Under consideration for publication in Formal Aspects of Computing

Automated Property Verification for Large Scale B Models with ProB[1]

Michael Leuschel[1], Jérôme Falampin[2], Fabian Fritz[1], Daniel Plagge[1]

[1] Universität Düsseldorf, Universitätsstr. 1, D-40225 Düsseldorf

[2] Siemens Transportation Systems, 150, avenue de la République, BP 101, 92323 Châtillon Cedex, France

Abstract.

In this paper we describe the successful application of the ProB validation tool for several industrial applications. The initial case study centred on the San Juan metro system installed by Siemens. The control software was developed and formally proven with B. However, the development contains certain assumptions about the actual rail network topology which have to be validated separately in order to ensure safe operation. For this task, Siemens has developed custom proof rules for Atelier B. Atelier B, however, was unable to deal with about 80 properties of the deployment (running out of memory). These properties thus had to be validated by hand at great expense (and they need to be revalidated whenever the rail network infrastructure changes).

In this paper we show how we were able to use ProB to validate all of the about 300 properties of the San Juan deployment, detecting exactly the same faults automatically in a few minutes that were manually uncovered in about one man-month. We have repeated this task for three ongoing projects at Siemens, notably the ongoing automatisation of the line 1 of the Paris Métro. Here again, about a man month of effort has been replaced by a few minutes of computation.

This achievement required the extension of the ProB kernel for large sets as well as an improved constraint propagation phase. We also outline some of the effort and features that were required in moving from a tool capable of dealing with medium-sized examples towards a tool able to deal with actual industrial specifications. We also describe the issue of validating ProB, so that it can be integrated into the SIL4 development chain at Siemens.

Keywords: B-Method, Model Checking, Constraint-Solving, Tools, Industrial Applications.

This article is an extended version of the conference paper [28]. The most important additions of the present article are:

- This article describes the use of our technique in three active deployments, namely the upgrading of the Paris Metro Line 1 for driverless trains, line 4 of the Sao Paolo metro and line 9 of the Barcelona metro. The paper [28] only contained the initial San Juan case study, which was used to evaluate the potential of our approach.

[1] Part of this research has been funded by the EU FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

Correspondence and offprint requests to: Michael Leuschel[1], Jérôme Falampin[2], Fabian Fritz[1], Daniel Plagge[1]

[1] Universität Düsseldorf, Universitätsstr. 1, D-40225 Düsseldorf

[2] Siemens Transportation Systems, 150, avenue de la République, BP 101, 92323 Châtillon Cedex, France

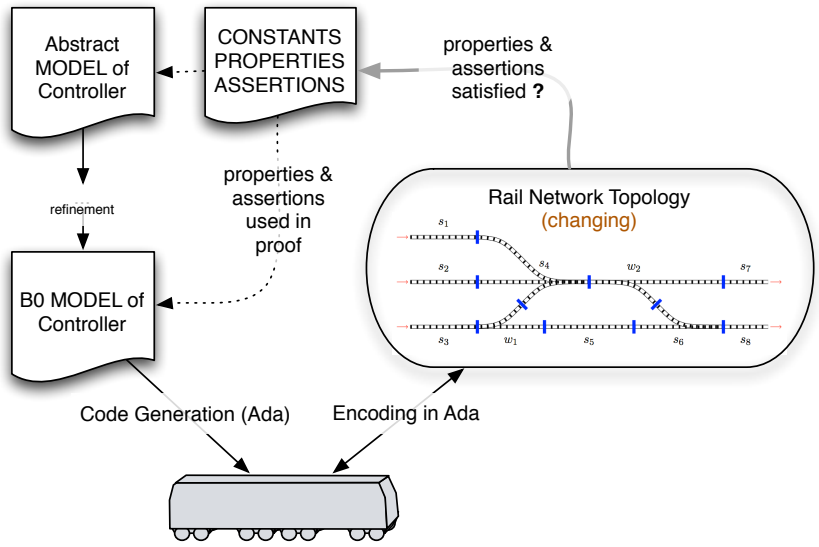

Fig. 1. Overview of the Constants Validity Problem

- In this article we describe the previous method adopted by Siemens in much more detail, as well as explaining the performance issues with Atelier B.
- More comparisons and empirical evaluations with other potential approaches and alternate tools (Brama, AnimB, BZ-TT and TLC) have been conducted.
- We provide more details about the ongoing validation process of ProB, which is required by Siemens for it to replace the existing method.

Also, since [28], ProB itself has been further improved inspired by the application, resulting in new optimisations in the kernel (cf. Section 3.2).

1. Background Industrial Application

Siemens Transportation Systems have been developing rail automation products using the B-method since 1998.[2] The best known example is the software for the fully automatic driverless line 14 of the Paris Métro, also called Météor (**Met**ro **e**st-**o**uest **r**apide) [8]. But since then, many other train control systems have been developed and installed worldwide by STS [14, 7, 16].

One particular development is in San Juan (Puerto Rico), which we will use as one case study in this paper. The line consists of 16 stations, 37 trains and a length of 17.2 km, transporting 115,000 passengers per day. Several entities of Siemens produced various components of this huge project, such as the rolling stock and the electrification. STS developed the ATC (Automatic Control System) named SACEM (Système d'Aide à la Conduite, à l'Exploitation et à la Maintenance).

Currently, STS is involved in the ongoing automatisation of the line 1 of the Paris Métro, which is historically the most heavily used line in Paris with up to 725,000 passengers per day.[3] Inspired by the success of Météor, this line is being upgraded to driverless automatic trains while remaining in operation.[4] Motivations are the shorter headway, leading to bigger transportation capacity, as well as improved security compared to manual operation (better response time, safety procedures are always enforced, etc.). First

[2] At that time Siemens Transportation Systems was named MTI (Matra Transport International).
[3] See `http://fr.wikipedia.org/wiki/Ligne_1_du_métro_de_Paris`.
[4] See `http://fr.wikipedia.org/wiki/Automatisation_de_la_ligne_1_du_métro_de_Paris`.

automatic trains are scheduled to run beginning of 2011, and the upgrade is scheduled to be completed in 2012. Other ongoing projects are the metro line 4 in Sao Paolo and the metro line 9 in Barcelona.

STS are successfully using the B-method and have over the years acquired considerable expertise in its application. STS use Atelier B [38], together with in-house developed automatic refinement tools, with great success. Indeed, starting from a high-level model of the control software, refinement is used to make the model more concrete. Each refinement step is formally proven correct. When the model is concrete enough, an Ada code generator is used. This results in a system ensuring a very high degree of safety, with SIL4 (see, e.g., [35]) certification. Indeed, quoting [37]: "*Since the commissioning of line 14 in Paris on 1998, not a single malfunction has been noted in the software developed using this principle*".

The Property Verification Problem

In this paper, we describe one aspect of the current development process which is unfortunately still problematic, namely the validation of properties of parameters only known at deployment time, such as the rail network topology parameters. These parameters are typically represented as constants in the formal B model.

Figure 1 gives an overview of this issue. Note, the figure is slightly simplified as there are actually two code generators and data redundancy checks during execution. The track is divided into several sub-sections, each sub-section is controlled by a safety critical software. Note that each subsection has its own hardware and software, because a centralised controller would require a huge computer as well as long cables for the interfaces (interfaces between the controller and the platform equipments, interlocking, signals, etc.). The decentralised control also has the advantage of better robustness in case of failure (cut cables, for instance).

In order to avoid multiple developments, each software is made from a generic B-model and data parameters that are specific to a sub-section and a particular deployment. These data parameters take the form of B functions describing, e.g., 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 properties of the track. We therefore have to make sure that the parameters used for each sub-section and deployment actually satisfy the formal assumptions.

For example, in case of the San Juan development, about 300 assumptions were made. It is vital that these assumptions are checked when the system is put in place, as well as whenever the rail network topology changes (e.g., due to line extension or addition or removal of certain track sections).

Siemens Existing Process

To solve this problem, Siemens Transportation Systems (STS) have developed the following approach (see also Figure 2):

1. The parameters and topology is extracted from the concrete Ada program and encoded in B syntax, written into Atelier B definition files. (Definition files contain only B DEFINITIONS, i.e., B macros.) This is done with the aid of a tool written in `lex`.
   Note, that Siemens not only wants to check that the assumptions about the data parameters hold, but also that these have been correctly encoded in the Ada code. Hence, the data is extracted from the Ada program, rather than directly from the higher-level description (which was used to generate the Ada code).
2. The relevant part of the B model is extracted and merged with the definition files containing the topology and the other parameters. The properties from the original B model on the concrete topology and parameters are translated into B assertions.
   In B assertions are predicates which should follow from the properties, and have to be proven. Properties themselves do not have to be proven, but can be used by the prover. By translating the topology properties into B assertions, we thus create proof obligations which stipulate that the topology and parameter properties must follow from the concrete values of the constants.
3. STS tries to prove the assertions with Atelier B, using custom proof rules and tactics, dedicated to dealing with explicit data values [9, 10].
4. Those assertions for which proof is unsuccessful are investigated manually.

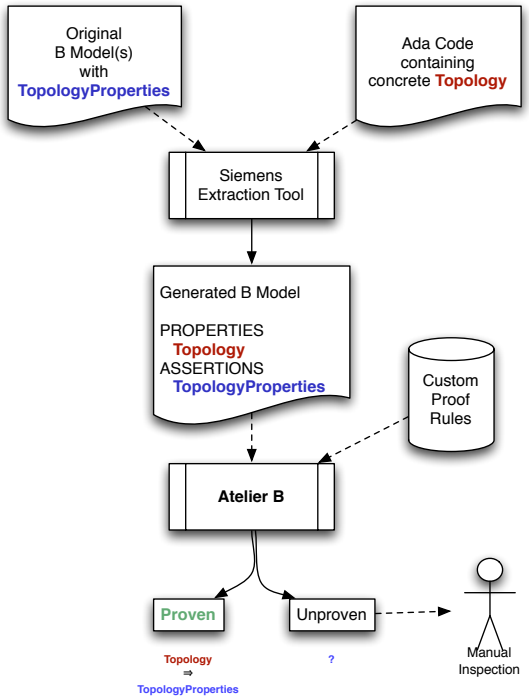

Fig. 2. Overview of the Current Approach

Let us look at a concrete example. The following is part of an initial B model, describing a function `cfg_aig_pos_defaut_i` which computes the default positions of the switches in the sector under consideration:

```
MACHINE acs_as_env_cfg_aiguille
SEES ...
CONCRETE_CONSTANT cfg_aig_pos_defaut_i
PROPERTIES
 cfg_aig_pos_defaut_i : t_nb_aig_par_acs --> t_etat_aig
END
```

As you can see, we have expressed the assumption that the constant `cfg_aig_pos_defaut_i` is a total function from `t_nb_aig_par_acs` (the switches in the sector) to `t_etat_aig` (the possible states of a switch). This property can and will be used in the proof of correctness of the train controller. However, the actual concrete value of `cfg_aig_pos_defaut_i` is as of yet unknown, and will actually vary from one deployment and sector to the other. If the actual value of `cfg_aig_pos_defaut_i` does not satisfy the property, the controller may not work correctly.

