14

From Animation to Data Validation: The ProB Constraint Solver 10 Years On

We present our 10 years of experience in developing and applying the PROB validation tool. Initially, the tool provided animation and model checking capabilities for the B-method. Over the years, it has been extended to other formal specification languages and provides various constraint-based validation techniques. The tool itself was developed in SICStus Prolog, and makes use of the finite domain library together with newly developed constraint solvers for Booleans, sets, relations and sequences. The various solvers are linked via reification and Prolog co-routines. The overall challenge of PROB is to solve constraints in full predicate logic with arithmetic, set theory and higher-order relations and functions for safety critical applications. In addition to the tool development, we also provide details about various industrial applications of the tool as well as about our efforts in qualifying the tool for usage in safety critical contexts. Finally, we also describe our experiences in applying alternate approaches, such as SAT or SMT.

14.1. The problem

The B-method [ABR 96] is a formal method for specifying safety critical systems, reasoning about those systems and generating code that is correct by construction. The first industrial usage of B was the development of the software for the fully automatic driverless Line 14 of the Paris Métro, also called Météor (*Metro est-ouest rapide*) [BEH 99]. This was a great success; quoting [SIE 09]: "*Since the commissioning of Line 14 in Paris in 1998, not a single malfunction has been noted in the software developed using this principle*". Since then, many other train control systems have been developed and installed worldwide [DOL 03, BAD 05, ESS 07].

Chapter written by Michael LEUSCHEL, Jens BENDISPOSTO, Ivo DOBRIKOV, Sebastian KRINGS and Daniel PLAGGE.

Formal Methods Applied to Complex Systems: Implementation of the B Method,
First Edition. Edited by Jean-Louis Boulanger.

Initially, the B-method was supported by two tools, BToolkit [BCO 99] and Atelier B [CLE 09], which provided mainly both automatic and interactive proving environments, as well as code generators. To be able to apply the code generators, one has to *refine* the initial high-level specifications into lower-level B (called B0). This process is illustrated in Figure 14.1. Every refinement step engenders proof obligations; if all proof obligations are discharged one has the guarantee that the final B0 specification correctly implements the initial high-level specification.[1]

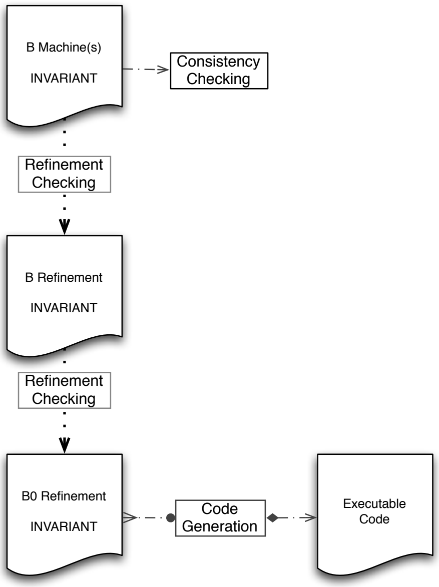

Figure 14.1. *B Development process*

14.1.1. *Animation for B*

In this "correct-by-construction" approach, it is of course vital that the initial high-level specification (at the top of Figure. 14.1) correctly covers the intended application. To some extent, this can be ensured by providing

1 The correctness of the code generation phase is of course also critical; here the industrial users typically use two different, independently developed code generators, targeting different hardware platforms which are both in operation in the final system.

invariants and assertions, but it quickly became obvious that additional validation of the high-level specification is important. In 2003, only very limited additional validation was available. The BToolkit provided an interactive animator with some strong limitations. Basically, the user had to provide values for parameters and existentially quantified variables, the validity of which was checked by the BToolkit prover. This approach was justified by the undecidability of the B language, but was tedious for the user and prevented automated validation.

The PROB validation tool was developed to fill this gap in the tooling landscape. The first problem that PROB set out to solve was to provide automatic animation, freeing up the user from providing values for parameters and quantified variables. This was to be achieved by developing the animator in logic and constraint logic programming.

