Chapter 1
Improving Railway Data Validation with ProB

Jérôme Falampim, Hung Le-Dang, Michael Leuschel, Mikael Mokrani, Daniel Plagge

Abstract In this chapter, we describe the successful application of ProB on industrial projects realized by Siemens. Siemens are successfully using the B-method to develop software components for zone controller and carbonne controller of CBTC systems. 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 properties related to large constants (relations with thousands of tuples). 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 chapter we show how we were able to use ProB to overcome this challenge. We describe the deployment and current use of ProB in the SIL4 development chain at Siemens. 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.

1.1 Introduction

Siemens have been developing Communication-based Train Control (CBTC) products using the B-method [1] since 1998 and have over the years acquired considerable expertise in its applications. Siemens use Atelier B [2], together with in-house developed automatic refinement tools, to develop critical control software components in CBTC systems with great success. 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 highest Safety Integrity

Level (SIL4). This railway software development process comlies fully with current railway standards (EN 50126, EN50128, EN50129) and reduces significantly the test cost at the validation step. Indeed, the unit test is no longer needed in this process as it was replaced by proof activities during the development.

The first and best known example is obviously the controller software component for the fully automatic driverless Line 14 of the Paris Métro, also called Météor (Metre est-ouest rapide). In-deed, quoting [3]: ”Since the commissioning of line 14 in Paris on 1998, not a single malfunction has been noted in the software developed using this principle”. Since then, many other train control systems have been developped and installed worldwide by Siemens [4, 5] : San Juan metro (Puerto Rico, commisionning on 2003), New York metro Canarsie line (USA, commisionning on 2009), Paris metro line 1 (France, Commisionning on 2011), Barcelona metro line 9 (Spain, Commisionning on 2009), Alger metro line 1 (Algeria, Commisionning on 2011), Sao Paolo metro line 4 (Brasil, commisionning on 2010), Charles de Gaulle Airport Shuttle line 1 (France, commisionning on 2005), e.t.c.

Relationship to Deploy While the work described in this chapter was carried out within the context of Deploy, Siemens use classical B and not Event-B in their current development process. Indeed, neither Event-B nor the Rodin tool is at the time of writing not ready for software development at an industrial level. However, Event-B and classical B obviously share a common basis, and the ProB tool used in this chapter works equally well for both formalisms.

The Property Verification Problem. One aspect of the current development process which is unfortunately still problematic is the validation of properties of parameters only known at deployment time, such as the rail network topology parameters. In CBTC systems, the track is divided into several sub-sections, each of which is controlled by safety critical software called ZC (Zone Controller). A Zone Controller contribues to realize ATP (Automatic Train Protection) and ATO (Automatic Train Operations) functions of CBTC systems for a portion of the track : train location, train anticollision, over-speed prevention, train movement regulation, train and platform door management, e.t.c. In order to avoid multiple developments, each ZC is made from a generic B-model and data parameters that are specific to a sub-section and a particular deployment (cf. Fig. 1.1).

These data parameters take the form of B functions describing, e.g., the tracks, switches, traffic lights, electrical connections and possible routes. These B functions are typically regrouped in basic invariant machines in which the assumptions about the topology properties are defined in the PROPERTIES clause. 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.

A trivial example is the slope of the track: the calculation of the stopping distance is made with an algorithm that requires the slope to be within +/-5%. In the generic software model, we do the assumption of a slope being between -5% and +5%, and it enables to complete the proof of the algorithm. Then, we also have to make sure that the assumption is correct, that is to say the slope is indeed between -5% and

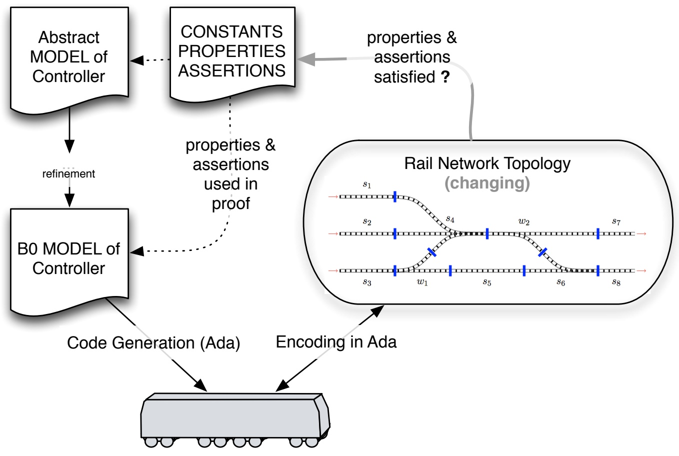

Abstract MODEL of Controller
CONSTANTS PROPERTIES ASSERTIONS
properties & assertions satisfied ?
refinement
properties & assertions used in proof
Rail Network Topology
(changing)
B0 MODEL of Controller
Code Generation (Ada)
Encoding in Ada

Fig. 1.1 A need for ZC Data Validation

+5% in the track data (and, of course, that the track data is correct regarding the actual track). More generally, data validation is needed when a generic software is used with several sets of specific data.

