Automated Property Verification for Large Scale B Models*

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

[1] Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{ leuschel, plagge } @cs.uni-duesseldorf.de
[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 on an industrial case study. The case study centres 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 AtelierB. AtelierB, 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 around 17 minutes that were manually uncovered in about one man-month. 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. Notably, a new parser and type checker had to be developed. We also touch upon 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.

1 Background Industrial Application

Siemens Transportation Systems have been developing rail automation products using the B-method since 1998.[1] The best known example is obviously the soft-

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

[1] At that time Siemens Transportation Systems was named MTI (Matra Transport International).

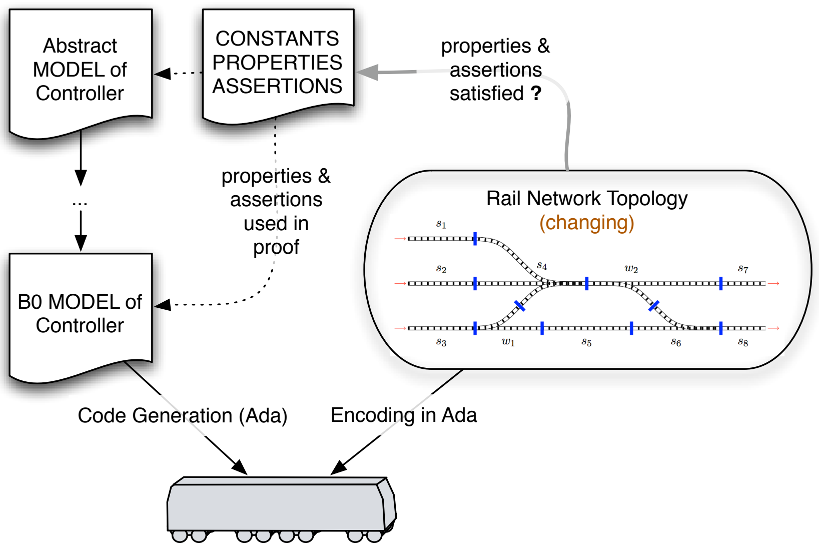


Fig. 1. Overview of the Constants Validity Problem

ware for the fully automatic driverless Line 14 of the Paris Métro, also called Météor (**Met**ro **e**st-**o**uest **r**apide) [4]. But since then, many other train control systems have been developed and installed worldwide by STS [7, 3, 9]. One particular development is in San Juan (Puerto Rico), which we will use as 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).

STS are successfully using the B-method and have over the years acquired considerable expertise in its application. STS use Atelier B [19], together with in-house developed automatic refinement tools. In this paper, we describe one aspect of the current development process which is far from optimal, namely the validation of properties of parameters only known at deployment time. The parameters are typically constants in the 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 check during execution. The track is divided into several sub-sections, each sub-section is controlled by a safety critical software. In order to avoid multiple developments, each software is made from a generic B-model and parameters that are specific to a sub-section. The proofs of the generic B-model rely on assumptions that formally describe the topology properties of the track. We therefore have to make sure that the parameters used for each sub-section actually verify the formal assumptions.

For example, in case of the San Juan development, about 300 assumptions were made.[2] 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).

For this, Siemens Transportation Systems (STS) have developed the following approach:

1. The topology is extracted from the ADA program and encoded in B syntax, written into AtelierB definition files.
2. The relevant part of the B model is extracted and conjoined with the definition files containing the topology.
3. The properties and assertions[3] are proven with Atelier B, using custom proof rules and tactics.

There are two problems with this approach.

– 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).
– The constants are very large (relations with thousands of tuples) and the properties so complex (see Figure 2) that Atelier B quite often runs out of memory. For example, for the San Juan development, 80 properties (out of the 300) could not be checked by Atelier B.

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). For the San Juan development, this meant about one man month of effort, which is likely to grow much further for larger developments such as [9].

The starting point of this paper was to try to automate this task, by using an alternative technology. Indeed, the ProB tool [13, 15] has to be capable of dealing with B properties in order to animate and model check B models. The big 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 datastructures that were required to deal with the large sets and relations of the case study. The results of the case study itself are presented in Section 4, while in Section 5 we present how we plan to validate ProB for integration into the development cycle at Siemens. Finally, in Section 6 we present more related work, discussions and an outlook.