14.1.1.1. *Challenge*

Indeed, the major challenge of animating or validating B is the expressiveness of its underlying language. B is based on predicate logic, augmented with arithmetic (over integers), (typed) set theory, as well as operators for relations, functions and sequences. As such, B provides a very expressive foundation which is familiar to many mathematicians and computer scientists. For example, Fermat's Last Theorem can be written in B as follows:

$$\forall n.(n > 2 \Rightarrow \neg\exists a, b, c\,.\,a^n + b^n = c^n).$$

In B's ASCII syntax (AMN), this is written as follows:

```
 !n.(n>2 => not(\#(a,b,c).(a{*}{*}n
+ b{*}{*}n = c{*}{*}n)))
```

A more typical example in formal specifications would be the integer square root function, which can be expressed in B as follows:

$$isqrt = \lambda n.(n \geq 0 \mid max(\{i \mid i^2 \leq n\})).$$

As another example, let us express the fact that two graphs g_1, g_2 are isomorphic in B:

$$\exists p \,.\, p \in 1..n \rightarrowtail\!\!\!\!\twoheadrightarrow 1..n \wedge \forall i.(i \in 1..n \Rightarrow p[g_1[\{i\}]] = g_2[\{p(i)\}])$$

where, $1..n \rightarrowtail\!\!\!\!\twoheadrightarrow 1..n$ is the set of all permutations between the set of integers $1..n$ and itself, and $g_k[\{x\}]$ stands for the relational image of g_k for $\{x\}$, i.e. the set of all successor nodes of x in the graph g_k. This predicate is also more typical of the kind of predicates that appear in high-level B specifications. In B, these predicates can appear as invariants over variables, assertions, guards of operations, conditions of conditional statements and many more.

Due to arithmetic and the inclusion of higher-order functions, the satisfiability of B formulas is obviously undecidable. As such, animation is also undecidable, as operation pre-conditions or guards in high-level models can be arbitrarily complex. We cannot expect to be able to determine the truth value of Fermat's Last Theorem automatically, but PROB *is* capable of "solving" the graph isomorphism predicate for reasonably sized graphs,[2] without the user providing a value for the permutation p. It is also capable of computing the integer square root function above, e.g. determining that $isqrt(101) = 10$ or $isqrt(1234567890) = 35136$.[3] The tool has now also been extended to animate Z and TLA$^+$, whose logical foundations share a large common basis.

14.1.2. *Model checking B*

Once implemented, the automatic animation capabilities also enabled a second kind of validation, namely systematically exploring the state space of a B specification, i.e. *model checking*. Initially [LEU 03], PROB allowed us to examine the state space of a B model for invariant violations, deadlocks and assertion violations. Later [PLA 10], this was extended to check temporal (LTL and CTL) properties of the high-level models. Model checking provides additional guarantees of correct behavior of a high-level specification. In addition, the combination of model checking and constraint solving can provide a more elegant approach in specifying and solving certain problems than either technique alone.

2 E.g. 0.2 s for solving a "hard" isomorphism between two graphs with 30 nodes and 90 edges or 40 s for determining that two random graphs with 1,000 nodes and 1,500 edges are not isomorphic.

3 This is one of the specifications which is given as an example of a non-executable specification in [HAY 89].

14.1.2.1. *Challenge*

From a constraint solving perspective, the challenges are very similar to those for animation. One additional complication is, however, that we now need to find *all* solutions to the constraints. This is important for ensuring that *all* possible executions of a B model are analyzed, and that solutions are normalized (to avoid duplicate states in the state space). Also, if we want to deal with large state spaces we need state space reduction techniques (e.g. detecting symmetries), fast animation steps and optimized memory usage.

14.1.3. *Data validation*

In order to avoid multiple developments, a safety critical program is often made from a generic B-model and data parameters that are specific to a particular deployment. In other words, the initial B machine in Figure 14.1 is very generic and requires various parameters to be provided at runtime. For example, in railway systems, these parameters would describe the tracks, switches, traffic lights, electrical connections and possible routes. Adapting the data parameters is also used to "tune" the system. The proofs of the generic B-model rely on assumptions about the data parameters, e.g. assumptions about the topology of the track. It is vital that these assumptions are checked when the system is put in place, as well as whenever the system is adapted (e.g. due to line extension or addition or removal of certain track sections in a railway system).

Before 2009, this validation of the parameters was basically conducted by using automated provers. Siemens, for example, was using Atelier B along with custom proof rules and tactics, dedicated to dealing with larger data values [BOI 00, BOI 02].[4] This approach had two major shortcomings. First, if the proof of a property fails, the feedback of the prover is not very useful (and it may be unclear whether there actually is a problem with the data or just with the power of the prover). Second, the data parameters became so large that the provers ran out of memory, even with maximum memory allocated. This led us [LEU 11] to apply PROB's constraint solving engine to validate those data properties automatically. This turned out to be very successful: full validation

4 Standard provers have not been developed with large, concrete values in mind. For example, many proof rules will duplicate parts of the goal to be proven. This frequently leads to out-of-memory problems when the duplicated parts contain large constants.

was now achieved in minutes rather than weeks or months. The tool is now used by various companies for similar data validation tasks, sometimes even in contexts where B itself is not used for the system development process. In those cases, the underlying language of B turns out to be very expressive and efficient to cleanly encode a large class of data properties.