1.2 An approach to validate ZC data using Atelier B.

Siemens have developed the following approach for validating ZC data :

1. The parameters and topology are extracted from the concrete Ada program (each of which corresponds to an invariant basic machine in the B-model. In B, a basic machine does not have a refinement nor an implementation but it has instead an implementation in ADA) and encoded in B0 (i.e B executable) 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 definition files containing the topology and the other parameters are merged with the relevant invariant basic machines to create assertion machines. The properties from a basic machines on the concrete topology and parameters are translated into B assertions the in relevant assertion machine. The macros derived from actual data extracted from ADA programs are used to define the constants value and this definition is realized as a properties in the assertion machines.

In B assertions are predicates which should follow from the properties or invariant, and have to be proved. Properties themselves do not have to be proved, 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. The machines with assertions obtained from step 2 are then grouped in a B project called IVP (Invariant Validation Project). Siemens tries to prove the assertions with Atelier B, using custom proof rules and tactics, dedicated to dealing with explicit data values.
4. Those assertions for which proof is unsuccessful are investigated manually.

Problems with the Existing Process. This approach initially worked quite well for ZC data, 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, 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. 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 Canarsie line [5].

1.3 First experiments with ProB

The starting point of the experiment was to try to automate the proof of the remaining proof obligations, by using an alternate technology. Indeed, the ProB tool [9, 11] has capabilities to deal 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 order to evaluate the feasibility of using ProB for checking the topology properties, Siemens sent the STUPS team at the University of Dusseldorf 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 analyzed: a simpler model and a hard model.

It then took STUPS a while to understand the models and get them through the new parser, whose development was being finalized at that time.

On 14th November 2008 STUPS were able to animate and analyze 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 in ProB to validate bigger models.

On the 8th of December 2008 STUPS were finally able to animate and validate the complicated model. This revealed four errors. Note that the STUPS team were not told about the presence of errors in the models (they were not even hinted at by Siemens), and initially STUPS believed that there was still a bug in ProB. In fact, 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 on a MacBook Pro with 2.33 GHz Core2 Duo. More information on this subject was presented at FM 2009 [7] and in the ensuing journal paper [8].

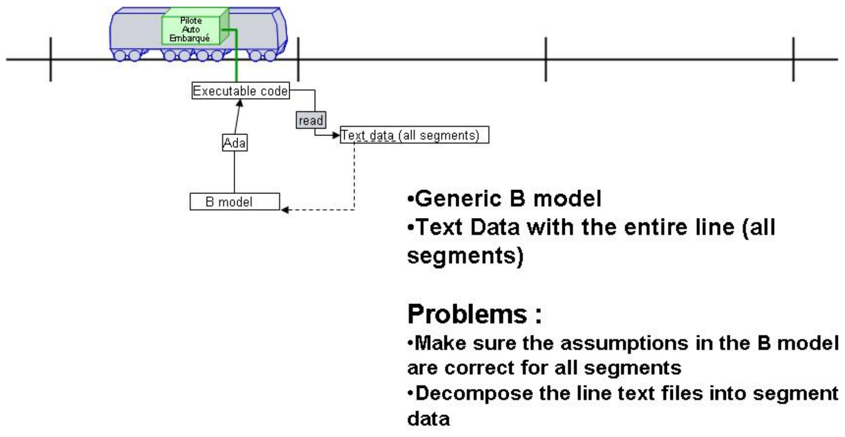
1.4 RDV : Railway Data Validator

In the first experiments, ProB was used instead of Atelier B, on the same IVP, with great success. But the creation of IVP was still problematic, with few atomization. As described in Sect. 1.1, each IVP is an encoding of a specific wayside configuration data in B; this is required in order to validate the configuration data against the formal properties in the generic B project.

The goal was to create a tool that could automatically generate the B projects (containing assertions machines), run ProB on created B projects, and collect the result in a synthesis report. In addition, this tool should not work only on ZC data, but also on CC (Carbonne Controller) data which were not formally validated before the use of ProB. Indeed, in the CC software development, as shown in Fig. 1.2, the topology data are contained in textfiles and loaded "on the fly" by the CC software component when need. Therefore, in order to enable the CC data validation, the macros in definition files are derived from topology text files instead of ADA programs. Such macros are then merged with variables defined in basic invariant machines to assertion machines for the segment in question.