[2] Our model contains 226 properties and 147 assertions; some of the properties, however, are extracted from the ADA code and determine the network topology and other parameters.

[3] In B assertions are predicates which should follow from the properties.

```
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. 2. A small selection of the assumptions about the constants of the San Juan topology

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.

Improved Parser Initially, ProB [13, 15] was built using the jbtools [20] parser. This parser was initially very useful to develop a tool that could handle a large subset of B. However, this 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 unsupported features) near impossible. Unfortunately, jbtools was also difficult to maintain and extend.[4] This was mainly due to the fact that the grammar had to be made suitable for top-down predictive parsing using JavaCC, and that it

[4] We managed to somewhat extend the capabilities of jbtools concerning definitions with parameters, but we were not able to fully support them.

used several pre- and post-passes to implement certain difficult features of B (such as the relational composition operator ';', which is also used for sequential composition of substitutions), which also prevented the generation of a clean abstract syntax tree.

Thus, the first step towards making ProB suitable for industrial usage, was the development of a new parser. This parser was built with extensibility in mind, and now supports almost all of the Atelier B syntax. We used SableCC 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 suitable for predictive top-down parsing) and to provide fully typed abstract syntax tree.

There are still a few minor differences with Atelier B syntax (which only required minimal changes to the model, basically adding a few parentheses). In fact, in some cases our parser is actually more powerful 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).

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 algorithm for ProB, also making use of the source code locations provided by the new parser to precisely pinpoint type errors. The type inference algorithm is based upon Prolog unification, and as such is more powerful than Atelier B's type checker,[5] and we also type check the definitions. 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 provides performance benefits to ProB, e.g., by disambiguating between Cartesian product and multiplication for example.

The scale of the specifications from STS also required a series of other efficiency improvements within ProB. For example, the abstract syntax tree of the main model takes 16.7 MB in Prolog form, which was highlighting several performance issues which did not arise in smaller models.

All in all, about eight man-months of effort went into these 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 opened up. Indeed, only in very rare circumstances can one expect industrialists to adapt their models to suit an academic tool.

[5] It is even more powerful than the Rodin [1] type checker, often providing better error messages.

In the next section we address the further issue of effectively dealing with the large data values manipulated upon by these specifications.

3 Checking Complicated Properties

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) contain 2324 tuples. Larger relations still can arise when evaluating the properties (e.g., by computing set union or set comprehensions).

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). E.g., after processing the predicates $card(s) = 2$ 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 datastructure 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.

To get an idea of the performance, take a look at the following operation, coming from a B formalisation of the Sieve of Eratosthenes, where `numbers` was initialised to `2..limit` and where `cur=2`:

```
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 on a MacBook Pro with 2.33 GHz Core2 Duo processor and 3 GB of RAM. With the new datastructure this operation, involving the computation of a lambda expression, the range of it and a set difference, is now almost instantaneous (0.2 seconds). For limit = 100,000 it requires 2.1 seconds, for limit = 1,000,000 ProB requires about 21.9 seconds, and for limit = 10,000,000 ProB requires about 226.8 seconds. Figure 3 contains an log-log plot of the runtime for various values of `limit`, and clearly shows that ProB's operations scale quasi linearly with the size of the sets operated upon (as the slope of the runtime curve is one).

There is one caveat, however: this datastructure can (for the moment) only be used 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 which value `X` will take on. Hence, for partially known values, the old-style list representation still has to be used. For efficiency, it is thus important to try to work with fully determined

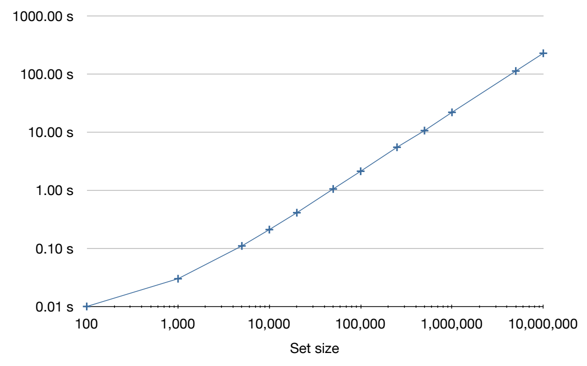


Fig. 3. Performance of the new ProB datastructure and operations on large sets

values as much as possible. For this we have improved the constraint propagation mechanism inside the ProB kernel. The previous version of ProB [15] 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 afterward 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.