14.1.3.1. *Challenge*

From a constraint solving perspective, the challenge here is clearly to deal with big data: one has relations containing tens of thousands or hundreds of thousands of elements and these should not *per se* grind the constraint solving engine to a halt. Also, numbers tend to be very large (e.g. positions of beacons expressed in millimeters).

Another challenge is validation of the output of the tool itself. In safety critical environments, we need to provide a case of why a validation tool itself can be trusted. Many standards, such as the EN 50128 in the railway domain, provide criteria that have to be met for a tool to be usable for certain applications. For PROB this meant the development of a rigorous testing and validation process, along with the generation of a validation report which is actively maintained alongside the tool itself.

14.1.4. *Constraint-based checking and disproving for B*

Over the years, PROB's constraint solving capabilities have increased and as such new possibilities for validation have opened up. Some of these applications use the constraint solver as a complement to the prover:

– check if an invariant of a B machine is inductive, i.e. check if it is possible to prove the B machine correct by induction. If a counter example is found, we know that any proof attempt will be futile.

– check if an invariant of a B machine ensures that no deadlock can appear [HAL 11].

– check whether the refinement proof obligations are provable or not.

These applications are complementary to proof; the main difference is that counter examples can be provided which can be inspected within the animation interface. In addition, many other applications arise: test-case generation, understanding certain aspects of a B model, such as control flow.

14.1.4.1. *Challenge*

The main challenge for this application is to deal with complicated constraints and possibly very big constraints (as is the case for deadlock checking [HAL 11]). In addition, we want to try and provide counter-examples even if some of the variables are not bounded to a finite domain. Finally, we want to detect when the result of the constraint solver (i.e. the fact that no counter example was found) can be used as a formal proof.

14.1.5. ***Summary***

In essence, the challenge and ultimate goal of PROB is to solve constraints:

– for an undecidable formal method with existential and universal quantification, higher-order functions and relations, unbounded variables;

– very large data;

– infinite functions to be dealt with symbolically;

– in a reliable way, e.g. so as to satisfy standard EN50128;

– fast solving, with minimal overhead and memory consumption;

– being able to find all solutions for predicates;

– dealing with big constraints and complicated constraints.

14.2. Choice of implementation technology

14.2.1. ***What was used before?***

14.2.1.1. *Proof*

Before the development of PROB, the validation of B models relied solely on the B provers. For animation, the user had to provide values for quantified variables; the correctness of the values were checked by the prover in automatic mode. When the prover failed, it was not necessarily clear whether this was due to the weakness of the prover or because the user had chosen wrong values. Similarly, data validation relied on automatic proof, along with custom proof rules.

All this meant that animation was very tedious for the user and no automated validation, such as model checking, (section 14.1.2) was possible. Concerning data validation (section 14.1.3) meant that about a month of work was necessary to validate new configuration data.

14.2.1.2. *Naive enumeration*

Tools for other languages, such as the TLC model checker for TLA, [YU 99] use naive enumeration for solving constraints. These tools can be very fast when no constraint solving is necessary, but are obviously very bad at solving constraints and thus bad for animating or model checking high-level specifications or for constraint-based checking.

The power of using logic programming was realized by several works: Bowen [BOW 98] developed an animator for Verilog in Prolog, [KIN 01] pursued a Horn logic approach to encode denotational semantics, [WIN 98] presents an animator for Z implemented in Mercury. None of these works, though, used the potential of coroutines or constraint logic programming.

14.2.2. *Why was constraint logic programming used?*

When moving into the formal methods field and coming from a logic programming background it quickly become obvious that the existing tools had severe limitations and that those could be overcome by constraint logic programming. From within the field of formal methods, there was also the feeling that certain specifications were inherently not executable [HAY 89]. But when moving from a logic programming group to a formal methods group (as was the case for the first author) it was obvious that one could do much better than the state-of-the-art at the time using Prolog and constraint solving.

Unknown to us at the time (around 2000), another team pursued similar ideas leading to the CLP-S solver [BOU 02] and the BZTT tool [AMB 02] based on it. This work also gave rise to a company (Lerios), which concentrated on model-based testcase generation and later ported the technology to an imperative programming language. Unfortunately, the development of BZTT and CLP-S has been halted; the tool is no longer available.

In this context, we could mention many other works (such as, e.g. [DEL 01]) which used constraint solving for validation of formal models.

Another interesting related work is the Setlog [DOV 00] constraint solver. Compared to PROB, Setlog has powerful unification procedure, but only deals with sets and has problems dealing with larger sets.[5]

14.3. Implementation of the PROB constraint solver

14.3.1. *Architecture*

14.3.1.1. *Overview*

The PROB kernel can be viewed as a constraint solver for the basic datatypes of B (and Z) 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 are made up 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. An overview of the various solvers residing within the PROB kernel can be seen in Figure 14.2. We explain these parts and their history in the following sections.

