a combined solver is thus more complex than just running the tools in parallel. For the combined solver to be of use, we have to enable ProB to identify predicates that it should submit to the provers and make it react to the two possible outcomes "proof" and "unknown".

The empirical evaluation showed that the enumeration strategies employed by ProB are not fully suitable for solving SMT-LIB. In future work, we want to enrich our constraint solver by further enumeration strategies. This will not only aid the SMT solver but will strengthen the model checker as well.

Along with tuning the enumeration strategy we would like to deepen our understanding of the effectiveness of the additional propagation rules and how they interact with CLP(FD). The evaluation shows that the added rules render certain benchmarks solvable while decreasing the performance on others. As of now, we have no proper way of deciding upfront which rules to enable.

8 Discussion and Conclusion

Summarizing, we have presented a translation from SMT-LIB to B that allows all tools available for the B method to be employed on SMT-LIB files. In addition to Atelier B and ProB, there are other model checkers such as pyB [41] and eboc [36], both using different approaches to constraint solving. A new platform for automatic proof is developed in the BWare [20] project.

We laid the groundwork for an analysis or proof of SMT solving algorithms using the B method by embedding SMT-LIB in B or Event-B. We were able to mimic the semantics of SMT-LIB in B, including constructs like if-then-else. Aside from ProB, the translation allows further B method tools like Atelier B to examine SMT-LIB data structures, expressions and algorithms. We have performed a case study showing how to prove SMT propagation laws using the

B method. Hence, our translation could help reasoning about solvers and solving procedures in a structured way.

Furthermore, we made the constraint solving and model finding capabilities of ProB available in the form of a standalone constraint solver that takes the SMT-LIB language as input, making it available in a more general context. This not only enables users to use ProB without having to learn B, it also gives us access to the benchmarks and test cases of the SMT-LIB collection. Our evaluation showed that it is competitive for special types of benchmarks, i.e., it was able to win the NIA track of the 2016 SMT competition. However, the overall performance is rather weak.

Performance aside, using ProB to solve benchmarks taken from the SMT-LIB collection helped us discover different errors and inconsistencies in ProB itself. Deploying ProB on the numerous new test cases has increased our test coverage. In particular, SMT-LIB benchmarks tend to exercise different parts of ProB's kernel than classical B machines due to the diverse usage of constraints.

During benchmarking, we found bugs both in the SMT-LIB translation as well as in ProB's kernel. Usually, these were made obvious by the SMT solvers disagreeing ProB, e.g., Z3 reported unsatisfiability while ProB found a model. Furthermore, several performance bottlenecks have been brought to our attention and have been resolved. In summary, making ProB available as a general purpose SMT solver has helped us to improve ProB itself.

In summary, despite the relatively low number of test cases successfully solved by ProB, we still believe a translation from SMT-LIB to B is beneficial:

- It allows for cross-checking of results using a different approach to solving.
- It enables using SMT-LIB's benchmarks to validate and improve B tools.
- Using Atelier B it provides interactive (assisted) proof. As far as we know, there is currently no other way to interactively tackle SMT-LIB constraints.
- ProB already provides reasoning over sets and strings, features just starting to find their way into SMT-LIB and is competitive for special kinds of benchmarks.

Last, we hope that our article is able to narrow the gap between the SMT solving, the constraint logic programming and the formal methods communities and eases mutual understanding of algorithms and design principles.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 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, and L. Voisin. An open extensible tool environment for Event-B. In *Proceedings ICFEM*, volume 4260 of *LNCS*, pages 588–605. Springer, 2006.
4. J.-R. Abrial and L. Mussat. On Using Conditional Definitions in Formal Theories. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Proceedings ZB'2002*, volume 2272 of *LNCS*, pages 242–269. Springer, 2002.

5. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings CAV*, LNCS, pages 171–177. Springer, 2011.
6. C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. Rajamani, editors, *Proceedings CAV*, pages 20–23. Springer, 2005.
7. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.
8. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT Solvers. In *Proceedings CADE*, LNCS, pages 116–130. Springer, 2011.
9. J. C. Blanchette and T. Nipkow. Nitpick: A Counterexample Generator for Higher-order Logic Based on a Relational Model Finder. In *Proceedings ITP*, LNCS, pages 131–146. Springer, 2010.
10. R. T. Boute. The Euclidean Definition of the Functions Div and Mod. *ACM Trans. Program. Lang. Syst.*, 14(2):127–144, 1992.
11. T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Proceedings CADE*, LNCS. Springer, 2009.
12. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In S. Kowalewski and A. Philippou, editors, *Proceedings TACAS*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.
13. D. Cantone and C. Zarba. A New Fast Tableau-Based Decision Procedure for an Unquantified Fragment of Set Theory. In R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *LNCS*, pages 126–136. Springer, 2000.
14. 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*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
15. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings CAV*, LNCS, pages 359–364. Springer, 2002.
16. ClearSy. *Atelier B 4.1 Release Notes*. Aix-en-Provence, France, 2009. Available at `http://www.atelierb.eu/`.
17. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.
18. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings TACAS*, LNCS, pages 337–340. Springer, 2008.
19. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT Solvers for Rodin. In *Proceedings ABZ*, LNCS, pages 194–207. Springer, 2012.
20. D. Delahaye, C. Dubois, C. Marché, and D. Mentré. *The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations*, volume 8477 of *LNCS*, pages 290–293. Springer, 2014.
21. G. J. Duck. SMCHR: satisfiability modulo constraint handling rules. *CoRR*, abs/1210.5307, 2012.
22. B. Dutertre. Yices 2.2. In *Proceedings CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.
23. 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.

24. A. Fröhlich, G. Kovásznai, and A. Biere. Efficiently Solving Bit-Vector Problems Using Model Checkers. In *Proceedings SMT Workshop*, pages 6–15. 2013.
25. T. Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(1–3):95–138, 1998.
26. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Proceedings CAV*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
27. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM*, volume 7321 of *LNCS*, pages 24–38. Springer, 2012.
28. J. M. Howe and A. King. Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings. pages 165–174. Springer, 2010.
29. J. M. Howe and A. King. A Pearl on SAT and SMT Solving in Prolog. *Theor. Comput. Sci.*, 435:43–55, June 2012.
30. D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison Wesley Longman Publishing Co., Inc., 1997.
31. E. Kotelnikov, L. Kovács, G. Reger, and A. Voronkov. The Vampire and the FOOL. In *Proceedings CPP*, CPP 2016, pages 37–48. ACM, 2016.
32. E. Kotelnikov, L. Kovács, and A. Voronkov. A First Class Boolean Sort in First-Order Theorem Proving and TPTP. In *Proceedings CICM*, volume 9150 of *LNCS*, pages 71–86. Springer, 2015.
33. S. Krings and M. Leuschel. SMT Solvers for Validation of B and Event-B models. In *Proceedings iFM*, volume 9681 of *LNCS*. Springer, 2016.
34. 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, 2014.
35. J. Marques-Silva and K. Sakallah. GRASP: a search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.
36. P. Matos, B. Fischer, and J. Marques-Silva. A Lazy Unbounded Model Checker for Event-B. In *Proceedings FMSE*, volume 5885 of *LNCS*, pages 485–503. Springer, 2009.
37. G. Reger, M. Suda, and A. Voronkov. Instantiation and Pretending to be an SMT Solver with VAMPIRE. In *Proceedings SMT Workshop*. 2012.
38. D. Schneider, M. Leuschel, and T. Witt. Model-Based Problem Solving for University Timetable Validation and Improvement. In *Proceedings FM*, volume 9109 of *LNCS*, pages 487–495. Springer, 2015.
39. A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. In *Proceedings Automated Reasoning*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.
40. G. Weissenbacher, D. Kröning, and P. Rümmer. A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard. In *Proceedings SMT Workshop*. 2009.
41. J. Witulski and M. Leuschel. Checking Computations of Formal Method Tools - A Secondary Toolchain for ProB. In *Proceedings F-IDE*, volume 149 of *EPTCS*. Electronic Proceedings in Theoretical Computer Science, 2014.