Let us now look at the corresponding part of the Ada code, for one particular sector. Here, the Ada array variable `CFG_AIG_POS_DEFAUT` is the realisation of the B constant `cfg_aig_pos_defaut_i`, and it is initialised with a concrete value:

```
CFG_AIG_POS_DEFAUT : constant T_CFG_AIG_POS_DEFAUT :=
    T_CFG_AIG_POS_DEFAUT'(AIG_YE5 => AIG_ACS_DIRECT,
                          AIG_YW5 => AIG_ACS_DIRECT,
                          others  => AIG_ACS_DECONTROLE)
```

The array is initialised with the default value `AIG_ACS_DECONTROLE`, except at positions `AIG_YE5` and `AIG_YW5`. A `lex` based tool developed by Siemens extracts this information from the Ada code, and translates it into B format. Observe that it uses the B override operator to translate the Ada construct with the default "`others`" keyword:

```
PROPERTIES
  cfg_aig_pos_defaut_i =
```

```
   (T_NB_AIG_PAR_ACS ) * {AIG_ACS_DECONTROLE} <+
                                {  AIG_YE5 |-> AIG_ACS_DIRECT,
                                   AIG_YW5 |-> AIG_ACS_DIRECT  }
ASSERTIONS
  cfg_aig_pos_defaut_i : t_nb_aig_par_acs --> t_etat_aig
```

As a side note, the values of the integer variables `AIG_YE5`, `AIG_YE5`, ... are put into a separate definition file. Note that the definition of the concrete value for `cfg_aig_pos_defaut_i` has been put into the PROPERTIES clause, meaning that it does not have to be proven by Atelier B. However, the former property about `cfg_aig_pos_defaut_i` has been moved to the ASSERTIONS clause, so that now we have to prove that it follows from the PROPERTIES clause. This leads to the following proof obligation:

```
(T_NB_AIG_PAR_ACS ) * {AIG_ACS_DECONTROLE} <+
    {  AIG_YE5 |-> AIG_ACS_DIRECT,
       AIG_YW5 |-> AIG_ACS_DIRECT  }  :  t_nb_aig_par_acs --> t_etat_aig
```

Using the values for the variables, this proof obligation is rewritten into the following one:

```
(0..10 ) * {0} <+ {  3|->1 , 7|->1  } : 0..10 --> 0..2
```

Which in turn gets translated by the normalisation process of Atelier B into two subgoals:

```
(0..10 ) * {0} <+ {  3|->1 , 7|->1  } : 0..10 +-> 0..2
```

and

```
dom( (0..10 ) * {0} <+ {  3|->1 , 7|->1  } ) = 0..10
```

If we manage to discharge all proof obligations of the new B machine, we will have established that the concrete constant from the Ada code satisfies the required topology properties.

Next, in order to discharge the proof obligations of this B machine we could use the standard provers of Atelier B. It turns out, however, that this rarely works, as the 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 leads to out-of-memory problems when the duplicated parts contain large constants. As such, custom proof rules were developed [9, 10] which try to prevent this blow up.

For example, one of these custom proof rules is the following one, which can be applied to the second subgoal above:

```
dom(A) <: B
     =>
dom(B*{a}<+ A)= B ;
```

Problems with the Existing Process

This approach initially worked quite well for Siemens, but ran into considerable problems:

- First, if the proof of a property fails, the feedback of the prover is not very useful in locating the problem (and it may be unclear whether there actually is a problem with the topology or "simply" with the power of the prover).
- Second, and more importantly, the constants are nowadays becoming so large (relations with thousands of tuples) that Atelier B quite often runs out of memory, even with the dedicated proof rules and with maximum memory allocated. In some of the bigger, more recent models, even simply substituting values for variables fails with out-of-memory conditions.

This is especially difficult, as some of the properties are very large and complicated (see Figures 3 and 4), and the prover typically fails on these properties. For example, for the San Juan development, 80 properties (out of the 300) could not be checked by Atelier B, neither automatically nor interactively (with reasonable effort; sometimes loading the proof obligation already fails with an out-of-memory condition).

The second point means that these properties have to be checked by hand (e.g., by creating huge spreadsheets on paper for the compatibility constraints of all possible itineraries), which is very costly and arguably less reliable than automated checking (see, e.g., Section 5.2). For the San Juan development, this meant about one man month of effort, which is likely to grow further for larger developments such as the Carnasie line [16].

```
cfg_ipart_cdv_dest_aig_i : t_nb_iti_partiel_par_acs --> t_nb_cdv_par_acs;

!(aa,bb).(aa : t_iti_partiel_acs &  bb : cfg_cdv_aig &
   aa |-> bb : t_iti_partiel_acs <| cfg_ipart_cdv_transit_dernier_i |> cfg_cdv_aig
    => bb : cfg_ipart_cdv_transit_liste_i[(cfg_ipart_cdv_transit_deb(aa)
                    .. cfg_ipart_cdv_transit_fin(aa))]);

cfg_ipart_pc1_adj_i~[{TRUE}] /\ cfg_ipart_pc2_adj_i~[{TRUE}] = {};

!(aa,bb).(aa : t_aig_acs & cfg_aig_cdv_encl_deb(aa) <= bb &
          bb <= cfg_aig_cdv_encl_fin(aa)
   => cfg_aig_cdv_encl_liste_i(bb) : t_cdv_acs);

!(aa).(aa : t_aig_acs
   => t_cdv_acs <| cfg_aig_cdv_encl_liste_i~ |>
   cfg_aig_cdv_encl_deb(aa)..cfg_aig_cdv_encl_fin(aa):t_cdv_acs +-> NATURAL);

cfg_canton_cdv_liste_i |> t_cdv_acs : seq(t_cdv_acs);

cfg_cdv_i~[{c_cdv_aig}] /\ cfg_cdv_i~[{c_cdv_block}] = {};

dom({aa,bb|aa : t_aig_acs & bb : t_cdv_acs &
     bb : cfg_aig_cdv_encl_liste_i[(cfg_aig_cdv_encl_deb(aa) ..
         cfg_aig_cdv_encl_fin(aa))]}) = t_aig_acs;

ran({aa,bb|aa : t_aig_acs & bb : t_cdv_acs &
     bb : cfg_aig_cdv_encl_liste_i[(cfg_aig_cdv_encl_deb(aa) ..
         cfg_aig_cdv_encl_fin(aa))]}) = cfg_cdv_i~[{c_cdv_aig}];
```

Fig. 3. A small selection of the assumptions about the constants of the San Juan topology

```
 ran({aws_ts, troncon_rg_variant_bf | aws_ts : t_aws_ts_pas &
     troncon_rg_variant_bf : (t_troncon_pas * t_rg_variant_bf) &
     dom(%troncon.(troncon : t_troncon_pas &
           inv_aws_ts_cm_autorise_sn_rg_variant_bf_i(aws_ts|->troncon) : t_rg_variant_bf |
           inv_aws_ts_cm_autorise_sn_rg_variant_bf_i(aws_ts|->troncon)))
       = inv_aws_ts_troncon[{aws_ts}] &
     troncon_rg_variant_bf : %troncon.(troncon : t_troncon_pas &
        inv_aws_ts_cm_autorise_sn_rg_variant_bf_i(aws_ts|->troncon) : t_rg_variant_bf |
        inv_aws_ts_cm_autorise_sn_rg_variant_bf_i(aws_ts|->troncon))
   })
 /\
 ran({aws_ts, troncon_rg_variant_bf | aws_ts : t_aws_ts_pas &
     troncon_rg_variant_bf : (t_troncon_pas * t_rg_variant_bf) &
     dom(%troncon.(troncon : t_troncon_pas &
          inv_aws_ts_dde_cm_interdit_rg_variant_bf_i(aws_ts|->troncon) : t_rg_variant_bf |
          inv_aws_ts_dde_cm_interdit_rg_variant_bf_i(aws_ts|->troncon)))
      = inv_aws_ts_troncon[{aws_ts}] &
     troncon_rg_variant_bf : %troncon.(troncon : t_troncon_pas &
       inv_aws_ts_dde_cm_interdit_rg_variant_bf_i(aws_ts|->troncon) : t_rg_variant_bf |
       inv_aws_ts_dde_cm_interdit_rg_variant_bf_i(aws_ts|->troncon))})
 = {}
```

Fig. 4. A pathologically large property about a rail topology

The starting point of this paper was to try to automate this task, by using alternative technology. Indeed, the ProB tool [25, 27] has to be capable of dealing with B properties in order to animate and model check B models. The question was, whether the technology would scale to deal with the industrial models and the large constants in this case study.

In Section 2 we elaborate on what had to be done to be able to parse and load large scale industrial B models into the ProB tool. In Section 3 we present the new constraint propagation algorithms and data structures that were required to deal with the large sets and relations of the case study. The results of the first case study itself are presented in Section 4. In Section 5 we present the outcome of the first real industrial applications, while in Section 6 we present how we plan to validate ProB for integration into the development cycle at Siemens. Finally, in Section 7 we present more related work, discussions and an outlook.

2. Parsing and Loading Industrial Specifications

First, it is vital that our tool is capable of dealing with the actual Atelier B syntax employed by STS. Whereas for small case studies it is feasible to adapt and slightly rewrite specifications, this is not an option here, due to the size and complexity of the specification. Indeed, for the San Juan case study we received a folder containing 79 files with a total of over 23,000 lines of B.

2.1. Improved Parser

Initially, ProB [25, 27] was built using the existing jbtools [39] parser, whose XML output is convenient for importing into other tools. Unfortunately, the jbtools parser does not support all of Atelier B's features. In particular, jbtools is missing support for DEFINITIONS with parameters, for certain Atelier B notations (tuples with commas rather than `|->`), as well as for definition files. This would have made a translation of the San Juan example (containing 24 definition files and making heavy usage of the features not supported by jbtools) near impossible. Unfortunately, jbtools is also difficult to maintain and extend. We managed to somewhat extend the capabilities of jbtools concerning definitions with parameters, but we were not able to fully support them and it was near impossible to generate a clean abstract syntax tree. Indeed, jbtools is built using the JavaCC top-down recursive-descent parser generator, and as such the B grammar has to be made suitable for top-down predictive parsing (which leads to cluttered parse trees, see, e.g., chapter 4 of [6]). Furthermore, jbtools uses several pre- and post-passes to implement certain difficult features of B (such as the relational composition operator; we return to this issue below). This approach prevents the generation of a clean abstract syntax tree. Indeed, certain of the jbtools phases rely, e.g., on the presence of parentheses for their work. As such, we were not able to remove the parentheses (and other syntactic "clutter") from the concrete syntax tree (a common simplification when going from a concrete to an abstract syntax tree).

Thus, the first step towards making ProB suitable for industrial usage, was the development of a new parser. We used the bottom-up LALR parser generator SableCC [19] rather than JavaCC to develop the parser, which allowed us to use a cleaner and more readable grammar (as it did not have to be transformed for predictive top-down parsing) and to provide a fully typed abstract syntax tree. Our parser was built with extensibility in mind, and now supports almost all of the Atelier B syntax. Most areas where we deviate from Atelier B are related to definitions.

Parsing Definitions

Definitions in B are introduced in the `DEFINITIONS` section of a machine. Definitions are allowed to have parameters and, once introduced, can then be used in other contexts. A formal semantics for definitions is not provided in [1] (on page 273 it is only mentioned that the body of a definition should be a "Formal_Text"). In Atelier B, a definition call is replaced syntactically by the corresponding definition body, suitably instantiated for the actual definition parameters. In other words, definitions in Atelier B are treated like macros. However, as is well known, macros can pose a wide range of subtle problems.[5] For example, given the definition `sm(x,y) == x+y` one would expect `sm(1,1)*2` to be equal to `4`. Unfortunately, in Atelier B, `sm(1,1)` is replaced textually by `1+1`, yielding the text `1+1*2` which evaluates to `3`.

We believe that in a formal specification language, these problems should be prevented. Also, to help users in locating errors in their specifications, definitions should be parsed and type checked in isolation and not simply be treated as a piece of text. Thus, in ProB `sm(1,1)*2` evaluates to `4` as expected.[6]

It has already been noted earlier that B is difficult to parse, especially in light of definitions (see Chapter 2 of [31]). Indeed, some of the operators in B are overloaded, such as for example:

- `;` can either mean relational composition when applied to expressions or sequential composition when applied to substitutions. Furthermore, the semicolon `;` is used as a separator of operations, sets, definitions and assertions.
- `||` can either denote the parallel product of two relation expressions or the parallel composition of two substitutions.