14.3.1.2. *Coroutines and determinism*

All versions of PROB have been developed using SICStus Prolog. The initial versions of PROB tried to delay enumeration and give priority to deterministic computations. This was first implemented using coroutines via the `when` metapredicate. More precisely, choice points (such as a predicate $x \in \{1, 2\}$) were guarded by a `when` predicate to ensure that:

– either enough information was available to resolve the choice point deterministically;

– or the solver has switched to enumeration mode.

Later, most of the uses of `when` were replaced by the more restricted but much faster `block` declarations.

14.3.1.3. *Controlling choice points via waitflags*

It was identified very quickly that a more fine-grained enumeration was required, in order to prioritize the choice points once the solver had switched

5 E.g. computing the union of two intervals `un(int(1,10),int(2,15),R)` takes minutes in setlog (4.6.17), while computing the B equivalent $1..10 \cup 2..15$ in PROB is instantaneous.

to enumeration mode. This led to the development of a *waitflags* library which stores choice points and their expected number of solutions. The idea was that whenever the PROB kernel was to create a choice point it would have to:

1) estimate the number of solutions for the choice points;

2) obtain a "waitflag" variable from the library;

3) block on this variable and only execute the choice point once this variable had been grounded.

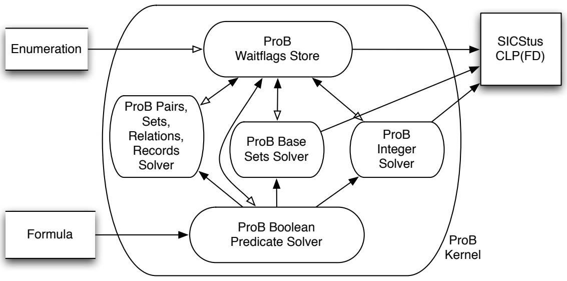

Figure 14.2. *A view of the* PROB *kernel*

The grounding of the waitflags was conducted and explicitly controlled by the kernel in the enumeration phase.

14.3.1.4. *Coping with large datavalues*

This scheme was later refined further driven by the requirements of the data validation application (see section 14.1.3), in particular the requirement to deal with large integers and large relations. Indeed, initially PROB represented sets using Prolog lists. This scheme clearly breaks down for larger sets, and a second representation for fully known sets was introduced. This representation used the AVL library of SICStus and guaranteed for instance membership checks with logarithmic complexity (in the size of the set). It is vital that the PROB kernel uses the AVL representation rather than the list representation whenever possible. For this, a special priority of 0 was introduced in the waitflags library: it is used for those operations that are deterministic *and* are guaranteed to produce data values in an efficient representation.

14.3.1.5. *Integrating CLP(FD)*

Initially, ProB provided its own support for arithmetic. This was obviously less efficient than the built-in CLP(FD) library of SICStus Prolog. However, at the time SICStus discouraged the joint use of coroutines and CLP(FD). This issue was solved in version 4.1 of SICStus Prolog and we then started to integrate CLP(FD) into the ProB kernel. After the introduction, a few more issues were uncovered which led to erroneous results or segmentation faults when CLP(FD) was used in conjunction with coroutines. These issues have now been sorted out, and CLP(FD) can now be reliably used for improved solving of arithmetic constraints. ProB still has the option of being able to run without CLP(FD), falling back on the more limited Prolog implementation of arithmetic in the ProB kernel. This can have two uses: in case a B specification manipulates large numbers the CLP(FD) library will generate overflow exceptions. (The ProB kernel catches those exceptions when they occur directly upon posting an arithmetic constraint. In this case ProB falls back to the Prolog implementation, which uses Prolog's arbitrary precision integers. However, this scheme cannot be applied if the exception occurs later, e.g. due to some instantiation of an unbound variable.) Also, for low-level B machines, which do not require constraint solving, the posting of CLP(FD) constraints induces a certain overhead, which can be avoided by turning CLP(FD) off.

We also investigated using the CLP(B) solver from SICStus Prolog 4. This, however, was less successful. The solver often had problems with larger formulas. For example, CLP(B) runs out of memory after about 5 min on the most complicated SATLIB example in [HOW 10] (flat200-90 with 600 Boolean variables and 2237 Clauses), whereas ProB solves it under 2 s.

14.3.1.6. *Linking solvers via reification*

Much later, inspired by an industrial case study [HAL 11] requiring solving very big constraints, we realized the importance of reification as a way of linking various parts of the ProB kernel. Indeed, propagation of information from one solver of ProB to another was often suboptimal, which became apparent in this case study.