RDV is a new tool realized by Siemens. Via a graphical interface (cf. Fig. 1.3), RDV provides following services :

IVP Generation: this function generates an IVP for a sub-section in case of ZC or a segment in case of CC. The generated IVP is almost ready to be used by ProB or Atelier B. Indeed, we still need some manual modifications related to properties of non-function constants, however, in comparison with the former tool, it reduces significantly manual modifications on generated B machines. In

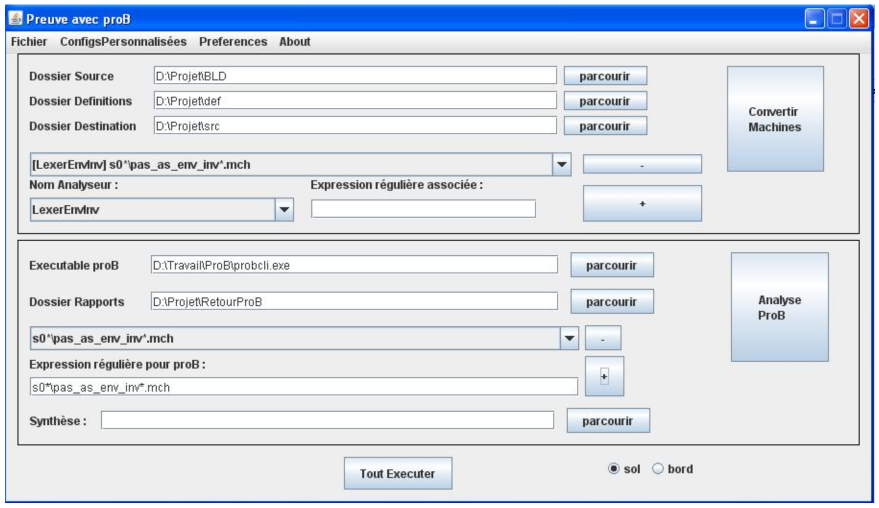


•Generic B model
•Text Data with the entire line (all segments)

Problems :

•Make sure the assumptions in the B model are correct for all segments
•Decompose the line text files into segment data

Fig. 1.2 CC Data Validation



Dossier Source: D:\Projet\BLD
Dossier Definitions: D:\Projet\def
Dossier Destination: D:\Projet\src
[LexerEnvInv] s0*\pas_as_env_inv*.mch
Nom Analyseur : LexerEnvInv
Expression régulière associée :
Executable proB: D:\Travail\ProB\probcli.exe
Dossier Rapports: D:\Projet\RetourProB
s0*\pas_as_env_inv*.mch
Expression régulière pour proB : s0*\pas_as_env_inv*.mch
Synthèse :
[✓ sol] bord

Fig. 1.3 RDV interface

addition, with the file selection function based on regular expression, RDV enables the generation of a subset of assertions machines (i.e, only machines with modifications). Moreover, the automation of CC IVP creation is a great help for safety engineers as there are about several hundreds IVP to be crated (one project per segment).

ProB Launch: RDV enables users to parameterize ProB before launching it. ProB is is called on each assertion machine in order to analyze the assertions contained in each machine. It does not require B experts to carry out data validation. Indeed, the B experts are required only in case of problem, whereas in the former process, B experts were required in any case, for long and fastidious tasks.

In addition, using ProB significantly reduces the time checking data properties: from 2 or 3 days with old tool to 2 or 3 hours per project with RDV. The analyse of CC assertion machines with ProB reduces signgificantly the interaction with user as one does not need launch Atelier B several hundreds times on created CC IVP.

Assertion-Proof Graph Generation: This function provides a graphical way to investigate the proof on an assertion. Is is based on a service provided by ProB to compute values of B expressions and the truth-values of B predicates, as well as all sub-expressions and sub-predicates. This is very useful because users often want to know more about the exact source why an assertion fails. 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.

Validation Synthesis Report: The results of analysis realized by ProB on assertions machine are recorded in a set of .rp and .err files (one per B component and per sector/segment). Each rp (report) file contains the normal results of the analysis with ProB :

- Values used during initialization of machines;
- Proof details on each assertion checked;
- Result of the analysis (true, false, timeout...).

Each err (error) file contains the abnormal results of the analysis:

- Variables/Constants with incorrect type;
- Variables/Constants with multiple values, redefinition of value or missing value;
- Error during execution of ProB (missing file...).

When all report and error files have been generated, an html synthesis report is issued. This report gives the result (number of false/true assertions, timeout, unknown results...) for each sector/segment. For each B component, a hyperlink to the detailed results, error and report files gives access to the results of the component (in order to know which assertion is false, for instance).

1.5 ZC data validation

As the CC data validation only differs from the ZC data validation in the way the assertion machines are created, we present here an example of the ZC data validation to show how data validation is carried out with RDV.