[5] See, e.g., `http://gcc.gnu.org/onlinedocs/cpp/Macro-Pitfalls.html`.

[6] We plan to issue a warning to the user that a non-parenthesised definition body is used and that the result obtained by ProB may differ from Atelier B.

- `*` can either denote the Cartesian product or integer multiplication.

This on its own is not yet that problematic. However, combined with the DEFINITIONS feature of B we get ambiguity, as definition bodies can stand for expressions, predicates or substitutions. For example, the meaning of the definition `d(x) == x;x;` cannot be determined on its own and depends on the context it is being used in: `d({1|->2,2|->3})` represents a relational composition expression (with value `{1|->3}`) whereas `d(d(y:=y+1))` denotes a substitution, (whose effect is to increment the variable y by 4).

The solution proposed in [31] is to avoid the overloading and use different syntactic representations (e.g., `;;` for relational composition). Our solution is to require parentheses around sequential composition. For example, `(x;y)` is detected as relational composition and `x;y` as sequential composition. Indeed, normal parentheses cannot be used for substitutions (which are bracketed using BEGIN and END). Also, our parser will first try to parse a definition body as an expression, then as a predicate and finally as a substitution. Only if all three passes fail, do we generate a parse error.

Comma Notation for Pairs

In Atelier B one can use the comma to denote pairs, i.e., one can write `x,y` instead of `x|->y`. This leads to an ambiguity in the syntax: `op(x,y)` could denote calling the operation `op` with one or two parameters. We have solved this issue by requiring parentheses around pairs written using the comma. In other words, `op(x,y)` denotes calling `op` with two parameters and `op((x,y))` denotes calling `op` with one parameter.

Parsing Expressions and Predicates

In some cases our parser is more flexible than the Atelier B variant: the Atelier B parser does not distinguish between expressions and predicates, while our parser does and as such requires less parentheses. For example, according to the grammar in the Appendix B of the Atelier B "B Language Reference Manual 1.8.6" [38], one is allowed to write the predicate `2=1 <=> 1=2`. Our parser accepts this predicate. However, in Atelier B this expression results in a syntax error and one has to write `(2=1) <=> (1=2)`. The reason is that the Atelier B parser does not distinguish between expressions and predicates, and the operator `<=>` binds tighter than `=`.

Summary

In our experience, it is relatively rare that an Atelier-B model needs to be rewritten and thus far, this has not been an issue in practice. Our experience has confirmed that the SableCC abstract syntax tree is much more convenient to work with than the jbtools one. There are, however, also a few drawbacks of using SableCC. For instance, SableCC is relatively rigid, as there are no semantic actions. This meant that in order to obtain line and column information we needed to modify the parser classes generated by SableCC. This was done by weaving aspects into the parser code using AspectJ, so that SableCC can be rerun, e.g., on a modified grammar. However, this obviously complicates the building process and makes the source code more fragile with respect to future changes of SableCC. Another drawback is the performance, which lies below that of the jbtools parser (typically by a factor of 5; see [18]). Still, the performance was sufficient to deal with the largest industrial models we seen so far.

After we had developed our parser, ClearSy have released the Atelier B parser and type checker "bcomp" on sourceforge.[7] Had we known this earlier, we could have tried to import the syntax tree of bcomp into ProB. The drawback would have been the absence of checking of definitions and we also would not have uncovered two of the bugs in the Atelier B parser (see Section 5).

2.2. Improved Type Inference

In the previous version of ProB, the type inference was relatively limited, meaning that additional typing predicates had to be added with respect to Atelier B. Again, for a large industrial development this would have become a major hurdle. Hence, we have also implemented a complete type inference and checking

[7] `http://sourceforge.net/projects/bcomp/`

algorithm for B within ProB. We are making use of the source code locations provided by the new parser to precisely pinpoint type errors. Like the main part of ProB, the type checker has been implemented in Prolog and the algorithm is based upon Prolog unification, in the style of the Hindley-Milner type inference algorithm [34]. As such, this is more powerful than Atelier B's type checker, which proceeds strictly from left-to-right. It is also more powerful than the Rodin [2] type checker for Event-B, often providing better error messages. Indeed, the Rodin type checker [33] is not unification based, but uses a syntax-directed translation scheme [4] based on inherited and synthesised attributes. For example, in contrast to Atelier B and Rodin, our type checker can infer the types for x, y and z in the predicate $x \subseteq y \land y \subseteq z \land card(z) \in z$.

The machine structuring and visibility rules of B are now also checked by the type checker. The integration of this type checker also provides advantages in other contexts: indeed, we realised that many users (e.g., students) were using ProB without Atelier B or a similar tool for type checking.

The new type checker also improves the performance of ProB, e.g., by disambiguating between Cartesian product and multiplication for example. Indeed, previously the kernel of ProB contained code which had to deal with *both* Cartesian product and multiplication, and had to determine at runtime which operation was required.

Finally, another interesting point is that we can also type check the definitions, which we think is important for a formal specification language, to avoid subtle and hard to track errors.

2.3. Other Improvements

The scale of the specifications from STS also required a series of other efficiency improvements within ProB. Indeed, the abstract syntax tree of the main model of the San Juan case study takes 17.6 MB in Prolog pretty-printed form.[8] This was highlighting several performance issues which did not arise in smaller models. For example, there were performance issues in syntax highlighting the textual representation of large data values or when manipulating or displaying a very large number of predicates.

All in all, about eight man-months of effort went into the parser, type checker and the various other improvements, simply to ensure that our tool is capable of loading industrial-sized formal specifications. The development of the parser alone took 4-5 man months of effort.

One lesson of our paper is that it is important for academic tools to work directly on the full language used in industry. One should not underestimate this effort, but it is well worth it for the exploitation avenues it opens up. Indeed, one cannot expect industrial users to adapt their models to suit an academic tool.

In the next section, we address the further issue of effectively dealing with the large data values manipulated upon by the STS specifications. This development took an additional 3 man months of effort.

3. Checking Complicated Properties

In this section we investigate the major challenge of our application, namely the appearance of variables and constants whose values are very large sets and relations, e.g., representing the topology of a railway network. As we have seen in Section 1, this is also what was responsible for the failure of Atelier B. To give the reader an indication, the San Juan case study contains 142 constants, the two largest of which (`cfg_ipart_pos_aig_direct_i`, `cfg_ipart_pos_aig_devie_i`) are relations which contain 2324 tuples. Larger relations still can arise when evaluating the properties or assertions (e.g., by computing set union or set comprehensions).

3.1. Improved Data Structure

The previous version of ProB represented sets (and thus relations) as Prolog lists. For example, the set $\{1, 2\}$ would be represented as `[int(1),int(2)]`. This scheme allows to represent partial knowledge about a set (by partially instantiating the Prolog structure). For example, after processing the predicates $card(s) = 2$

[8] This includes position information; once loaded by Prolog it takes 3.2 MB. Note that the Prolog pretty-printed form is usually at least by a factor of two more compact than the XML representation generated by jbtools. Also note that the SableCC abstract syntax tree takes up 15.06 MB within the Java parser.

and $1 \in s$, ProB would obtain `[int(1),X]` as its internal representation for s (where `X` is an unbound Prolog variable).

However, this representation clearly breaks down with sets containing thousands or tens of thousands of elements. We need a data structure that allows us to quickly determine whether something is an element of a set, and we also need to be able to efficiently update sets to implement the various B operations on sets and relations.

For this we have used an alternative representation for sets using AVL trees — self-balancing binary search trees with logarithmic lookup, insertion and deletion (see, e.g., Section 6.2.3 of [22]). We have used the AVL library of SICStus Prolog 4. Note that when ProB stores the variable values of a state, they are systematically normalised, which will translate all sets into the more efficient AVL from.

An alternative to AVL trees would have been using bit vectors. It would, however, have made representing more complicated sets (e.g., sets of sets) more difficult or even impossible (e.g., for sets of mathematical integers).

3.2. Improved Algorithms

Since the appearance of [28], we have undertaken a thorough examination of the remaining bottlenecks of our kernel. As such, we have optimised many of the basic B operations for our new data structure (e.g., the various restriction operators `<|`, `|>`, `<<|`, `|>>`).

We have also identified that the STS properties and assertion contain a lot of intervals. For example, the property `cfg_aig_pos_defaut_i : t_nb_aig_par_acs --> t_etat_aig` from Section 1 contains two intervals (`t_nb_aig_par_acs` = `0..10` and `t_etat_aig` = `0..2`). The fact that the domain of `cfg_aig_pos_defaut_i` lies within `0..10` can actually be checked very efficiently. Indeed, our AVL representation stores the elements of `cfg_aig_pos_defaut_i` in lexicographic order. We can obtain the minimum $mn \mapsto t_1$ and maximum $mx \mapsto t_2$ elements of the set in logarithmic time. Due to the lexicographic ordering used, we know that mn is the minimum of the domain and mx the maximum of the domain of `cfg_aig_pos_defaut_i`. Hence, we can check `dom(cfg_aig_pos_defaut_i) <: 0..10` in logarithmic time by checking that $mn \geq 0$ and $mx \leq 10$. Doing a similar check for the range of `cfg_aig_pos_defaut_i` still requires traversing the whole set, as t_1 is probably not the minimum of the range, and t_2 probably not the maximum of the range. We have implemented various specialised algorithms for subset and domain checking involving intervals and AVL trees.

These further improvements have led to a factor 20 speedup over [28] for the San Juan case study, and prepares our tool for larger future applications (see Section 7).

3.3. A Performance Experiment

We want to get an idea of the performance of our algorithms for manipulating large sets and relations. For this we performed a small experiment using ProB and a variety of other tools. All experiments were run on a MacBook Pro with 2.33 GHz Core2 Duo processor and 3 GB of RAM.

In our experiment, we measured the time to compute the effect of the following assignment, coming from a B formalisation of the Sieve of Eratosthenes. We assume that `numbers` was initialised to `2..limit` and that `cur=2` (i.e., we measure the first step in the Sieve algorithm, which removes all even numbers).

```
numbers := numbers - ran(%n.(n:cur..limit/cur|cur*n))
```

With limit=10,000 the previous version of ProB ran out of memory after about 2 minutes. With the new data structure this operation, involving the computation of a lambda expression, taking the range and performing a set difference operation, is now almost instantaneous (0.2 seconds) in ProB 1.3.2. For limit = 10,000 it requires 0.09 seconds, for limit = 100,000 it requires 0.91 seconds, while for limit = 1,000,000 ProB 1.3.2 requires about 9.60 seconds. These numbers suggest that ProB scales almost linearly for this example.

Let us compare this performance with that of other formal methods tools. First, we have performed the experiment of Section 3.3 with AnimB [32], an animator for Event-B written in Java. We used version 0.1.1 of AnimB running within Rodin 1.2. The results are summarised in Figures 5 and 6. For example, for limit= 20,000, it took AnimB 5 minutes 8 seconds to compute `numbers := numbers - ran(%n.(n:cur..limit/cur|cur*n))`,

limit	ProB 1.3.2	AnimB	Kodkod	TLC	CoreASM
5,000	0.05 s	18 s	85.78 s	3 s	1.8 s
10,000	0.09 s	75 s	363.18 s	5 s	2.7 s
20,000	0.19 s	308 s	1530.49 s	11 s	6.5 s
100,000	0.91 s	-	-	259 s	216.4 s
1,000,000	9.60 s	-	-	$>$ 16200 s	-

Fig. 5. Some figures of the Sieve experiment (time to compute the first transition)

compared to 0.19 seconds for ProB 1.3.2. We can see thus that there are three orders of magnitude difference for 20,000 elements, and we also see in the log-log plot of Figure 6 that AnimB does not scale linearly (it seems to scale roughly quadratically).