CLP(FD) provides reification for certain constraints. For example, one can post the constraint `R #<=> (Y #> 0)`. This can serve as a way to link CLP(FD) with another solver:

– We can block a coroutine on the variable `R`; this coroutine will be triggered when the truth value of the test `Y #> 0` is known. This way information propagates from CLP(FD) to the other solver.

– In turn, if this coroutine can decide that `Y #> 0` must either be true or false it can simply set the variable `R` to either 0 or 1. This way information propagates the other solver to CLP(FD).

We have provided reification for the arithmetic operators via CLP(FD) and for the basic set operators of B $x \in S$, $x \notin S$, $S \subseteq S'$, $S \subset S'$, $S \not\subseteq S'$, $S \not\subset S'$ as well as $x = y$ and $x \neq y$ in the PROB kernel itself. The kernel tries to avoid setting up choice points whenever possible, using reified versions instead. For example, to compute $f(x)$ with $f = \{1 \mapsto 0, 2 \mapsto 2\}$ we could reify $x = 1$ and wait until the result of this test is known. Once it is known, we can compute the result of $f(x)$.[6] This mechanism is used in many places of the kernel, providing support for operators such as $\cup$, $\cap$, ...

14.3.1.7. *Challenges*

One challenge is that PROB tries to catch well-definedness errors, such as division by zero, the application of a function outside of its domain or applying the maximum operator to an empty set. To some extent, this hinders constraint propagation (e.g. from $10/y = 5$ we cannot infer that $y = 2$ unless we also know that $y \neq 0$) and makes the implementation of the solver more complicated.

Another issue is that upon encountering something like $x/y = 10 \wedge y = 0$, the PROB kernel cannot directly raise an error; it could be that some other constraint restricts y to be non-zero. The solution here is to post-pone raising an error until enumeration is complete. Other technical challenges are related to graceful treatment of timeouts.

14.3.2. *Validation*

PROB is being used as a tool of class T2 according to the norm [CEN 11] for data validation within Alstom and Siemens. A tool of class T2 “supports the test or verification of the design or executable code, where errors in the

6 Obviously, we would also wait on the output value being known and propagate information backwards. i.e. once we know that $f(x) = 0$ we can infer that $x = 1$.

tool can fail to reveal defects but cannot directly create errors in the executable software" [CEN 11, section 3.1.43]. However, we strive for PROB to be also used as a tool of class T3, i.e. a tool that "generates outputs which can directly or indirectly contribute to the executable code (including data) of the safety related system" [CEN 11, section 3.1.44]. To achieve this, a validation report is being maintained along with extensive testing and validation infrastructure.

14.3.2.1. *Testing and continuous integration*

PROB contains unit tests, integration and regression tests as well as model check tests for mathematical laws. All of these tests are run automatically on our continuous integration platform "Jenkins".[7] When a test fails, an email is sent automatically to the PROB development team.

14.3.2.2. *Self- model check with mathematical laws*

With this approach, we use PROB's model checker to check itself, in particular the PROB kernel and the B interpreter. The idea is to formulate a wide variety of mathematical laws and then use the model checker to ensure that no counterexample to these laws can be found. Definitively, PROB now checks itself for over 500 mathematical laws. These even uncovered several bugs in the underlying SICStus Prolog compiler using self model check, e.g.:

– The Prolog `findall` sometimes dropped a list constructor, which means that instead of `[[]]` it sometimes returned `[]`. In terms of B, this meant that instead of $\{\varnothing\}$ we received the empty set $\varnothing$. This violated some of our mathematical laws about sets. This bug was reported to SICS, and it was fixed in SICStus Prolog 4.0.2.

– A bug in the AVL library (notably in the predicate `avl_max` computing the maximum element of an AVL-tree) was found and reported to SICS. The bug was fixed in SICStus Prolog 4.0.5.

14.3.2.3. *Test coverage*

The above validation techniques are complemented by code coverage analysis techniques. In particular, we try to ensure that the unit tests and the self-model checks (section 14.3.2.2) above cover all predicates and clauses of the PROB kernel.

7 See http://en.wikipedia.org/wiki/Jenkins_(software).

14.3.2.4. *Positive and negative evaluation*

For data validation, all properties and assertions are checked twice, both positively and negatively. Indeed, PROB has two Prolog predicates to evaluate B predicates: one positive version which will succeed and enumerate solutions if the predicate is true and the another is a negative version, which will succeed if the predicate is false and then enumerate solutions to the negation of the predicate. The reason for the existence of these two Prolog predicates is that Prolog's built-in negation is generally unsound and cannot be used to enumerate solutions in case of failure.