As shown in fig. 1.1, the track is decomposed into several sectors (from 2 to 10), there is one zone controller dedicated for each sector. The Ada data are linked with the Ada code trans-coded from the generic B model. Below is a B machine with some properties on the data inv_zaum_quai_i and inv_zaum_troncon_i:

```
MACHINE
 pas_as_env_inv_zaum
SEES
 pas_as_env_typ_quai,
 pas_as_env_typ_zaum,
 pas_as_env_typ_troncon
CONCRETE_CONSTANTS
 inv_zaum_quai_i,
 inv_zaum_troncon_i
DEFINITIONS
 typage_tab == (
   inv_zaum_quai_i : t_nb_zaum_par_pas --> t_nb_quai_par_pas &
   inv_zaum_troncon_i : t_nb_zaum_par_pas --> t_nb_troncon_par_pas )
PROPERTIES
 typage_tab &
   /* Proprietes */
  !xx.(xx : t_zaum_pas
         =>
         inv_zaum_quai_i(xx) : t_quai_pas \/ {c_quai_indet}) &
  !xx.(xx : t_zaum_pas
         =>
         inv_zaum_troncon_i(xx) : t_troncon_pas \/ {c_troncon_indet})
END
```

An example of the relevant ADA program with configuration data is as followed:

```
with SEC_OPEL_FONC;
with PAS_AS_ENV_TYP_FONC;

use SEC_OPEL_FONC;

package pas_as_env_inv_zaum_val is
    subtype T_INV_ZAUM_TRONCON_I
       is SEC_OPEL_FONC.FINT_ARRAY_1(PAS_AS_ENV_TYP_FONC.T_NB_ZAUM_PAR_PAS);
    subtype T_INV_ZAUM_QUAI_I
       is SEC_OPEL_FONC.FINT_ARRAY_1(PAS_AS_ENV_TYP_FONC.T_NB_ZAUM_PAR_PAS);
  INV_ZAUM_TRONCON_I : constant T_INV_ZAUM_TRONCON_I := T_INV_ZAUM_TRONCON_I'(
      1 => 3,      2 => 3,      3 => 3,      4 => 3,
      5 => 4,      6 => 4,      7 => 3,      8 => 3,
      9 => 3,      10 => 3,      11 => 4,      12 => 4,
      13 => 4,      14 => 4,      15 => 4,      16 => 4,
      17 => 3,      18 => 3,      19 => 3,      21 => 3,
      22 => 1,      23 => 1,      24 => 1,      25 => 1,
      26 => 1,      27 => 2,      28 => 2,      29 => 2,
      30 => 1,      31 => 2,      32 => 2,      33 => 2,
      34 => 2,      35 => 2,      36 => 3,
      OTHERS=>0
    );
  INV_ZAUM_QUAI_I : constant T_INV_ZAUM_QUAI_I := T_INV_ZAUM_QUAI_I'(
      23 => 1,      25 => 2,
      27 => 3,      31 => 4,
      33 => 5,
      OTHERS=>0
    );
end pas_as_env_inv_zaum_val;
```

Using RDV, we obtained the following assertion machine:

```
MACHINE
 pas_as_env_inv_zaum
SEES
 pas_as_env_typ_quai,
 pas_as_env_typ_zaum,
 pas_as_env_typ_troncon
CONCRETE_CONSTANTS
 inv_zaum_quai_i,
 inv_zaum_troncon_i
DEFINITIONS
 "pas_as_env_inv_zaum_val.1.def";
 typage_tab == (
    inv_zaum_quai_i    : t_nb_zaum_par_pas --> t_nb_quai_par_pas  &
    inv_zaum_troncon_i : t_nb_zaum_par_pas --> t_nb_troncon_par_pas )
PROPERTIES
 inv_zaum_quai_i = inv_zaum_quai_i_indetermine <+ inv_zaum_quai_i_surcharge &
 inv_zaum_troncon_i = inv_zaum_troncon_i_indetermine <+ inv_zaum_troncon_i_surcharge
ASSERTIONS
 typage_tab ;
 !xx.(xx : t_zaum_pas => inv_zaum_quai_i(xx) : t_quai_pas \/ {c_quai_indet}) ;
 !xx.(xx : t_zaum_pas => inv_zaum_troncon_i(xx) : t_troncon_pas \/ {c_troncon_indet})
END
```

The machine name is unchanged for traceability reasons. The ASSERTIONS clause in the new machine contains predicates which were in the PROPERTIES clause of the orignal machine. The DEFINITIONS clause includes the definition file pas_as_env_inv_zaum_val.1.def which contains macros derived from data defined in pas_as_env_inv_zaum_val.1.ada:

- inv_zaum_quai_i_indetermine and inv_zaum_quai_i_surchage which were derived from the configuration data INV_ZAUM_QUAI_I.
- inv_zaum_troncon_i_indetermine and inv_zaum_troncon_i_surchage which were derived from the configuration data INV_ZAUM_QUAI_I.

```
DEFINITIONS
        inv_zaum_quai_i_indetermine == (t_nb_zaum_par_pas)*{0};
        inv_zaum_quai_i_surcharge =={
                23|->1, 25|->2, 27|->3, 31|->4, 33|->5
        };

        inv_zaum_troncon_i_indetermine == (t_nb_zaum_par_pas)*{0};
        inv_zaum_troncon_i_surcharge =={
                1|->3, 2|->3, 3|->3, 4|->3, 5|->4, 6|->4, 7|->3,
                8|->3, 9|->3, 10|->3, 11|->4, 12|->4, 13|->4,
                14|->4, 15|->4, 16|->4, 17|->3, 18|->3, 19|->3,
                21|->3, 22|->1, 23|->1, 24|->1, 25|->1, 26|->1,
                27|->2, 28|->2, 29|->2, 30|->1, 31|->2, 32|->2,
                33|->2, 34|->2, 35|->2, 36|->3
        }
```

The macros inv_zaum_quai_i_indetermine and inv_zaum_quai_i_surchage are then used to define inv_zaum_quai_i as shown in the PROPERTIES clause. As the modified PROPERTIES clause is instantiated using the definition file, so that inv_zaum_quai_i is replaced by the actual data:

```
 inv_zaum_quai_i = (t_nb_zaum_par_pas)*{0}
                    <+ {23|->1,25|->2,27|->3,31|->4,33|->5}
```

Similarly, the macros inv_zaum_troncon_i_indetermine and inv_zaum_troncon_i_surchage are used to define and instantiate inv_zaum_troncon_i.

The goal of the modification presented above is therefore to include the actual data (in PROPERTIES clause), include the definition files (in DEFINITION clause), and displace the assumptions on data in the ASSERTIONS clause. This modification will lead to generate some proof obligations "data" = "assumptions". For example, with inv_zaum_quai_i, we will have to prove the (simplified) following proof obligation:

```
inv_zaum_quai_i = (t_nb_zaum_par_pas)*{0} <+ {23|->1,25|->2,27|->3,
31|->4,33|->5} => inv_zaum_quai_i : t_nb_zaum_par_pas --> t_nb_quai_par_pas
```

1.6 Industrial projects already validated using RDV

In addition to San Juan case study, Currently, ProB has been used in all ongoing projects in parallel with AtelierB in data validation; the results show that ProB is more effective and less restrictive than Atelier B for railway data validation. Below are experiences from typical projects.

ALGER line 1 (ZC). This is the first driverless metro line in Africa. This line is composed of 10 stations over 10 km. The line is divided into two sections, each of which is controlled by a ZC. There were 25,000 lines of B spread over 130 files. ProB was used for the last 3 versions of this project railway data. Table 1.1 shows the results obtained with ProB on the last version :

Table 1.1 Alger line 1 (ZC)

Sector	Predicates	TRUE	FALSE	UNKNOWN	TIMEOUT
*pas_as_inv_s*01.*html*	1174	1164	10	0	0
*pas_as_inv_s*02.*html*	1174	1162	12	0	0

Each line represents the summary result for one sector. The Predicates column shows the number of assertion to be analyzed, the TRUE column represents the number of assertions verified by ProB (no counter-example found by ProB). The FALSE column represents the number assertions failed by ProB (with counter-example). The UNKNOWN column represents the number of assertions that ProB does not know how to verify. The TIMEOUT collones corresponds to assertions that ProB encoutered a time_out problemen during analysis.
For each sector, there were two assertions un-proved with Atelier B due tu their complexity. One of which is shown below. This property has rightfully been proven wrong with proB with the railway data for both sectors.

```
ran(inv_quai_variants_nord_troncon >< (((((((((((t_quai_pas <|
inv_quai_adh_red_nord_rg_variant_bf_i) |> t_rg_variant_bf) \/ ((t_quai_pas <|
inv_quai_ato_inhibe_nord_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_mto_inhibe_nord_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_atp_inhibe_nord_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_arret_tete_nord_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_arret_centre_nord_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_arret_queue_nord_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_tete_ape_nord_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_centre_ape_nord_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_queue_ape_nord_rg_variant_bf_i) |> t_rg_variant_bf))) /\
ran(inv_quai_variants_sud_troncon >< ((((((((((t_quai_pas <|
inv_quai_adh_red_sud_rg_variant_bf_i) |> t_rg_variant_bf) \/ ((t_quai_pas <|
inv_quai_ato_inhibe_sud_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_mto_inhibe_sud_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_atp_inhibe_sud_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_arret_tete_sud_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_arret_centre_sud_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_arret_queue_sud_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_tete_ape_sud_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_centre_ape_sud_rg_variant_bf_i) |> t_rg_variant_bf)) \/ ((t_quai_pas <|
inv_quai_queue_ape_sud_rg_variant_bf_i) |> t_rg_variant_bf))) = {}
```

Sao Paolo line 4 (ZC). This is the first driverless metro line in Sud-America. This line has 11 stations over 12,8km and is divided into 3 sections. The model consisted of 210 files with over 30,000 lines of B. ProB has been used for the last 6 versions of this project railway data. For the last version, the results are presented in Tab. 1.2.