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.

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 datastructure to represent and manipulate large sets and relations. A new, more fine grained constraint propagation algorithm was also required to ensure that this datastructure could actually be used in the industrial application.

4 The 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 and a hard model. 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 datastructure 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 errors were genuine and they were *exactly* the same errors that Siemens had uncovered themselves by manual inspection.

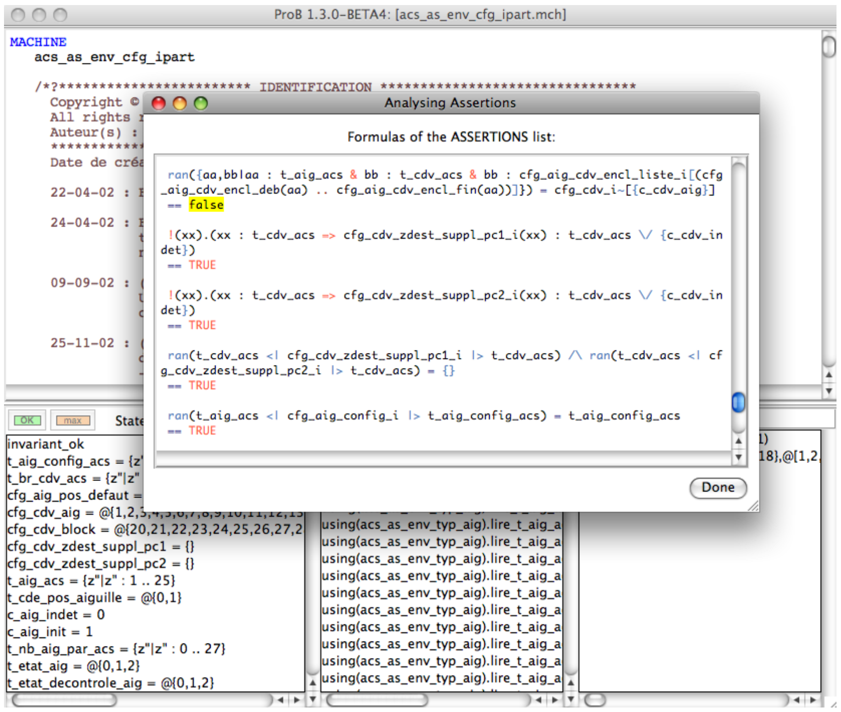
The manual inspection of the properties took Siemens several weeks (about a man month of effort). Checking the properties takes 4.15 seconds, and checking the assertions takes 1017.7 seconds (i.e., roughly 17 minutes) using ProB 1.3.0-final.4 on a MacBook Pro with 2.33 GHz Core2 Duo (see also Figure 4).

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 5.

The four false formulas found by ProB are the following ones:

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.


ProB 1.3.0-BETA4: [acs_as_env_cfg_ipart.mch]
Analysing Assertions
Formulas of the ASSERTIONS list:
Done
OK
max
State

```
MACHINE
   acs_as_env_cfg_ipart

   /*?************************ IDENTIFICATION ********************************
     Copyright ©
     All rights r
     Auteur(s) :
     ***********
     Date de crée

     22-04-02 : E

     24-04-02 : E

     09-09-02 : (

     25-11-02 : (
```

```
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}]
 == false

 !(xx).(xx : t_cdv_acs => cfg_cdv_zdest_suppl_pc1_i(xx) : t_cdv_acs \/ {c_cdv_in
det})
 == TRUE

 !(xx).(xx : t_cdv_acs => cfg_cdv_zdest_suppl_pc2_i(xx) : t_cdv_acs \/ {c_cdv_in
det})
 == TRUE

 ran(t_cdv_acs <| cfg_cdv_zdest_suppl_pc1_i |> t_cdv_acs) /\ ran(t_cdv_acs <| cf
g_cdv_zdest_suppl_pc2_i |> t_cdv_acs) = {}
 == TRUE

 ran(t_aig_acs <| cfg_aig_config_i |> t_aig_config_acs) = t_aig_config_acs
 == TRUE
```

```
invariant_ok
t_aig_config_acs = {z'
t_br_cdv_acs = {z"|z"
cfg_aig_pos_defaut =
cfg_cdv_aig = @{1,2,3,4,5,6,7,8,9,10,11,12,13
cfg_cdv_block = @{20,21,22,23,24,25,26,27,2
cfg_cdv_zdest_suppl_pc1 = {}
cfg_cdv_zdest_suppl_pc2 = {}
t_aig_acs = {z"|z" : 1 .. 25}
t_cde_pos_aiguille = @{0,1}
c_aig_indet = 0
c_aig_init = 1
t_nb_aig_par_acs = {z"|z" : 0 .. 27}
t_etat_aig = @{0,1,2}
t_etat_decontrole_aig = @{0,1,2}
```

```
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
using(acs_as_env_typ_aig).lire_t_aig_a
```

```
1)
18},@[1,2,
```