For a formula to be classified as true the positive Prolog predicate must succeed *and* the negative Prolog predicate must fail, introducing a certain amount of redundancy (admittedly with common error modes). In fact, if both the positive and negative Prolog predicates would succeed for a particular B predicate then a bug in PROB would have been uncovered. If both fail, then either the B predicate is undefined or we have again a bug in PROB.

This validation aspect can detect errors in the predicate evaluation parts of PROB i.e. the treatment of the Boolean connectives $\vee$, $\wedge$, $\Rightarrow$, $\neg$, $\Leftrightarrow$, quantification $\forall$, $\exists$, and the various predicate operators such as $\in$, $\notin$, =, $\neq$, $<$, ... This redundancy can not detect bugs inside expressions (e.g. $+$, $-$, ...) or substitutions (but the other validation aspects mentioned above can).

14.4. Added value of constraint programming

14.4.1. *Cost of development*

The first version of PROB was made available about 10 years ago. Before that, various experimental prototypes had been under development since about 1999. A reasonably large team of researchers has since then helped in developing, maintaining and improving the tool. Initial usage concentrated on animation and model checking. The first industrial usage for data validation started at the end of 2008, and PROB has been used in industry to that effect since 2009.

14.4.2. *User feedback*

The animation and model checking capabilities have helped many users understand and debug their specifications. On many occasions, errors were found in proven models. Various other tools and techniques have been built on top of PROB, leading to over 400 citations of the two main articles about PROB[LEU 03, LEU 08].

For data validation, the tool has achieved a reduction from one man-month to several minutes for validating a new railway configuration [LEU 11]. The tool has also discovered errors that were not previously seen. The PROB tool is now in relatively widespread use in the railway domain (Siemens, Alstom, ClearSy, Systerel, ...), and the university spin-off company Formal Mind has been created for commercial exploitation.

14.4.3. *Was it difficult/necessary for the end user to understand constraint technology?*

The goal of PROB was always (and still is) to provide automated validation of high-level specifications without the user having to understand constraint solving. For many industrial specifications, this goal has now been achieved, but obviously sometimes debugging is still necessary. Maybe surprisingly, many B specifications can be animated by PROB out of the box. The tool now provides external B predicates which can be inserted into a specification to help debug a B specification when run by PROB and help identify (performance) problems. Also, sometimes user annotations are required to mark certain infinite B functions as symbolic, so as to prevent PROB from trying to expand them.[8] But the long-term goal is to further increase the automation, and even to be able for ordinary users to describe their own constraint solving problems in B and have them solved using PROB.

14.4.4. *Comparison with non-constraint solving tools*

We have already discussed the proof-based BToolkit animator earlier in the paper. In the meantime, a variety of other tools have been developed for animating or model checking high-level specifications. These tools, such as Brama [SER 07] and AnimB [MÉT 10] for Event-B or TLC [YU 99] for TLA$^+$, rely on naive enumeration. They can be used if the models are

8 In particular in Event-B [ABR 10] users currently have to axiomatize their own transitive closure, which poses problems if not dealt with symbolically.

relatively concrete, possibly by providing additional animation values in the setup of the tools. However, there is little chance in using such tools for constraint-based checking (section 14.1.4). For example, TLC takes hours to find an isomorphism for two graphs with 9 nodes (using a specification similar to the one seen in section 14.1.1.1; see [LEU 11] for more details). TLC however, can be very efficient for concrete models, where the overhead of constraint solving provides no practical advantage.

14.4.5. *Comparison with other technologies*

In the past few years, we have also investigated a variety of alternate technologies to replace or complement the constraint solver of PROB: BDD-Datalog- based approaches, SAT- and SMT-solving techniques. We have given up the BDD-approach very quickly (see, e.g. [PLA 09]): due to the lack of data types that are more abstract than bit vectors, the complexity of a direct translation from B was too high, even for small models.

For SAT, we have implemented an alternate backend for first-order B in [PLA 12] using the Kodkod interface [TOR 07]. For certain complicated constraints, in particular those involving relational operators, this approach fared very well. The power of clause learning and intelligent backtracking are a distinct advantage here over classical constraint solvers. However, for arithmetic the SAT approach usually has problems scaling to larger integers.

As an example, take the following set comprehension with 49,646 solutions. The PROB solver takes 1.36 s to find the solutions, while the Kodkod-SAT approach takes 145 s on the same hardware:

```
{x,y,z|x:1..10000 \& z = x/y \& z:200..500 \& y : 10..20}
```

Quite often, the SAT approach is better for inconsistent predicates, while the PROB constraint solver often fared better when the predicates were satisfiable. Also, the SAT approach often has problems dealing with large data and cannot deal with unbounded values or with infinite or higher-order functions. Here, an SMT-based approach could be more promising. We have also experimented with SMT-solvers, in particular a SMT-plugin for Event-B [DEH]. So far, the results were rather disappointing, but this may be due to the translation rather than the SMT solvers used. For a cruise control case study of [HAL 11], the SAT and SMT alternatives were not successful in solving the constraints.