Table 1.2 Sao Paolo line 4 (ZC)

Sector	Predicates	TRUE	FALSE	UNKNOWN	TIMEOUT
pas_as_inv_s036.html	1465	1459	6	0	0
pas_as_inv_s037.html	1165	1460	5	0	0
pas_as_inv_s038.html	1465	1457	8	0	0

ProB has detected issues with a group of properties which had to be put in commentary in machines used with Atelier B because they were crashing the predicates prover. Here an example of one of them:

```
!(cv_o,cv_d).(((cv_d : t_cv_pas & cv_o : t_cv_pas) & cv_o :
inv_lien_cv_cv_orig_i[inv_chainage_cv_liste_i[inv_chainage_cv_deb(cv_d) ..
inv_chainage_cv_fin(cv_d)]])
& not(inv_lien_cv_cv_dest_i((t_cv_pas <| inv_lien_cv_cv_orig_i~)|>
inv_chainage_cv_liste_i[inv_chainage_cv_deb(cv_d) ..
inv_chainage_cv_fin(cv_d)](cv_o)) = cv_d)
=> inv_lien_cv_cv_dest_i((t_cv_pas <| inv_lien_cv_cv_orig_i~) |>
inv_chainage_cv_liste_i[inv_chainage_cv_deb(cv_d) ..
inv_chainage_cv_fin(cv_d)](cv_o)) : inv_cv_pas_modifiable_i~[{TRUE}]))
```

Thankfully, after analysis, we have concluded that the problem was not critical. Nonetheless, without ProB, it would have been a lot harder to find these problems. In this case, the assertion-proof graphs were useful to understand better where the problems were coming from.

Paris line 1 (ZC). This is a project to automate the line 1 (25 stations over 16,6 km)) of the Paris Métro. The line has been gradually being upgraded to driverless train while it remains in operation. The first driverless train has been comminsionning since Nomvember 2011. In February 2012, there have been 17 of 49 driverless trains in operation in conjunction with manual trains. The line is going to be full driverless in the early 2013. The line is divided in 6 section. The model to be checked is the same as the SPL4's one. ProB has been used for the last 7 versions of railway data. For the last version, the results are presented in Tab. 1.3.

Table 1.3 Paris line 1 (ZC)

Sector	Predicates	TRUE	FALSE	UNKNOWN	TIMEOUT
*pas_as_inv_s*011*.html*	1503	1501	2	0	0
*pas_as_inv_s*012*.html*	1503	1498	5	0	0
*pas_as_inv_s*013*.html*	1503	1496	7	0	0
*pas_as_inv_s*014*.html*	1503	1499	4	0	0
*pas_as_inv_s*015*.html*	1503	1498	5	0	0
*pas_as_inv_s*016*.html*	1503	1498	5	0	0

Paris line 1 (PAL) . PAL (Pilote Automatique Ligne) is a controller line who realizes the Automatic Train Supervision (ATS) function of CBTC systems. The B models of PAL consisted of 74 files with over 10,000 lines of B. In all 2024 assertions about concrete data of the PAL needed to be checked. ProB found 12 in under 5 minutes. These problems have been examined and confirmed by manual inspection aterward at Siemens.

CDGVAL LISA (Charles de Gaulle Véhicle Automatique Léger . This is an extension of CDVVAL qui has been commisionning since 2005. This line is going to be operational in the early 2012. The LISA model consisted of 10,000 line of B over 30 files. This project has 3 sections. ProB has been used for the last 3 versions of this project railway data. For the last version, the results are presented in Tab. 1.4.

Table 1.4 Roissy LISA (ZC)

Sector	Predicates	TRUE	FALSE	UNKNOWN	TIMEOUT
*ry_pads_as_inv_pa*31*.html*	1038	1038	0	0	0
*ry_pads_as_inv_pa*32*.html*	957	957	0	0	0
ry_pads_as_inv_pagat.html	1038	1038	0	0	0

1.7 Tool issues

A crucial aspect when trying to deploy a tool like PROB successfully to an industrial project is that only minimal changes must me made to the existing models because changes to models are usually regarded as cost- and time-intensive. We cannot expect industrial users to adapt their models to an academic tool Thus it is important to work on the full language used in industry.