We have also reprogrammed the above experiment in TLA$^+$ [23] and used the TLC model checker (version 3.5 of the TLA Toolbox) [43]. Indeed, TLA$^+$ and the B-Method share a common basis of predicate logic and set theory. The time to run the first step of the Sieve Algorithm again be found in Figures 5 and 6. Timings for TLC were obtained using a stopwatch. As one can see, TLC performs better than AnimB, but there is still a considerable performance gap compared to ProB (a factor of 284 for n=100,000). Note that for 1,000,000 numbers, TLC had not yet finished after 4 hours and 30 minutes.

Note that running the entire Sieve Algorithm to find the 2,262 prime numbers until 20,000 takes 4 minutes and 26 seconds with TLC, but only 2.36 seconds with ProB. Again, note that we measure just one particular aspect here: the efficiency of treating large sets. There are other applications where TLC will be more efficient than ProB.

Roozbeh Farahbod has encoded the sieve experiment using CoreASM [17] for us. The CoreASM tool (Carma version 0.7.1 and CoreASM Engine 1.1.0-alpha, written in Java) took 6.5 seconds for 20,000, and roughly 3 minutes 36 seconds for 100,000 elements; i.e., it was slightly faster than TLC but still considerably slower than ProB.

Our last experiment involved Kodkod [40] which provides a high-level interface to SAT-solvers, and is also at the heart of Alloy [20]. We are currently investigating using Kodkod [40] as an alternative engine to solve or evaluate certain complicated constraints within ProB. Indeed, for certain complicated constraints over first-order relations, Kodkod can be much more efficient than ProB. However, Alloy is based upon the "small scope hypothesis" [21], which obviously does not apply for the particular industrial application of formal methods in this paper. As such, Kodkod is probably not the right tool to manipulate relations containing thousands or tens of thousands of elements. We encoded the above experiment using Kodkod, and it is indeed about two orders of magnitude slower than ProB for 1,000 elements and three orders of magnitude for 10,000 elements; see Figure 5, as well as the log-log plot in Figure 6.

In summary, the performance of ProB when manipulating large sets seems to be much better than other existing formal methods tools. Later, in Section 7, we will compare ProB's performance with a few more formal methods tools.

3.4. Improved Constraint Propagation

There is one caveat, however: the new AVL-tree data structure can (for the moment) only be used by ProB for fully determined values, as ordering is relevant to store and retrieve values in the AVL tree. For example, we cannot represent the term `[int(1),X]` from above as an AVL tree, as we do not know whether `X` will be greater or smaller than 1. Hence, for partially known values, the old-style list representation still has to be used. There are thus the following set representations currently in use by ProB:

- partially known sets stored as (possibly only partially instantiated) Prolog lists,
- fully known sets stored as AVL-trees,
- fully known sets representing an entire base type (i.e., a complete deferred or enumerated set, as well as `BOOL` and `INTEGER`), or one of the predefined sets of integers (`INT`, `NAT`, `NATURAL`, `NAT1`, `NATURAL1`).
- fully known sets stored as closures, to represent certain large sets symbolically (e.g., the set of partial functions over a certain domain and range, or the set of numbers within an interval).

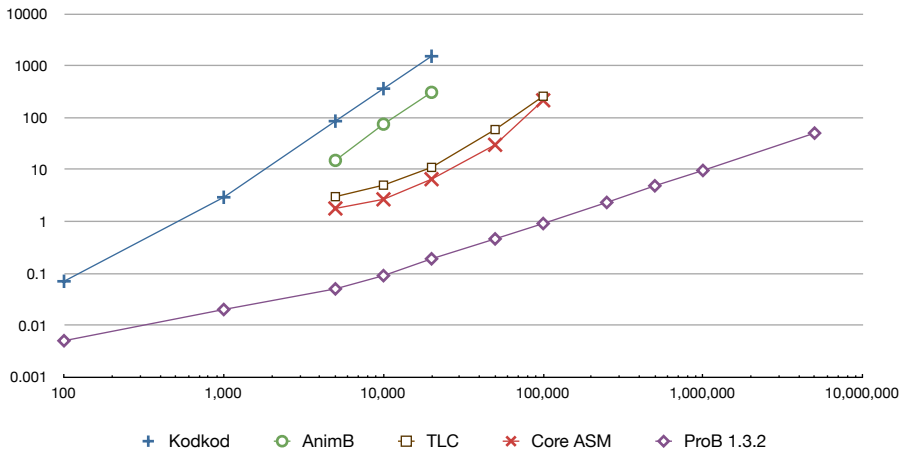

Fig. 6. Log-log Plot of the performance of ProB compared to Kodkod, TLC and AnimB for manipulating large sets

For efficiency, it is important to try to use the fully determined storage formats for large sets as much as possible. For example, in our STS application, not all constants are explicitly valued. As we have already seen, some have to be computed (e.g., using the override operator, as seen in Section 1). Furthermore, the models also contain abstract constants, which do not appear in the Ada code, and hence have to be inferred by our tool using the gluing invariants.

Example 1. For example, the harder model from Section 4 contains, amongst others, the abstract constants `cfg_cdv_block` and `cfg_cdv_aig` which are not explicitly valued. The properties of the model contain the following relevant conjuncts (in that order):

```
...
& cfg_cdv_i = (0 .. 56) * {2} <+ {1 |-> 0,2 |-> 0,3 |-> 0,4 |-> 0,5 |-> 0,6 |-> 0,7 |-> 0,8 |-> 0,9 |-> 0,
                10 |-> 0,11 |-> 0,12 |-> 0,13 |-> 0,14 |-> 0,15 |-> 0,16 |-> 0,17 |-> 0,18 |-> 0,19 |-> 0,
                20 |-> 1,21 |-> 1,22 |-> 1,23 |-> 1,24 |-> 1,25 |-> 1,26 |-> 1,27 |-> 1,28 |-> 1,29 |-> 1,
                30 |-> 1,31 |-> 1,32 |-> 1,33 |-> 1,34 |-> 1}
...
& cfg_cdv_aig <: t_cdv_acs                    /* Property 1 */
& cfg_cdv_block <: t_cdv_acs                  /* Property 2 */
...
& cfg_cdv_aig /\ cfg_cdv_block = {}           /* Property 3 */
& cfg_cdv_aig \/ cfg_cdv_block = t_cdv_acs    /* Property 4 */
...
& cfg_cdv_aig = cfg_cdv_i~[{c_cdv_aig}]       /* Property 5 */
& cfg_cdv_block = cfg_cdv_i~[{c_cdv_block}]   /* Property 6 */
...
& c_cdv_aig = 0
& c_cdv_block = 1
...
& t_cdv_acs = 1 .. 34
...
```

As we can see, there are six properties which talk directly about `cfg_cdv_block` and `cfg_cdv_aig`. The last one can be used to compute the abstract constant `cfg_cdv_block` efficiently, provided we use the information that `c_cdv_block = 1`, which appears later in the properties. The challenge is to find a way to compute the abstract (and concrete) constants in an efficient way, ideally using the new data structures from Section 3.1.

To address this challenge, we have improved the constraint propagation mechanism inside the ProB kernel. The previous version of ProB [27] basically had three constraint propagation phases: deterministic propagation, non-deterministic propagation and full enumeration. The new kernel now has a much more fine-grained constraint propagation, with arbitrary priorities. Every kernel computation gets a priority value, which is the estimated branching factor of that computation. A priority number of 1 corresponds to a deterministic computation. For example, the kernel computation associated with, $x = z$ would have a priority value of 1 while $x \in \{1, 2, 3\}$ would have a priority value of 3. A value of 0 indicates that the computation will yield a fully determined value. At every step, the kernel chooses the computation with the lowest priority value.

Take for example the predicate `x:NAT +-> NAT & x={y|->2} & y=3`. Here, `y=3` (priority value 0) would actually be executed before `x={y|->2}`, and thus ensure that afterwards a fully determined AVL-tree would be constructed for `x`. The check `x:NAT +-> NAT` is executed last, as it has the highest priority value (and thus the lowest priority).

In Example 1 above, the equality concerning `cfg_cdv_i` would be executed first, resulting in an AVL-tree representation of the relation (containing 57 tuples). Then `c_cdv_block = 1` would be executed, whereafter the properties 5 and 6 would be executed, resulting in AVL-tree representation for `cfg_cdv_aig` and `cfg_cdv_block`. Finally, properties 1–4 would be verified.

Compared to the old approach, enumeration can now be mixed with other computations and may even occur before other computations if this is advantageous. Also, there is now a much more fine-grained selection among the non-deterministic computations. Take for example, the following predicate:
`s1 = 9..100000 &  s2 = 5..100000 & s3 = 1..10 & x:s1 & x:s2 & x:s3`. The old version of ProB would have executed `x:s1` before `x:s2` and `x:s3`. Now, `x:s3` is chosen first, as it has the smallest possible branching factor. As such, ProB very quickly finds the two solutions $x = 9$ and $x = 10$ of this predicate.

Finite Domain Constraints

One further challenge arose in the Paris Line 1 models (see Section 5.1). These contained universally quantified formulas such as the following one:

```
!(aa,bb).(aa : type_a & bb : INTEGER &
             inv_deb(aa) <= bb  &  bb <= inv_fin(aa)
             =>
     bb : type_l)
```

Let us assume that `type_a` has a cardinality of 123 (which it actually does in the concrete example). When expanding the forall, our constraint kernel would chose to execute `aa : type_a` first, but it would still be left with 2^{32} possibilities for `bb` for every value of `aa`; most values of `bb` would make the predicate `inv_deb(aa) <= bb & bb <= inv_fin(aa)` false.

Our solution is to store domain information for every integer value, whose precise value is not yet known. Initially, when examining the premise of the forall, the predicate `bb : INTEGER` would set the domain of `bb` to $-2147483648..2147483647$. This then gets narrowed down by the first comparison `inv_deb(aa) <= bb` to $d..2147483647$, once the value of $d =$ `inv_deb(aa)` is known. The second comparison would then narrow down the domain further to $d..f$, once the value $f =$ `inv_fin(aa)` is known. As a consequence, `bb` would be enumerated in a much tighter range (sometimes even not at all, in case $d > f$).

To store the domain information, we have tied in the CLP(FD) solver of SICStus Prolog [12]. Note that this provides further advantages for a broad range of applications.

3.5. Advantages and Difficulties of Constraint Propagation

The constraint propagation as employed by ProB has big advantages in solving complicated predicates, e.g., arising in the industrial applications in this paper. Let us illustrate this advantage on the well-known n-Queens puzzle[9] expressed in B as follows:

$$q : 1..n \rightarrowtail\!\!\!\!\!\rightarrow 1..n \ \wedge \forall(i,j).(i : 1..n \wedge j : 2..n \wedge j > i \Rightarrow q(i) + j - i \neq q(j) \wedge q(i) - j + i \neq q(j))$$

ProB can deal with this problem quite effectively, taking, e.g., 0.01 seconds to find the first solution for $n = 8$ and solving the predicate for up to $n = 17$ in less than 0.1 seconds each on a MacBook Pro 3.06 GHz Core2 Duo. For $n = 70$ it is solved in about 9 seconds. A tool such as AnimB [32], which is written in Java and evaluates predicates from left-to-right, can only solve this for $n = 5$. (Strangely, AnimB cannot solve it for $n = 4$; nor can it determine that there are no solutions for $n = 3$. Changing the order of the predicates does not help either.) In a similar fashion, AnimB cannot solve the predicate from Example 1 stemming from our case study either.