14.4.6. *Future plans*

We are working on reducing the overhead of using B compared to directly encoding problems in a lower-level language such as Prolog. To that end, a partial evaluator has already been developed, which can specialize the PROB interpreter for a particular B specification. A long-term research challenge is to be able to use B and PROB as a programming language and as a constraint solving language.

Other avenues that are being pursued are parallel versions of PROB improved symmetry breaking during constraint solving, better constraint solving over unbounded integers (i.e. without a finite domain) using CHR or CHR-like techniques.

14.4.7. *Lessons*

Constraint logic programming in particular and constraint solving in general has a lot to offer for formal methods, and new applications are popping up all the time. Constraint solving can be made to deal with large data, something which is very difficult with SAT-based approaches. The combination of model checking and constraint solving can be very useful, allowing to express certain problems very concisely. Constraint solving is often good at finding solutions; but not so good at detecting unsatisfiable predicates.

The use of Prolog to implement PROB was both a blessing and a curse. SICStus Prolog is a very efficient Prolog engine, and the efficiency and memory consumption of PROB was often very satisfactory. Indeed, in the context of Event-B PROB often had fewer efficiency problems with large specifications than Java-based tools. However, in some aspects the use of Prolog prevents certain optimizations: we cannot easily re-order lists on the fly (e.g. to keep them sorted and remove duplicates); it is difficult to cache results (e.g. when expanding set comprehensions) because backtracking undoes bindings and assert/retract is expensive.

In conclusion, constraint solving has provided the foundation for many novel tools and techniques to validate formal models. While SAT- and SMT-based techniques have also played an increasingly important role in this area, constraint solving approaches have advantages when dealing with large data.

PROB is available for download at `http://www.stups.uni-duesseldorf.de/ProB`. An online logic calculator with examples is available at: `http://www.stups.uni-duesseldorf.de/ProB/index.php5/ProB_Logic_Calculator`.

14.5. Acknowledgments

We would thank all those people who have contributed toward the development of PROB and without whom the tool would not be where it is now: Michael Butler, Fabian Fritz, Marc Fontaine, Corinna Spermann and many more.

14.6. Bibliography

[ABR 96] ABRIAL J.-R., *The B-Book*. Cambridge University Press, 1996.

[ABR 10] ABRIAL J.-R., *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[AMB 02] AMBERT F., BOUQUET F., CHEMIN S., *et al.*, “BZ-testing-tools: a tool-set for test generation from Z and B using constraint logic programming”, *Proceedings FATES’02*, Technical Report, INRIA, pp. 105–120, August 2002.

[BCO 99] B-Core (UK) Ltd, Oxon, UK, *B-Toolkit, On-line manual*, 1999. Available at http://www.b-core.com/ONLINEDOC/Contents.html.

[BAD 05] BADEAU F., AMELOT A., “Using B as a high level programming language in an industrial project: Roissy VAL”, in TREHARNE H., KING S., HENSON M., *et al.*, (eds.), *Proceedings ZB’2005*, LNCS 3455, pp. 334–354. 2005.

[BEH 99] BEHM P., BENOIT P., FaiVRE A., *et al.*, “Météor: a successful application of B in a large project”, in WING J. M., WOODCOCK J., DAVIES J. (eds.), *World Congress on Formal Methods*, LNCS 1708, pp. 369–387. 1999.

[BOI 00] BOITE O., Méthode B et validation des invariants ferroviaires, Master’s Thesis, Denis Diderot University, 2000.

[BOI 02] BOITE O., “Automatiser les preuves d’un sous-langage de la méthode B”, *Technique et Science Informatiques*, vol. 21, no. 8, pp. 1099–1120, 2002.

[BOU 02] BOUQUET F., LEGEARD B., PEUREUX F., "CLPS-B – a constraint solver for B", in KATOEN J.-P., STEVENS P., (eds.), *Proceedings TACAS'02*, LNCS 2280, pp. 188–204, 2002.

[BOW 98] BOWEN J., "Animating the semantics of VERILOG using Prolog", UNU/IIST Technical Report no. 176, United Nations University, Macau, 1999.

[CEN 11] CENELEC, Railway applications – communication, signalling and processing systems – software for railway control and protection systems, Technical Report EN50128, European Standard, 2011.

[CLE 09] CLEARSY, Atelier B, user and reference manuals, Aix-en-Provence, France, 2009. Available at http://www.atelierb.eu/.