To make PROB an industrial usable tool, we had to make significant changes to the parser and type checker to even be able to load the models. Then several changes to PROB's core had to be made to make the large data sets in the models tractable.

The Parser. Previous versions of PROB used a freely available parser that lacked some features of Atelier B's syntax. In particular it had no support for parametrized DEFINITIONS, tuples that use commas instead of `|->` and definition files. We realised that the code base of the existing parser was very difficult to extend and to maintain and decided to completely rewrite the parser. We decided to use the parser generator SableCC [6], because it allowed us to write a clean and readable grammar. We briefly describe some of the aspects we encountered during the development of the parser and we describe where our parser's behaviour deviates from Atelier-B's behaviour.

- Atelier-B's definitions provide a macro-like functionality with parameters. I.e. that a call to a definition somewhere in the specification is syntactically replaced by the definition. A problem with this approach is that the use of such macros can lead to subtle problems. E.g. consider the definition `sm(x,y)==x+y`: The expression `2*sm(1,4)` is not equal to 10 as we might expect. Instead it is replaced by `2*1+4` which equals 6.
  We think this can easily lead to errors in the specification and decided to deviate from Atelier-B's behaviour in this aspect. Thus we treat `2*sm(1,4)` as `2*(1+4)` which equals 10 as expected.
- Another problem were an ambiguity in the grammar when dealing with definitions and overloaded operators like `;` and ||. `;` can be used to express composition of relations as well as sequential composition of substitutions. When such an operator is used in a definition like `d(x)==x;x` it is not clear which meaning is expected. It could be a substitution when calling it with `d(y:=y+1)` or in another context it could be an expression when calling it with two variables *a*, *b* as in `d(a,b)`. We resolved the problem by requiring the user to use parenthesis for expressions (like `(x;x)`). Substitutions are never put in parenthesis because they are bracket by `BEGIN` and `END`.
- The parser generator SableCC does not support source code positions. Thus we are not able to trace expressions in the abstract syntax tree back to the source code. This is e.g. important when we encounter errors in the specificatio to visualize where the problem lies. To circumvent the problem we modified the parser generator itself to support source code positions.

The resulting parser deviates in only a few aspects to Atelier B's parser. It is relatively rare that an Atelier B model needs to be rewritten to work with PROB.

The type checker. We re-implemented our type checker for B specifications because the former version often needed additional predicates to provide typing information. We implemented a type inference similar to the Hindley-Milner algorithm [10] that is more powerful than Atelier B's type checker and thus accepts all specifications that Atelier B would accept and more. A nice side-effect of the new type checker is that together with the parser's position information we can highlight an erroneous expression in the source code. This improves the user experience especially for new users.

Improved data structures and algorithms. Originally PROB represented all sets internally as lists. But with thousends of elements operations on lists performs very bad. We introduced an alternative represention based on self-balancing binary search trees (AVL trees) that provides much faster access to it's elements. In that context we examined the bottlenecks of the kernel in light of the industrial models given to us and implemented specialised versions of various operations. E.g. an expression like `dom(r)<:1..10` can be checked very efficiently when `r` is represented by an AVL tree. We can exploit the feature that the tree's elements are sorted and just check if the smallest and largest elements are in the interval.

Infinite sets. Several industrial specifications make use of definition of infinite sets like `INTEGER \ {x}`. PROB now detects certain patterns and keeps them symbolic. I.e. instead of trying to calculate all elements PROB just stores the condition that all elements fulfill and uses that condition for member checks and function applications. Examples for expressions that can be kept symbolic are

- integer sets and intervals,
- complement sets (like the example above),
- some lambda expressions and
- union and intersection of symbolic sets.

1.8 Tool validation

In order to be able to use PROB without resorting to Atelier-B, Siemens have asked UDUS to validate PROB. 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 PROBfunctions, 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.

Two versions of the validation report have already been produced, and a third version is under development. The test coverage reports are now generated completely automatically using the continuous integration platform "Jenkins".[1] This platform also runs all unit, regression, integration and other tests.

[1] See `http://en.wikipedia.org/wiki/Jenkins_(software).`

Below, we briefly describe the various tests as well as additional validation safeguards. In addition, the successful case studies in described earlier in this chapter also constitute a validation by practical experience: in all cases the PROB based approach has proven to be strictly superior to the existing approach.

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. In addition, there is an automatic unit test generator, which tests the PROB kernel predicates with many different scenarios and different set representations. For example, starting from the initial call:
`union([int(1)],[int(2)],[int(1),int(2)]),`
the test generator will derive 1358 unit tests. The latter kind of testing is particularly important for the PROB kernel which relies on co-routining: we want to check that the kernel predicates behave correctly no matter in which order (partial) information is propagated.