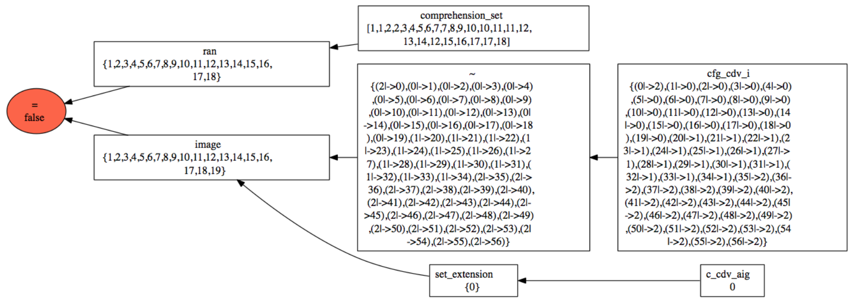
Fig. 4. Analysing the Assertions

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 compute values of B expressions and the truth-values of B predicates, as well as all sub-expressions and sub-predicates. The whole is assembled into a graphical tree representation.

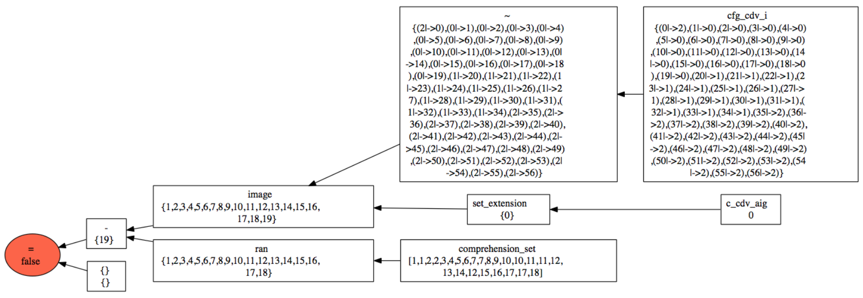
A graphical visualisation of the fourth false formula is shown in Figure 5. For each expression, we have two lines of text: the first indicates the type of the node, i.e., the top-level operator. The second line gives the value of evaluating the expression. For predicates, the situation is similar, except that there is a third line with the formula itself and that the nodes are coloured: true predicates are green and false predicates are red. (An earlier version of the graphical viewer is described in [17].)

Note that the user can type custom predicates to further inspect the state of the 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 6, where we can see that the number 19 is an element of `cfg_cdv_i [c_cdv_aig]` but not of the range expression.



=
false
ran {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}
image {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19}
comprehension_set [1,1,2,2,3,4,5,6,7,7,8,9,10,10,11,11,12,13,14,12,15,16,17,17,18]
~ {(2|->0),(0|->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),(1|->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),(2|->35),(2|->36),(2|->37),(2|->38),(2|->39),(2|->40),(2|->41),(2|->42),(2|->43),(2|->44),(2|->45),(2|->46),(2|->47),(2|->48),(2|->49),(2|->50),(2|->51),(2|->52),(2|->53),(2|->54),(2|->55),(2|->56)}
cfg_cdv_i {(0|->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),(35|->2),(36|->2),(37|->2),(38|->2),(39|->2),(40|->2),(41|->2),(42|->2),(43|->2),(44|->2),(45|->2),(46|->2),(47|->2),(48|->2),(49|->2),(50|->2),(51|->2),(52|->2),(53|->2),(54|->2),(55|->2),(56|->2)}
set_extension {0}
c_cdv_aig 0