In TLA$^+$ [23] the above predicate can be expressed as follows:

```
/\ q' \in [1..n -> 1..n]
/\ \A i \in 1..n : (\A j \in 2..n : i<j => q'[i] # q'[j] /\ q'[i]+i-j # q'[j] /\ q'[i]-i+j # q'[j])
```

The model checker TLC (version 3.5 of the TLA Toolbox) [43], is written in Java and is capable of evaluating complicated predicates and finding solutions for variables. Like AnimB, TLC deals with conjuncts from left-to-right (see page 239 in Chapter 14 of [23]). (One cannot change the order of the two conjuncts in the above example, otherwise TLC complains that q is undefined.) TLC can solve this predicate for n up to 6 quickly, but already takes 10 seconds for $n = 7$ and 4 minutes and 3 seconds for $n = 8$ (again on a MacBook Pro 3.06 GHz Core2 Duo), i.e., four orders of magnitude slower than ProB. For $n = 9$, TLC took over 1 hour and 45 minutes, i.e., more than 5 orders of magnitude slower than ProB(0.02 seconds). Note that we are only testing TLC's capability to solve predicates, not its (very effective) disk-based model checking capabilities.

We have also experimented with Alloy [20] (version 4.1.10), which does not treat conjuncts from left-to-right but translates the predicates to propositional logic formulas fed to a SAT solver. Solving the equivalent Alloy model for $n = 8$ takes 0.80 seconds with the default SAT4J sat solver. With minisat rather than SAT4J as backend, Alloy takes 0.24 seconds. In both cases, this is considerably faster than TLC. For $n = 15$ it takes Alloy 9.84 seconds with the default SAT4J solver and 2.08 seconds with minisat (compared to 0.06 seconds with ProB). For $n = 32$ it takes Alloy 32 minutes and 5 seconds with the default SAT4J solver and 4 minutes 5 seconds with minisat (compared to 0.52 seconds with ProB). (Note that the Alloy model uses the built-in Int type with a bit width of 5 for n up to 15, a bit width of 6 for $n > 15$ and $n < 32$, etc.)

As another small experiment, let us check whether two graphs with $n = 9$ nodes of out-degree exactly one are isomorphic by checking for the existence of a permutation p with $p \in 1..n \rightarrowtail\!\!\!\!\!\rightarrow 1..n \wedge \forall i.(i \in 1..n \Rightarrow p(graph1(i)) = graph2(p(i)))$. For the two graphs $graph1 = \{1 \mapsto 3, 2 \mapsto 3, 3 \mapsto 6, 4 \mapsto 6, 5 \mapsto 6, 8 \mapsto 9, 9 \mapsto 8, 6 \mapsto 6, 7 \mapsto 7\}$ and $graph2 = \{2 \mapsto 5, 3 \mapsto 5, 4 \mapsto 5, 6 \mapsto 4, 7 \mapsto 4, 1 \mapsto 9, 9 \mapsto 1, 5 \mapsto 5, 8 \mapsto 8\}$, TLC finds a permutation $([6, 7, 4, 2, 3, 5, 8, 1, 9])$ after 2 hours 6 minutes and 28 seconds; ProB takes 0.06 seconds to find the same solution for p, (and 0.1 seconds to find all 8 solutions for p), while Alloy takes 0.11 seconds with SAT4J and 0.05 seconds with minisat.

These small examples highlight the potential of constraint propagation approaches for solving logical predicates (but the example is by no means meant to reflect on the general performance for the various tools studied). However, the flexible constraint propagation of ProB also comes at a price, compared to other tools (such as AnimB, Brama or TLC) which evaluate predicates strictly from left-to-right:

- Unsatisfiable predicates:
  If a predicate is unsatisfiable (such as the guard of an operation or the properties clause of a machine), it is more difficult to pinpoint which conjunct(s) caused the predicate to become unsatisfiable. First, certain predicates (such as the properties of a machine) are decomposed into connected components. This improves efficiency but also helps the user pinpointing unsatisfiable components. Second, ProB provides

[9] http://en.wikipedia.org/wiki/Eight_queens_puzzle

a debugging command for predicates, which adds the conjuncts from left-to-right until no solution can be found. Finally, ProB's graphical formula viewer [30] computes a "maximal" satisfiable subpredicate.

- Time-out:
  It is not possible to "time out" on individual conjuncts. In other words, one cannot simply skip over conjuncts which take too long to solve, as the constraint solving of all conjuncts is intertwined. (However, it is possible to skip over components which take too long to solve.)
- Well-Definedness:
  It is more difficult to detect undefined expressions. Take for example the predicate $y \neq 0 \wedge x/y = 1$; it could happen that x/y is computed before $y \neq 0$ prunes the computation. As such, we cannot simply raise an error message if a division by zero (or other undefined operation) occurs. ProB employs two solutions for this problem:
  - when a division by zero is encountered, then ProB raises a conditional error message. This error message will suspend until the enumeration of all values has finished; if the surrounding predicate fails then no error message will be raised.
  - for efficiency reasons, some well-definedness errors are not caught by default, but the computation fails. This is the case for applying a function outside of its domain.[10] Thus, the evaluation of `{1|->1}(2)=3` will fail, as will its negation `not({1|->1}(2)=3)`. Thus, by evaluating a predicate both positively and negatively one can detect undefined predicates. This technique is applied by our formula viewer, as well as the assertion checking for the industrial applications in this paper.

  Both approaches encode a very liberal approach to well-definedness for conjunctions, which is basically independent of the order of the conjuncts. I.e., both $y \neq 0 \wedge x/y = 1$ and $x/y = 1 \wedge y \neq 0$ are considered to be well-defined. For conjunctions, our approach corresponds to the D-system of [3], where a conjunct $P \wedge Q$ is considered well-defined if either both P and Q are well-defined, P is well-defined and false, or Q is well-defined and false. However, for disjunctions our approach corresponds to the L-system of [3], where $P \Rightarrow Q$ is well-defined if P is well defined and Q is well-defined whenever P is true. In summary, ProB supports the L-system of [3] (also being used by Rodin), even though some formulas well-defined under the D-system are also accepted.

3.6. Summary

In summary, driven by the requirements of the industrial application, we have improved the scalability of the ProB kernel. This required the development of a new data structure to represent and manipulate large sets and relations. A new, more fine grained constraint propagation algorithm was also required to ensure that this data structure could actually be used in the industrial application.

4. The San Juan Case Study

As already mentioned, in order to evaluate the feasibility of using ProB for checking the topology properties, Siemens sent the STUPS team at the University of Düsseldorf the models for the San Juan case study on the 8th of July 2008. There were 23,000 lines of B spread over 79 files, two of which were to be analysed: a simpler model (acs_as_env_cfg_aiguille.mch) and a hard model (acs_as_env_cfg_ipart.mch). The harder model acs_as_env_cfg_ipart.mch contains 226 properties and 147 assertions. It then took us a while to understand the models and get them through our new parser, whose development was being finalised at that time.

On 14th of November 2008 we were able to animate and analyse the first model. This uncovered one error in the assertions. However, at that point it became apparent that a new data structure would be needed to validate bigger models. At that point the developments described in Section 3 were undertaken. On the 8th of December 2008 we were finally able to animate and validate the complicated model. This revealed four errors.

Note that we (the STUPS team) were not told about the presence of errors in the models (they were not even hinted at by Siemens), and initially we believed that there was still a bug in ProB. Luckily, the

[10] Even though ProB has a preference were well-definedness checking for function application can be turned on.

Activity	Resource Usage
B source files (12 Machines, 23 Definition files)	
↓	
1. Parsing:	13.7 sec
↓	
Prolog AST File	17.6 MB
↓	
2. Loading:	19.16 sec
3. Type Checking:	3.59 sec
↓	
typed internal Prolog data structure	7 MB
↓	
4. Properties Solving (225 properties):	0.33 sec
↓	
values for all concrete and abstract constants	0.69 MB
↓	
5. Assertion Checking (148 assertions):	40.49 sec

Fig. 7. Some statistics about the most complicated machine (acs_as_env_cfg_ipart.mch) of the San Juan Case study

errors were in the concrete data values. Furthermore, ProB found *exactly* the same errors that Siemens had uncovered themselves by manual inspection.

The manual inspection of the properties took Siemens several weeks (about a man month of effort). Checking the properties took 4.15 seconds, and checking the assertions took 1017.7 seconds (i.e., roughly 17 minutes) using ProB 1.3.0-final.4 on a MacBook Pro with 2.33 GHz Core2 Duo. With version 1.3.1, the runtime was further improved, to below 5 minutes, and now with version 1.3.2 on a more recent laptop the assertion checking is done in less than a minute. Statistics for the complete process, from parsing to assertion checking can be found in Figure 7. Note that ProB was not used in its capacity as a model checker here: ProB was used to find values for the concrete and abstract constants which satisfy the PROPERTIES clause and then was used to verify the truth-value of all 148 assertions. Note that all properties and assertions were checked twice, both positively and negatively, in order to detect undefined predicates (e.g., $0/0 = 1$ is undefined). We return to this issue in Section 6.

The four false formulas found by ProB are the following ones (see also Figure 8):

1. `ran(cfg_aig_cdv_encl) = cfg_cdv_aig`
2. `cfg_ipart_aig_tild_liste_i : t_liste_acs_2 --> t_nb_iti_partiel_par_acs`
3. `dom(t_iti_partiel_acs <| cfg_ipart_cdv_dest_aig_i |> cfg_cdv_aig) \/ dom(t_iti_partiel_acs <| cfg_ipart_cdv_dest_saig_i |> cfg_cdv_block) = t_iti_partiel_acs`
4. `ran(aa,bb|aa:t_aig_acs & bb:t_cdv_acs & bb:cfg_aig_cdv_encl_liste_i[ (cfg_aig_cdv_encl_deb(aa)..cfg_aig_cdv_encl_fin(aa))]) = cfg_cdv_i~[c_cdv_aig]`

Inspecting the Formulas

Once our tool has uncovered unexpected properties of a model, the user obviously wants to know more information about the exact source of the problem.

This was one problem in the Atelier B approach: when a proof fails it is very difficult to find out why the proof has failed, especially when large and complicated constants are present.

To address this issue, we have developed an algorithm to inspect the truth-values of B predicates, as well as all sub-expressions and sub-predicates. The whole is assembled into a graphical tree representation. (An earlier version of the graphical viewer is described in [30].)