Integration and Regression Tests PROB contains over 500 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, etc.).

Self-Model Check with Mathematical Laws With this approach we use PROB's model checker to check itself, in particular the PROB kernel and the B interpreter. The idea is to formulate a wide variety of mathematical laws (e.g., taken from [1]) and then use the model checker to ensure that no counterexample to these laws can be found. The self-model check has been very effective at uncovering errors in the PROB kernel and interpreter. Furthermore, the self-model checking tests rely on every component of the entire PROB execution environment working perfectly; otherwise a violation to a mathematical law could be found. I.e., in addition to the PROB main code, the parser, type checker, Prolog compiler, hardware and operating system all have to work perfectly. Indeed, we have identified a bug in our parser (`FIN` was treated like `FIN1`), using the self-model check. Furthermore, we have even uncovered two bugs in the underlying SICStus Prolog compiler using self-model check.

Validation of the parser We execute our parser on a large number of our regression test machines and pretty print the internal representation. 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.

Validation of the type checker For the moment we also read in a large number of our regression test machines and pretty print the internal representation, this time with explicit typing information inserted. We now run this automatically generated file through the Atelier B parser and 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.) Also, as the pretty printer only prints the minimal number of parentheses, we also ensure to some extent that our parser is compatible with the Atelier B parser. 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.

With this testing we actually identified 26 errors in the B syntax as described AtelierB English reference manuals, upon which our pretty printer and parser was based (the French versions were correct; our parser is now based on the French reference manuals). We also detected that Atelier-B reports a lexical error ("illegal token |-") if the vertical bar (|) of a lambda abstraction is followed directly by the minus sign.

Double Evaluation As an additional safeguard during data validation, 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. For an assertion to be classified as true the positive Prolog predicate must succeed *and* the negative Prolog predicate must fail, introducing a certain amount of redundancy (admittedly with common error modes). In fact, if both the positive and negative Prolog predicates would succeed for a particular B predicate then a bug in PROB would have been uncovered. If both fail, then either the B predicate is undefined or we have again a bug in PROB. This validation aspect can detect errors in the predicate evaluation parts of PROB i.e., the treatment of the Boolean connectives $\vee$, $\wedge$, $\Rightarrow$, $\neg$, $\Leftrightarrow$, quantification $\forall$, $\exists$, and the various predicate operators such as $\in$, $\notin$, $=$, $\neq$, $<$, ... This redundancy can not detect bugs inside expressions (e.g., $+$, $-$, ...) or substitutions (but the other validation aspects mentioned above can).

1.9 Conclusions

In this paper we describe the successful application of the PROB tool for data validation in several industrial applications. This required the extension of the PROB kernel for large sets as well as an improved constraint propagation algorithm. 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.

Acknowledgements

We would like to thank Jens Bendisposto, Fabian Fritz and Sebastian Krings for assisting us in various ways, both in writing the chapter and in applying PROB on the Siemens models. Most 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). Parts of this chapter are taken from [7, 8].

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996
2. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals,* 2009. Available at hhttp://www.atelierb.edu/
3. Siemens. *B method - optimum safety guaranteed*. Imagine, 10:12-13, June 2009.
4. F. Badeau and A. Amelot. *Using B as a high level programming language in an industrial project*. In H. Treharne, S. King, M.-C. Henson and S.-A. Schneider editors, Proceedings of ZB 2005 : Formal Specification and Development in Z and B, 4th International Conference of B and Z Users. LNCS 3455, pages 334-354, Springer-Verlag, 2005.
5. D. Essamé and D. Dollé. *B in large-scale projects: The Canarsie line CBTC experience*. In J. Julliand and O. Kouchnarenko, editors, Proceedings of B 2007 : 7th Int. Conference of B Users. LNCS 4355, pages 252-254, Springer-Verlag, 2007.
6. E. Gagnon. *SableCC, an object-oriented compiler framework*. Masters thesis, McGill University, Montreal, Canada, 1998. Available at `http://www.sablecc.org`.
7. M. Leuschel, J. Falampim, 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-Verlag, 2009.
8. M. Leuschel, J. Falampim, F. Fritz, and D. Plagge. *Automated property verification for large scale B Models with ProB*. Formal Asp. Comput. 23(6): 683–709, 2011.
9. M. Leuschel and M.-J. Butler. *ProB: A model checker for B*. In K. Araki, S. Gnesi, and D. Mandrioli, editors, Proceedings FME 2003: Formal Methods, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
10. R. Milner. *A theory of type polymorphism in programming*. Journal of Computer and System Sciences, 17:348375, 1978.
11. M. Leuschel and M.-J. Butler. *ProB: an automated analysis toolset for the B method*. STTT, 10(2): 185–203, 2008.