Fig. 5. Analysing the fourth false assertion



=
false
- {19}
{} {}
image {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19}
ran {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}
~ {(2|->0),(0|->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),(1|->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),(2|->35),(2|->36),(2|->37),(2|->38),(2|->39),(2|->40),(2|->41),(2|->42),(2|->43),(2|->44),(2|->45),(2|->46),(2|->47),(2|->48),(2|->49),(2|->50),(2|->51),(2|->52),(2|->53),(2|->54),(2|->55),(2|->56)}
cfg_cdv_i {(0|->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),(35|->2),(36|->2),(37|->2),(38|->2),(39|->2),(40|->2),(41|->2),(42|->2),(43|->2),(44|->2),(45|->2),(46|->2),(47|->2),(48|->2),(49|->2),(50|->2),(51|->2),(52|->2),(53|->2),(54|->2),(55|->2),(56|->2)}
set_extension {0}
c_cdv_aig 0
comprehension_set [1,1,2,2,3,4,5,6,7,7,8,9,10,10,11,11,12,13,14,12,15,16,17,17,18]

Fig. 6. Analysing a variation of the fourth false assertion

In summary, the outcome of this case study was extremely positive: a man-month of effort has been replaced by 17 minutes 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 the next section.

5 Validation of ProB

In this case study, ProB was compared with Atelier B. For this specific use, the performances of ProB are far better than the performances of Atelier B, but ProB is not yet qualified for use within a SIL 4 (the highest safety integrity level) development life cycle. If ProB evaluates an assumption to be true, Siemens would like to be able to rely on this result and not have to investigate the correctness of this assumption manually.

There are two ways this issue can be solved:

– Use a second, independently developed tool to validate the assumptions. One possibility would be Atelier B, but as we have already seen it is currently not capable to deal with the more complicated assumptions. Another possibility would be to use another animator, such as Brama [18] or AnimB. 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 both of these tools agreed upon the assumptions. This avenue is being investigated, but note that Brama and AnimB are much less developed as far as the constraint solving capabilities are concerned.[6] Hence, it is still unclear whether they can be used for the complicated properties under consideration.
– Validate ProB, at least those parts of ProB that have been used for checking the assumptions. We are undertaking this avenue, and provide some details in the remainder of this section.

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).[7]

1. Unit Tests:
   ProB contains over a 1,000 unit tests at the Prolog level. For instance, these check the proper functioning of the various core predicates operating on B's datastructures. E.g., it is checked that $\{1\} \cup \{2\}$ evaluates to $\{1, 2\}$.
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. This overcomes to some extent the fact that Prolog has no static typing.
3. Integration and Regression Tests:
   ProB contains over 180 regression tests which are made up of B models along with saved animation traces. These models are loaded, the saved animation trace replayed and the model is 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 counter example 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

[6] Both Brama and AnimB require all constants to be fully valued, AnimB for the moment is not capable of enumerating functions, etc.

[7] 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 [1] plugin.

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 7 contains some of these laws about functions.

   These tests have 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.

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. With these two predicates we can uncover undefined predicates:[8] 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 8 (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.

In order to complete the validation of ProB we are planning to do the following steps:

1. Validation of the parser (via pretty-printing and re-parsing and ensuring that a fixpoint is reached).
2. Validation of the type checker.
3. The development of a formal specification of core parts of ProB and its functionality.
4. An analysis of the statement coverage of the Prolog code via the above unit, integration and regression tests. In case the coverage is inadequate, the introduction of more tests to ensure satisfactory coverage at the Prolog level.
5. The development of a validation report, with description of ProB's functions, and a classification of functions into critical and non-critical, and a detailed description of the various techniques used to ensure proper functioning of ProB.