A graphical visualisation of the fourth false formula is shown in Figure 9. For each expression, we have two lines of text: the first indicates the type of the node, i.e., the top-level operator. The second line gives the value of evaluating the expression. For predicates, the situation is similar, except that there is a third

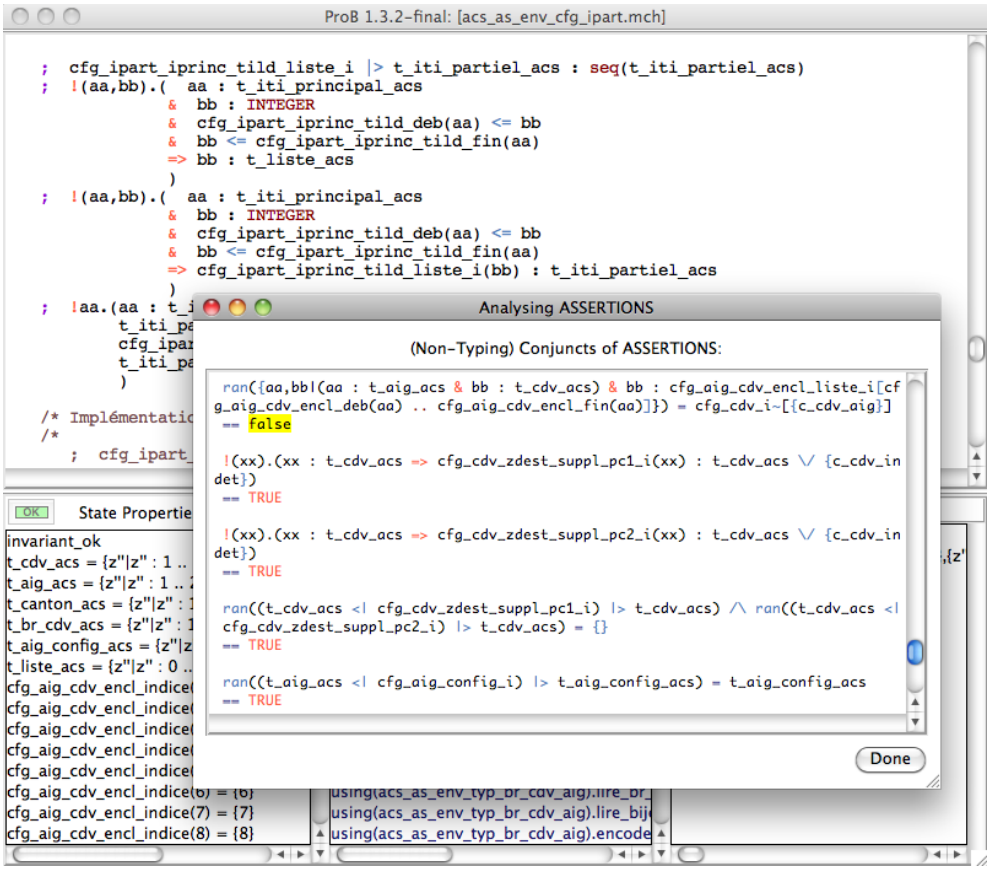

Fig. 8. Analysing the Assertions

line with the formula itself and that the nodes are coloured: true predicates are green and false predicates are red.

Note that the user can type custom predicates to further inspect the state of the variables of a specification. Thus, if the difference between the range expression and `cfg_cdv_i [c_cdv_aig]` is not sufficiently clear, one can evaluate the set difference between these two expressions. This is shown in Figure 10, where we can see that the number 19 is an element of `cfg_cdv_i [c_cdv_aig]` but not of the range expression.

In summary, the outcome of this case study was extremely positive: a man-month of effort has been replaced by a few minutes of computation on a laptop. Siemens are now planning to incorporate ProB into their development life cycle, and they are hoping to save a considerable amount of resources and money. For this, validation of the ProB tool is an important aspect, which we discuss in Section 6.

5. First Industrial Applications

5.1. Paris Line 1 Deployment

In October 2009, ProB was applied for the first time on an active development, concurrently to the classical approach using Atelier B and manual inspection (cf., Fig. 2).

Siemens is involved in the automatisation of the Line 1 of the Paris Métro. The line will be gradually upgraded to driverless trains, while the line remains in operation. So far, we have inspected the first component to be delivered by Siemens, namely the PAL (Pilote Automatique Ligne). The B models of the PAL

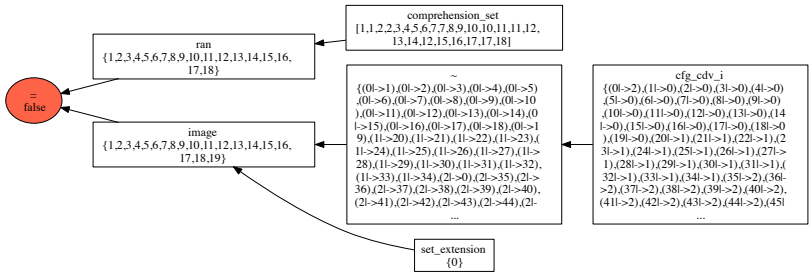

Fig. 9. Analysing the fourth false assertion from acs_as_env_cfg_ipart

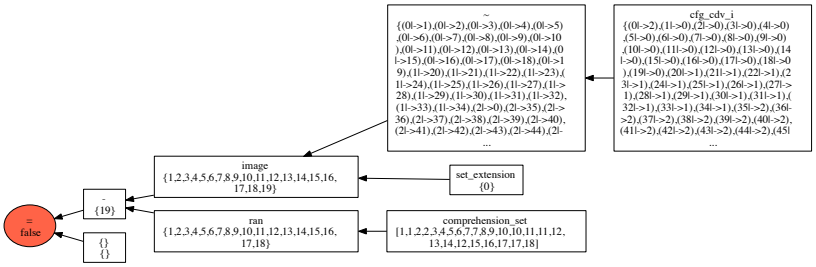

Fig. 10. Analysing a variation of the fourth false assertion

consisted of 74 files with over 10,000 lines of B. In all 2024 assertions about the concrete data of the PAL needed to be checked.

Again, ProB found all problems (12 in all) in under 5 minutes. These problems have of course been examined and fixed by Siemens before delivery. There were again no false alarms or mistakes by ProB, compared to the validation done by Siemens.

Initially, few minor tweaks were required to get the models through our type checker. Indeed, some of the models introduced DEFINITIONS which were overwriting visible constants from included machines. ProB did consider this to be an error, whereas Atelier B accepts this. This has been fixed now: ProB no longer generates an error but only a warning in those cases. Now, the PAL models can be loaded without modification into ProB.

Also note that we had to improve the ProB kernel for a few operators not yet encountered in the San Juan models (such as the `iterate` operator for relations, which did not yet use our new data structure for large relations). This took only about a couple of hours.

In response to a Siemens requirement, we have also made assertion checking possible from within a command-line version of ProB. This allows to run our tool in batch mode on a large number of files, and collect the results. Below is the summary information obtained by running our tool on the main PAL files: (runtimes are in milliseconds and only measure the time spent checking the assertions using the 64-bit command line version of ProB 1.3.2):

```
SUMMARY of checking ASSERTIONS
 total: Total number of conjuncts
 true: Total number of true conjuncts
 false: Total number of false conjuncts
 unknown: Total number of unknown conjuncts (because of timeout or undefinedness)
 timeout: Total number of conjuncts were time_out occurred (in true or false branch)
 runtime: Total runtime in ms for checking conjuncts
cbtc_mes_as_env_inv_easitf_bs.mch
 --> [total/127,true/127,false/0,unknown/0,timeout/0,runtime/370]
cbtc_mes_as_env_inv_etors_bs.mch
```

```
 --> [total/142,true/142,false/0,unknown/0,timeout/0,runtime/1230]
cbtc_mes_as_env_inv_stors_bs.mch
 --> [total/148,true/147,false/1,unknown/0,timeout/0,runtime/80]
par1_mes_as_env_inv_etors_bs.mch
 --> [total/125,true/125,false/0,unknown/0,timeout/0,runtime/40]
par1_pans_as_env_inv_etors_bs.mch
 --> [total/188,true/188,false/0,unknown/0,timeout/0,runtime/50]
par1_pans_as_env_inv_etors_dist_bs.mch
 --> [total/184,true/184,false/0,unknown/0,timeout/0,runtime/80]
par1_pans_as_env_inv_mesitf_bs.mch
 --> [total/182,true/182,false/0,unknown/0,timeout/0,runtime/40]
par1_pans_as_env_inv_pasitf_bs.mch
 --> [total/177,true/176,false/1,unknown/0,timeout/0,runtime/30]
par1_pans_as_env_inv_se_dj_bs.mch
 --> [total/176,true/174,false/2,unknown/0,timeout/0,runtime/30]
par1_pans_as_env_inv_ss_bs.mch
 --> [total/192,true/192,false/0,unknown/0,timeout/0,runtime/340]
par1_pans_as_env_inv_stors_bs.mch
 --> [total/192,true/192,false/0,unknown/0,timeout/0,runtime/50]
par1_pans_as_env_inv_tcs_bs.mch
 --> [total/191,true/183,false/8,unknown/0,timeout/0,runtime/70]
-----
TOTALS: total/2024 true/2012 false/12 unknown/0 timeout/0 runtime/2410
```

5.2. Sao Paolo Line 4

The next task was the validation of the CBTC Ground controller for Line 4 of Sao Paolo, which began operation in May 2010.

The first author received the models on March 11 2010, because Siemens was unable to validate one crucial property (neither with Atelier B, nor by "hand"). The models consisted of 210 files with over 30,000 lines of B and over 2500 assertions. It then took about a day of the first author's time to validate the crucial property using ProB.

The reasons the validation took a day rather than minutes were the following:

- The generated B models contained syntax errors and locating the error within 210 files was difficult. The parser of ProB has now been improved to also output information about the file containing the error. The syntax errors (missing semicolons after definition file imports) were not found by Atelier B due to a bug in the parser. Siemens have now fixed their tool to avoid those syntax errors in the future.
- The models highlighted performance and memory issues with some of the B operators; these operators were not required in the previous case studies. The ProB kernel has again been improved to deal with those operators.
- The models contained many inconsistencies in the PROPERTIES. This makes starting ProB more difficult. Note that the Paris Line 1 models from Section 5.1 contained no inconsistencies, and the San Juan case study contained only a single inconsistency.
  To address this issue better in future applications, ProB now partitions the PROPERTIES into independent components and detects inconsistent components. This helps the user to isolate the problem more quickly.

Some of the other B machines of the Sao Paolo Line 4 made use of infinite sets (e.g., setting `s = INTEGER - {x}` and then later checking `y:s`). ProB now detects those infinite complement sets and keeps them symbolic, even if ProB is not in symbolic mode. There is also support for performing certain B operations on those symbolic sets (union, intersection, membership test, ...). All of the additional improvements to ProB took about a week to implement.

The crucial property which defied validation by Atelier B and by humans was the following one. The ProB analysis result can be found in Figure 11.

```
(!iti_ztr.(iti_ztr: t_iti_ztr_pas =>
  !(cv_ztr1,cv_ztr2).(cv_ztr1: t_cv_ztr & cv_ztr2: t_cv_ztr &
     cv_ztr1: {aa,bb | aa: t_iti_ztr_pas & bb: t_cv_ztr &
              bb: inv_iti_ztr_cv_liste_i[inv_iti_ztr_cv_deb(aa)..inv_iti_ztr_cv_fin(aa)]}[{iti_ztr}] &
     cv_ztr2: {aa,bb | aa: t_iti_ztr_pas & bb: t_cv_ztr &
```

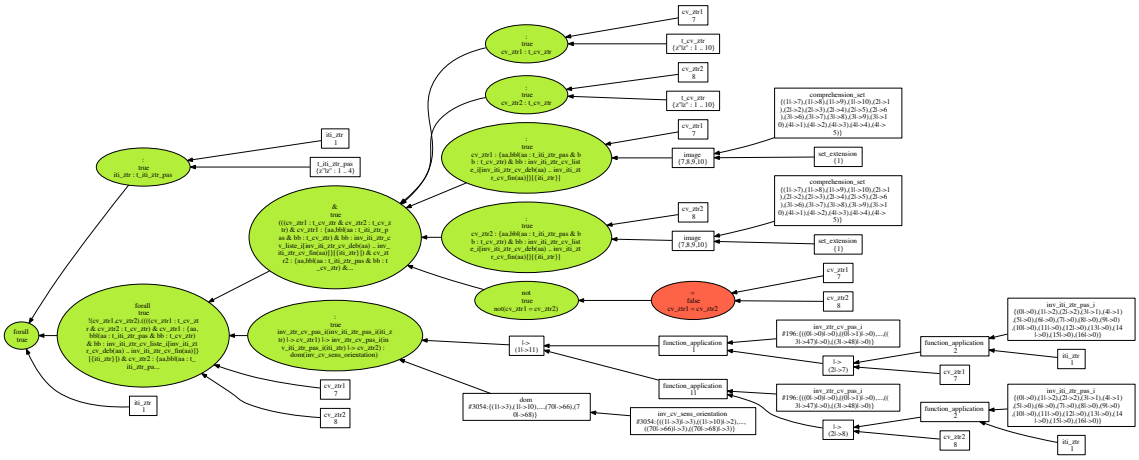

Fig. 11. ProB output for a complicated property of Sao Paolo Line 9

```
        bb: inv_iti_ztr_cv_liste_i[inv_iti_ztr_cv_deb(aa)..inv_iti_ztr_cv_fin(aa)]}[{iti_ztr}] &
 not(cv_ztr1 = cv_ztr2)
=>
 inv_ztr_cv_pas_i(inv_iti_ztr_pas_i(iti_ztr)|->cv_ztr1)
                  |->inv_ztr_cv_pas_i(inv_iti_ztr_pas_i(iti_ztr)|->cv_ztr2)
  :
 dom(inv_cv_sens_orientation))))
```

The ProB output from Figure 11 was used in the official Siemens validation report. Also note that for the first time, ProB also detected errors that were *not* detected by the existing approach of Siemens.