[DEH] DEHARBE D., FONTAINE P., GUYOT Y., *et al.,* "Smt solvers for rodin", *Proceedings ABZ'2012*, LNCS. Springer.

[DEL 01] DELZANNO G., PODELSKI A., "Constraint-based deductive model checking", *STTT*, vol. 3, no. 3, pp. 250–270, 2001.

[DOL 03] DOLLÉ D., Essamé D., Falampin J., "B dans le tranport ferroviaire, L'expérience de Siemens transportation systems", *Technique et Science Informatiques*, vol. 22, no. 1, pp. 11–32, 2003.

[DOV 00] DOVIER A., PIAZZA C., PONTELLI E., *et al.,* "Sets and constraint logic programming", *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 5, pp. 861–931, 2000.

[ESS 07] ESSAMÉ D., DOLLÉ D., "B in large-scale projects: the Canarsie line CBTC experience", in JULLIAND J., KOUCHNARENKO O., (eds.), *Proceedings B'2007*, LNCS 4355, pp. 252–254, Springer-Verlag, 2007.

[HAL 11] HALLERSTEDE S., LEUSCHEL M., "Constraint-based deadlock checking of high-level specifications", *TPLP*, vol. 11, nos. 4–5, pp. 767–782, 2011.

[HAY 89] HAYES I., JONES C.B., "Specifications are not (necessarily) executable", *Softw. Eng. J.*, vol. 4, no. 6, pp. 330–338, November 1989.

[HOW 10] HOWE J.M., KING A., "A pearl on SAT solving in Prolog", in BLUME M., KOBAYASHI N., VIDAL G., (eds.), *Proceedings FLOPS'10*, LNCS 6009, pp. 165–174, Springer, 2010.

[KIN 01] KING L., GUPTA G., PONTELLI E., "Verification of a controller for BART", in WINTER V.L., BHATTACHARYA S., (eds.), *High Integrity Software*, Kluwer Academic Publishers, pp. 265–299, 2001.

[LEU 03] LEUSCHEL M., BUTLER M., "ProB: a model checker for B", in ARAKI K., GNESI S., MANDRIOLI D., (eds.), *FME 2003: Formal Methods*, LNCS 2805, pp. 855–874, Springer-Verlag, 2003.

[LEU 08] LEUSCHEL M., BUTLER M.J., "ProB: an automated analysis toolset for the B method", *STTT*, vol. 10, no. 2, pp. 185–203, 2008.

[LEU 11] LEUSCHEL M., FALAMPIN J., FRITZ F., *et al.,* "Automated property verification for large scale b models with ProB", *Formal Asp. Comput.*, vol. 23, no. 6, pp. 683–709, 2011.

[MÉT 10] MÉTAYER C., AnimB 0.1.1, 2010. Available at http://wiki.event-b.org/index.php/AnimB.

[PLA 10] PLAGGE D., LEUSCHEL M., "Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more", *STTT*, vol. 11, pp. 9–21, 2010.

[PLA 12] PLAGGE D., LEUSCHEL M., "Validating B, Z and TLA+ using ProB and Kodkod", in GIANNAKOPOULOU D., MÉRYD. (eds.), *Proceedings FM'2012*, LNCS 7436, pp. 372–386. Springer, 2012.

[PLA 09] PLAGGE D., LEUSCHEL M., LOPATKIN I., *et al.,* "SAL, Kodkod, and BDDs for validation of B models, lessons and outlook", *Proceedings AFM 2009*, pp. 16–22, 2009.

[SER 07] SERVAT T., "Brama: a new graphic animation tool for B models", in JULLIAND J., KOUCHNARENKO O., (eds.), *Proceedings B'2007*, LNCS 4355, pp. 274–276. Springer-Verlag, 2007.

[SIE 09] SIEMENS, "B method - optimum safety guaranteed", *Imagine*, vol. 10, pp. 12–13, June 2009.

[TOR 07] TORLAK E., JACKSON D., "Kodkod: a relational model finder", in GRUMBERG O., HUTH M., (eds.), *Proceedings TACAS'07*, LNCS 4424, pp. 632–647, Springer-Verlag, 2007.

[WIN 98] WINIKOFF M., DART P., KAZMIERCZAK E., "Rapid prototyping using formal specifications", in *Proceedings of the 21st Australasian Computer Science Conference*, pp. 279–294, Perth, Australia, February 1998.

[YU 99] YU Y., MANOLIOS P., LAMPORT L., "Model checking TLA$^+$ specifications", in PIERRE L., KROPF T., (eds.), *Proceedings of CHARME'99*, LNCS 1703, pp. 54–66, Springer-Verlag, 1999.