[8] Another 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.

```
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. 7. A small selection of the laws about B functions

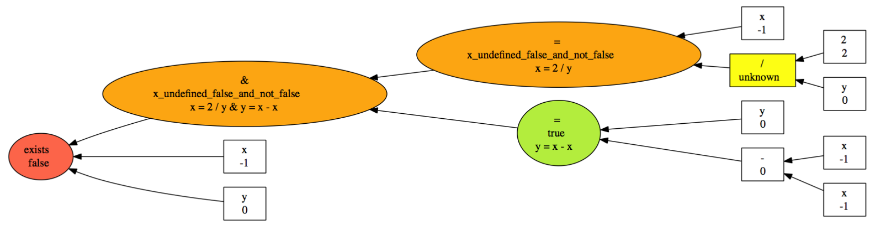


Fig. 8. Visualising an undefined property

6 More Related Work, Conclusion and Outlook

More Related Work In addition to the already discussed approach using Atelier B and proof, and the animators Brama and AnimB we would like to mention the BZ-TT [12], a test generation tool for B and Z specifications. A specification is translated into constraints and the CLPS-B constraint solver [5] is used to find boundary values and to determine test sequences [2]. BZ-TT is now part of a commercial product named LEIRIOS Test Generator. The 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, ...).

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 [23], 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 currently investigating using Kodkod [21] as an alternative engine to solve or evaluate complicated constraints. Kodkod provides a high-level interface to SAT-solvers, and is also at the heart of Alloy [10]. Indeed, for certain complicated constraints over first-order relations, Alloy can be much more efficient than ProB. However, it seems unlikely that Kodkod will be able to effectively deal with relations containing thousands or tens of thousands of elements, as it was not designed for this kind of task. Indeed, Alloy is based upon the "small scope hypothesis" [11], which evidently is not appropriate for the particular industrial application of formal methods in this paper. In our experience, Alloy and Kodkod do not seem to scale linearly with the size of sets and relations. For example, we reprogrammed the test from Figure 3 using Kodkod, and it is about two orders of magnitude slower than ProB for 1,000 elements and three orders of magnitude for 10,000 elements (363.2 s versus 0.21 s; see also the log-log plot in Figure 9 which indicates an exponential growth as the slope of the Kodkod curve is > 1). In addition, the higher-order aspects of B would in all likelihood still have to be solved by ProB (Alloy and Kodkod only support first-order relations).

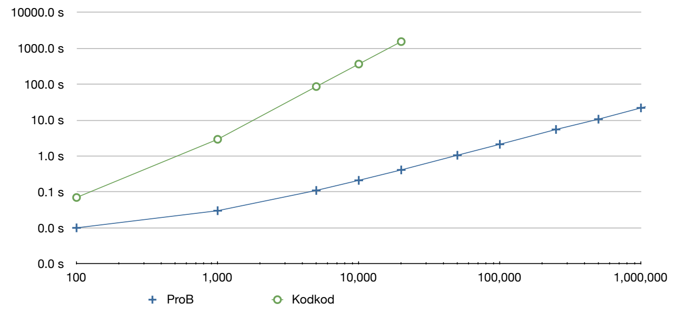


Fig. 9. Performance of the new ProB vs Kodkod on large sets

We are also investigating whether the SMT solver Yices [8] 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 datastructures the translation to SAL breaks down (see also the translation from Z to SAL in [6] and the discussion about the performance compared to ProB in [16]). 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 datastructure 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 17 minutes of computation. 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 use it within the SIL4 development cycle at STS. 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 [9], up from 100,000 lines for Météor [4]). As far as runtime is concerned, there is still a lot of leeway. In the San Juan case study 17 minutes were required on a two year old laptop to check all properties and assertions. The individual formulas could actually be easily split up amongst several computers and even several hours of runtime would still be acceptable for Siemens. As far as memory consumption is concerned, for one universally quantified property we were running very close to the available memory (3 GB). Luckily, we can compile ProB for a 64 bit system and we are also investigating the use ProB's symmetry reduction techniques [14, 22] inside quantified formulas.

References

1. 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.
2. 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.
3. F. Badeau and A. Amelot. Using b as a high level programming language in an industrial project: Roissy val. In *ZB*, pages 334–354, 2005.
4. 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.
5. 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.
6. 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.

7. 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.
8. 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.
9. D. Essamé and D. Dollé. B in large-scale projects: The Canarsie line CBTC experience. In *Proceedings B'07*, pages 252–254, 2007.
10. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.
11. D. Jackson. *Software Abstractions: Logic, Language and Analysis.* MIT Press, 2006.
12. 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.
13. 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.
14. 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.
15. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
16. 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.
17. 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.
18. 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.
19. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com`.
20. B. Tatibouet. The jbtools package. available at http://lifc.univ-fcomte.fr/PEOPLE/tatibouet/JBTOOLS/BParser_en.html, 2001.
21. 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.
22. 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.
23. 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.