5.3. Barcelona Line 9 and CDGVAL

After the delivery of the Sao Paolo line, ProB was applied in May 2010 to validate the date of the zone controller of the Barcelona Line 9, which "will be one of the longest automatic metro lines in Europe."[11].

First, ProB detected syntax errors in the definitions of the generated B machines. For example, one definition file contained the following line, where an operator is missing:

```
co_nb_max_heure_il_par_pas  ==   2 ** 24  1;
```

These errors were not detected by the Atelier B parser, as the offending definitions were not actually used in the rest of the model.

Once the syntax errors were corrected, the data validation was performed independently and with success by Siemens. As a minor improvement, we made sure that ProB tries to detect infinite lambda expressions and set comprehensions, ensuring that those are not expanded, even if ProB is not in symbolic mode. For example, the following two lambda expressions appear in the Barcelona Line 9 models:

```
abs = %xx.(xx : INTEGER | max ({xx, -xx})) &
sqrt = %xx.(xx : NATURAL | max({yy | yy : INTEGER & yy * yy <= xx}))
```

After that, Siemens used ProB for data validation for the CDGVAL (Charles de Gaulle Véhicle Automatique Léger) automated shuttle at the Charles de Gaulle airport in Paris. The validation was performed completely independently and successfully by Siemens; no adaptation of ProB was performed.

[11] http://en.wikipedia.org/wiki/Barcelona_Metro_line_9

6. Validation of ProB

In this case study, ProB was compared with Atelier B. For this specific use, the performance of ProB is far better than the performance of Atelier B. However, in contrast to Atelier B, ProB is not yet qualified for use within a development life cycle producing SIL 4 (the highest safety integrity level; see, e.g., [35]) certified systems. To be able to routinely use ProB, Siemens have to be able to rely on ProB's output if it evaluates a property to true. There are two ways this can be achieved:

- Using a second, independently developed tool to validate the data properties.
  One possibility would be Atelier B, but as we have already seen it is currently not capable to deal with the more complicated properties. Another possibility would be to use another animator, such as Brama [36] or AnimB [32]. These tools were developed by different teams using very different programming languages and technology, and as such it would be a strong safety argument if our tool and either Brama or AnimB returned the same verdict. This avenue is being investigated, but note that Brama and AnimB are much less developed as far as the constraint solving capabilities are concerned. We discuss these two tools in more detail later in Section 7.
- Validate ProB, at least those parts of ProB that have been used for checking the properties.
  There are no general requirements for using a tool within a SIL 4 development chain; the amount of validation depends on the criticality of the tool in the development or validation chain. In this case, Siemens require:
  - a list of all critical modules of the ProB tool, i.e., modules used by the data validation task, that can lead to a property being marked wrongly as fulfilled
  - a complete coverage of these modules by tests.
  - a validation report, with description of ProB's functions, and a classification of functions into critical and non-critical, as long with a detailed description of the various techniques used to ensure proper functioning of ProB.

  We are currently pursuing the second option, and provide some details in the remainder of this section.

Validation Techniques

The source code of ProB contains >40,000 lines of Prolog, >7,000 lines of Tcl/Tk, > 5,000 lines of C (for LTL and symmetry reduction), 1,216 lines of SableCC grammar along with 9,025 lines of Java for the parser (which are expanded by SableCC into 91,000 lines of Java). In addition, there are > 5,000 lines of Haskell code for the CSP parser and about 50,000 lines of Java code for the Rodin [2] plugin. These statistics concern version 1.3.0 of ProB.

1. Unit Tests:
   ProB contains over a 1,000 manually entered unit tests at the Prolog level. For instance, these check the proper functioning of the various core predicates operating on B's data structures. For example, it is checked that $\{1\} \cup \{2\}$ evaluates to $\{1, 2\}$.
   In addition, we have now also added an automatic unit test generator, which tests the ProB kernel predicates with many different scenarios and set representations. For example, starting from the call `union([int(1)],[int(2)],[int(1),int(2)])`, the test generator will derive 1358 unit tests. It will use AVL or closure representations of the sets; it will swap the order of the first two arguments, as union is commutative; it will check various orderings in which the information about the sets can arrive, e.g., it could be that first the result of the union is known, then the second argument.
2. Run Time Checking:
   The Prolog code contains a monitoring module which — when turned on — will check pre- and post-conditions of certain predicate calls and also detect unexpected failures. Many kernel predicates also check for unexpected arguments. All of this overcomes to some extent the fact that Prolog has no static typing.
3. Integration and Regression Tests:
   ProB contains 220 regression tests which are made up of B models along with saved animation traces. These models are loaded, the saved animation traces replayed and the models are also run through the model checker. These tests have turned out to be extremely valuable in ensuring that a bug once fixed

remains fixed. They are also very effective at uncovering errors in arbitrary parts of the system (e.g., the parser, type checker, the interpreter, the ProB kernel, ...).

4. Self-Model Check:
   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.
   Concretely, ProB now checks itself for over 500 mathematical laws. There are laws for booleans (39 laws), arithmetic laws (40 laws), laws for sets (81 laws), relations (189 laws), functions (73 laws) and sequences (61 laws), as well as some specific laws about integer ranges (24 laws) and the various basic integer sets (7 laws). Figure 12 contains some of these laws about functions.
   The self-model check has been very effective at uncovering errors in the ProB kernel and interpreter. So much so, that even two errors in the underlying SICStus Prolog compiler were uncovered via this approach:
   - The Prolog `findall` did sometimes drop a list constructor, meaning 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. For example, the ProB model checker found the value of `SS = {{},{{}}}` , violating the following two laws:
     - `POW1(SS) = POW(SS) - {{}}`
     - `FIN1(SS) = FIN(SS) - {{}}`

     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.

   Note: these problems would not have been detected by validating or proving the code of ProB correct. It was essential to test the actual code of ProB. The model checker together with the mathematical laws enabled this testing to be performed very effectively.
5. Positive and Negative Evaluation:
   As already mentioned, all properties and assertions were 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 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 example, given the Prolog rule `p(eq(X,Y)) :- X=Y.` the query `not(p(eq(X,0)))` would fail, i.e., one could erroneously conclude that there is no value `X` which is different from `0`. In ProB, we have a dedicated predicate for the negation, which will suspend until it can determine the result correctly. For the above example, one could write `not_p(eq(X,Y)) :- dif(X,Y).`
   With these two predicates we can uncover undefined predicates: if for a given B predicate both the positive and negative Prolog predicates fail then the formula is undefined. For example, the property `x = 2/y & y = x-x` over the constants `x` and `y` would be detected as being undefined, and would be visualised by our graphical formula viewer as in Figure 13 (yellow and orange parts are undefined).
   In the context of validation, this approach has another advantage: 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.
   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).

Code Coverage

The above validation techniques are complemented by code coverage analysis techniques. In particular, we try to ensure that Points 1 and 4 above cover all predicates and clauses of the ProB kernel. As we are not

```
law1 ==  (dom(ff\/gg) = dom(ff) \/ dom(gg));
law2 ==  (ran(ff\/gg) = ran(ff) \/ ran(gg));
law3 ==  (dom(ff/\gg) <: dom(ff) /\ dom(gg));
law4 ==  (ran(ff/\gg) <: ran(ff) /\ ran(gg));
law5 ==  ( (ff \/ gg)~ = ff~ \/ gg~);
law6 ==  (dom((ff ; (gg~))) <: dom(ff));
...
law10 == (ff : setX >->> setY  <=>  (ff : setX >-> setY  &  ff~: setY >-> setX));
law11 == (ff : setX >+> setY  <=> (ff: setX +-> setY &
                     !(xx,yy).(xx:setX & yy:setX & xx/=yy & xx:dom(ff) &
                   yy: dom(ff)  => ff(xx)/=ff(yy)))) ;
law12 == (ff : setX +->> setY  <=>  (ff: setX +-> setY &
                                    !yy.(yy:setY => yy: ran(ff))));
```

Fig. 12. A small selection of the laws about B functions

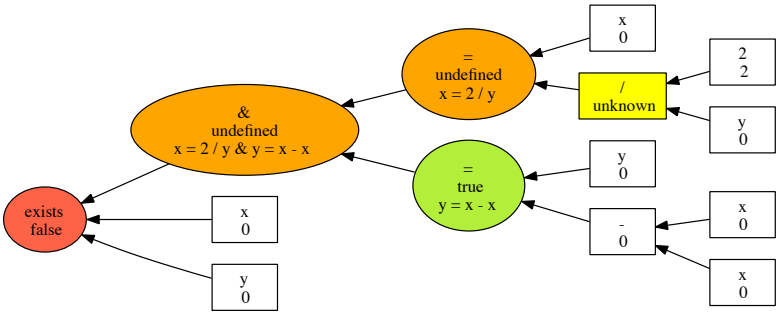

Fig. 13. Visualising an undefined property

aware of any tool that computes code coverage for Prolog, we have developed our own code coverage tool for SICStus Prolog. The tool uses Prolog's term expansion facility to keep a record of which program points are covered. With our tool we can detect:

- completely uncovered Prolog predicates,
- clauses of a Prolog predicate that are never called (i.e., no call unifies with the head of the clause),
- clauses of a Prolog predicate that never succeed,
- choice points inside a clause which are not completely covered,
- execution paths through a clause which are not covered.

Our tool also contains a graphical front-end that visualises the information on the source code. So far, this has been very useful in extending our unit tests and mathematical laws. As an example, 96.5 % of the clauses of the `kernel_mappings` module are now covered by the unit tests and the mathematical laws. The only uncovered clauses relate to the Z compaction operator, and one error condition that was not triggered by the tests. In summary, the code coverage has helped us write better tests and has allowed us to uncover a few undetected errors in the kernel.

Infrastructure Validation

We cannot hope to validate the entire environment in which ProB is run (Prolog compiler, operating system, hardware,...). But note that some of our tests exercise ProB with the complete infrastructure. As such, we have identified a bug in the parser (`FIN` was treated like `FIN1`), as well as the SICStus Prolog bugs mentioned above. The parser and type checker are two further components of ProB which we try to validate separately:

1. Validation of the parser:
   We execute our parser on a large number of our regression tests and pretty print the internal represen-

tation. We then parse the internal representation and pretty print it again, verifying (with `diff`) that we get exactly the same result. This type of validation can easily be applied to a large number of B machines, and will detect if the parser omits, reorders or modifies expressions, provided the pretty printer does not compensate errors of the parser. On the downside, the validation will only detect those errors in machines generated by the pretty printer, which may prevent us from catching errors which only appear in non-pretty printed machines, e.g. when parentheses in expressions are set incorrectly.

2. Validation of the type checker:
   For the moment we also read in a large number of our regression tests and pretty print the internal representation, this time with explicit typing information inserted. We now run this automatically generated file through the Atelier B type checker. With this we test whether the typing information inferred by our tool is compatible with the Atelier B type checker. (Of course, we cannot use this approach in cases where our type checker detects a type error.) Again, this validation can easily be applied to a large number of B machines. More importantly, it can be systematically applied to the machines that ProB validates for Siemens: provided the parser and pretty printer are correct, this gives us a guarantee that the typing information for those machines is correct. The latest version of ProB has a command to cross check the typing of the internal representation with Atelier B in this manner.

7. More Related Work, Conclusion and Outlook

More Related Work: Brama, and BZ-TT

We have already mentioned the Brama [36] and AnimB [32] animators. They both use the same predicate and expression evaluator for B expressions (written in Java, package `com.clearsy.predicateB`). Both Brama and AnimB require all constants to be fully valued, AnimB for the moment is not capable of enumerating functions, etc. Hence, they can not be used directly to validate all the properties in our case studies and industrial applications (e.g., AnimB cannot solve the predicate shown in Example 1). It should, however, be possible to apply AnimB or Brama only on those properties that apply fully valued concrete constants. Indeed, the transit operator RATP for Paris are doing just that, using the tool Ovado developed by ClearSy and using the underlying engine of Brama to cross-check the results provided by STS. Furthermore, like ProB, Ovado has the capability to keep set comprehensions in a symbolic form. Unfortunately, we do not have access to experience reports or performance results of RATP. However, we were able to compare ProB's output with Ovado's output for a particular benchmark model of ClearSy, confirming that both tools obtained the same validation result.

We have rerun the experiment from Section 3.3 using Brama. Brama does not run on the latest Rodin release nor on the latest Mac OS X operating system (which we used for the earlier experiments). Hence, we experimented with Brama version 0.0.22 on Rodin 0.9.2.1 on a 1.83 GHz, 2GB Mac Mini Core2 Duo running Mac OS X 10.5.8. As is to be expected, performance of Brama is very similar to AnimB. For limit=10,000, it takes Brama 1 minute 22 seconds to compute `numbers := numbers - ran(%n.(n:cur..limit/cur|cur*n))`. ProB 1.3.2 took 0.15 seconds on the same hardware. For limit=20,000 it takes Brama 5 minutes 23 seconds, compared to ProB which takes 0.30 seconds on the same hardware.

In addition to the animators Brama and AnimB we would like to mention BZ-TT [24], a test generation tool for B and Z specifications. A specification is translated into constraints and the CLPS-B constraint solver [11] is used to find boundary values and to determine test sequences [5]. BZ-TT is now part of a commercial product named LEIRIOS Test Generator. Like ProB, the core of BZ-TT is written in Prolog.

The BZ-TT tool is focused on test generation; many of the features required for the Siemens case study are *not* supported by BZ-TT (e.g., set comprehensions, machine structuring, definitions and definition files). Also, unfortunately, BZ-TT is no longer available for download. Still, we managed to obtain version BETA 1.00 of BZ-TT for Linux, which we ran on Ubuntu 9.4 with 512 MB using Parallels Desktop. We could not apply BZ-TT to the same Sieve experiment as above, due to the lack of support for lambda abstractions by BZ-TT. We thus experimented with a simplified version. Generally speaking, when working with sets larger than a thousand elements, BZ-TT would often fail to start up the animator.[12] When successful, it took BZ-TT, e.g., 87 seconds to compute `yy := 2..5000` and then another 108 seconds to compute `xx :=`

[12] The message being shown is {`ERROR: Memory allocation failed (upper 4 bits do not match MallocBase)`}.

`yy - {2,3,5,6,11,13,17,19,23,27}`. ProB took less than 0.01 seconds for the first operation and 0.04 seconds for the second one, on the same hardware. When doubling the set size, the runtimes were more than multiplied by four: computing `yy := 2..10000` took over 6 minutes (368 seconds) and `xx := yy - {2,3,5,6,11,13,17,19,23,27}` took about 8 and a half minutes (509 seconds). ProB took less than 0.01 seconds for the first operation and 0.09 seconds for the second one, on the same hardware. These experiments show that the BZ-TT set operations do not scale linearly, and indicate a performance difference of more than four orders of magnitude compared with our new version of ProB for sets between 5,000 and 10,000 elements.

In summary, none of the existing alternative animators for B seem to be able to deal well with large sets and relations.

Alternative Approaches

We have been — and still are — investigating alternative approaches for scalable validation of models, complementing ProB's constraint solving approach.

One candidate was the bddbddb package [42], which provides a simple relational interface to binary decision diagrams and has been successfully used for scalable static analysis of imperative programs. However, we found out that for dealing with the B language, the operations provided by the bddbddb package were much too low level (everything has to be mapped to bit vectors), and we abandoned this avenue of research relatively quickly.

We are also investigating whether the SMT solver Yices [15] could be used to complement ProB's constraint solving engine. First experiments with SAL (which is based upon Yices) were only partially successful: some simple examples with arithmetic give big speedups, but for more complicated data structures the translation to SAL breaks down (see also the translation from Z to SAL in [13] and the discussion about the performance compared to ProB in [29]). However, not all features of SAL are required and some useful features of Yices are not accessible via SAL. So, we plan to investigate this research direction further.

Conclusion and Outlook

In order to overcome the challenges of this case study, various research and development issues had to be addressed. We had to develop a new parser with an integrated type checker, and we had to devise a new data structure for large sets and relations along with an improved constraint propagation algorithm. The result of this study shows that ProB is now capable of dealing with large scale industrial models and is more efficient than Atelier B for dealing with large data sets and complex properties. About a man month of effort has been replaced by a few minutes of computation on standard hardware. Furthermore, ProB provides help in locating the faulty data when a property is not fulfilled. The latest version of ProB can therefore be used for debugging large industrial models.

In the future, Siemens plans to replace Atelier B by ProB for this specific use (data proof regarding formal properties). STS and the University of Düsseldorf will validate ProB in order to enable STS to use it within its development cycle for producing systems with SIL4 certification. We have described the necessary steps towards validation. In particular, we are using ProB's model checking capabilities to check ProB itself, which has amongst others uncovered two errors in the underlying Prolog compiler.

We also plan to work on even bigger specifications such as the model of the Canarsie line (the complete B model of which contains 273,000 lines of B [16], up from 100,000 lines for Météor [8]). As far as runtime is concerned, there are still possibilities for further improvement. Initially, it took ProB 17 minutes [28] to check all properties and assertions of the San Juan case study. With the additional improvements in the algorithms described in Section 3.2, we have already reduced this time to less than a minute. Further improvements in the ProB kernel are probably still possible. Parallelisation is also a potential avenue of further improvement: the individual assertions could actually be easily shared out amongst several computers. As far as memory consumption is concerned, for one universally quantified property we were running very close to the available memory (3 GB) in [28]. The new improved algorithms from Section 3.2 have alleviated this issue. Furthermore, we can compile ProB for a 64 bit system to increase the amount of available memory. We are also investigating the use ProB's symmetry reduction techniques [26, 41] inside quantified formulas, as a further way to improve ProB's performance.

Acknowledgements

First, we would like to thank the anonymous referees of FM'2009 and of Formal Aspects of Computing for their extensive and very thorough feedback, which helped us considerably to improve paper. We also would like to thank Jens Bendisposto for assisting us in various ways in writing the paper. We are also grateful to Roozbeh Farahbod and Stefan Hallerstede for encoding the Sieve Algorithm in CoreASM and TLA$^+$ for us.

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In *ICFEM06*, LNCS 4260, pages 588–605. Springer, 2006.

[3] 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*, LNCS 2272, pages 242–269, 2002.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools (Second Edition)*. Addison Wesley, 2007.

[5] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02*, pages 105–120, August 2002. Technical Report, INRIA.

[6] A. W. Appel. *Modern Compiler Implementation in Java (Second Edition)*. Cambridge University Press, 2002.

[7] F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: Roissy val. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 334–354. Springer-Verlag, 2005.

[8] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A successful application of B in a large project. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, LNCS 1708, pages 369–387. Springer, 1999.

[9] O. Boite. Méthode B et validation des invariants ferroviaires. Master's thesis, 2000. M´moire de DEA de logique et fondements de l'informatique.

[10] O. Boite. Automatiser les preuves d'un sous-langage de la méthode B. *Technique et Science Informatiques*, 21(8):1099–1120, 2002.

[11] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.

[12] M. Carlsson and G. Ottosson. An open-ended finite domain constraint solver. In H. G. Glaser, P. H. Hartel, and H. Kuchen, editors, *Proc. Programming Languages: Implementations, Logics, and Programs*, LNCS 1292, pages 191–206. Springer-Verlag, 1997.

[13] J. Derrick, S. North, and T. Simons. Issues in implementing a model checker for Z. In Z. Liu and J. He, editors, *ICFEM*, LNCS 4260, pages 678–696. Springer, 2006.

[14] D. Dollé, D. Essamé, and J. Falampin. B dans le tranport ferroviaire. L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques*, 22(1):11–32, 2003.

[15] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *CAV*, LNCS 4144, pages 81–94. Springer, 2006.

[16] D. Essamé and D. Dollé. B in large-scale projects: The Canarsie line CBTC experience. In *Proceedings B'07*, pages 252–254, 2007.

[17] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. *Fundam. Inform.*, 77(1-2):71–103, 2007.

[18] F. Fritz. An object oriented parser for B specifications. Bachelor's thesis, Institut für Informatik, Universität Düsseldorf, 2008.

[19] E. Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, Montreal, Canada, 1998. Available at `http://www.sablecc.org`.

[20] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.

[21] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[22] D. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, 1983.

[23] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[24] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.

[25] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[26] M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.

[27] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[28] M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models. In A. Cavalcanti and D. Dams, editors, *Proceedings FM 2009*, LNCS 5850, pages 708–723. Springer, 2009.

[29] M. Leuschel and D. Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Y. A. Ameur, F. Boniol, and V. Wiels, editors, *Proceedings Isola 2007*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*, pages 73–84. Cépaduès-Éditions, 2007.

[30] M. Leuschel, M. Samia, J. Bendisposto, and L. Luo. Easy Graphical Animation and Formula Viewing for Teaching B. *The B Method: from Research to Teaching*, pages 17–32, 2008.

[31] G. Mariano. *Évaluation de Logiciels Critiques Développés par la Méthode B: Une Approche Quantitive*. PhD thesis, Université de Valenciennes et Du Hainaut-Cambrésis, December 1997.

[32] C. Métayer. *AnimB 0.1.1*, 2010. Available at `http://wiki.event-b.org/index.php/AnimB`.

[33] C. Métayer and L. Voisin. The Event-B mathematical language. Available at `http://wiki.event-b.org/index.php/Event-B_Mathematical_Language`, 2009.

[34] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[35] F. Redmill. Safety integrity levels ? theory and problems. In *Lessons in System Safety: Proceedings of the Eighth Safety-critical Systems Symposium*, Southampton, UK, 2000. Available at http://www.csr.ncl.ac.uk/FELIX_Web/new_index.html.

[36] T. Servat. Brama: A new graphic animation tool for B models. In J. Julliand and O. Kouchnarenko, editors, *Proceedings B 2007*, LNCS 4355, pages 274–276. Springer, 2006.

[37] Siemens. B method - optimum safety guaranteed. *Imagine*, (10):12–13, June 2009.

[38] F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 2009. Available at `http://www.atelierb.eu/`.

[39] B. Tatibouet. The jbtools package. Available at `http://lifc.univ-fcomte.fr/PEOPLE/tatibouet/JBTOOLS/BParser\_en.html`, 2001.

[40] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, LNCS 4424, pages 632–647. Springer, 2007.

[41] E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.

[42] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[43] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA$^+$ specifications. In L. Pierre and T. Kropf, editors, *CHARME*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.