Chapter 4
Improving Railway Data Validation with ProB

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

Abstract In this chapter, we describe the successful application of ProB in industrial projects realised by Siemens. Siemens is successfully using the B-method to develop software components for the zone and carborne controllers of CBTC systems. However, the development relies on certain assumptions about the actual rail network topology which have to be validated separately in order to ensure safe operation. For this purpose, Siemens has developed custom proof rules for Atelier B. Atelier B was, however, unable to deal with properties related to large constants (relations with thousands of tuples). These properties thus have, until now, had to be validated by hand at great expense (and revalidated whenever the rail network infrastructure changes). In this chapter we show how we have used ProB to overcome this challenge. We describe the deployment and current use of ProB in the SIL4 development chain at Siemens. To achieve this, it has been necessary to extend the ProB kernel for large sets and improve the constraint propagation phase. We also outline some of the effort and features involved in moving from a tool capable of dealing with medium-sized examples to one able to cope with actual industrial specifications. Notably, a new parser and type checker have had to be developed. We also touch upon the issue of validating ProB.

J. Falampin (✉) · H. Le-Dang · M. Mokrani
Siemens SAS IC-MOL, Paris, France
e-mail: jerome.falampin@gmail.com

H. Le-Dang
e-mail: hung.ledang@siemens.com

M. Mokrani
e-mail: mikael.mokrani@siemens.com

M. Leuschel · D. Plagge
University of Düsseldorf, Düsseldorf, Germany

M. Leuschel
e-mail: leuschel@cs.uni-duesseldorf.de

D. Plagge
e-mail: plagge@cs.uni-duesseldorf.de

A. Romanovsky, M. Thomas (eds.), *Industrial Deployment of System Engineering Methods*, DOI 10.1007/978-3-642-33170-1_4,

4.1 Introduction

Siemens has been developing Communication-based Train Control (CBTC) products using the B-method [1] since 1998 and has over the years acquired considerable expertise in its applications. Siemens uses Atelier B [11], together with automatic refinement tools developed in-house, to successfully develop critical control software components in CBTC systems. 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 the system ensuring the highest Safety Integrity Level (SIL4). This railway software development process fully complies with current railway standards (EN50126, EN50128, EN50129) and significantly reduces the test cost at the validation step. In fact, the unit test is no longer needed in this process as it is 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 (Metro Est-Ouest Rapide). To quote [10], "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 developed and installed worldwide by Siemens [2, 3]: San Juan metro (Puerto Rico, commissioned in 2003), New York metro Canarsie line (USA, commissioned in 2009), Paris metro line 1 (France, commissioned in 2011), Barcelona metro line 9 (Spain, commissioned in 2009), Algiers metro line 1 (Algeria, commissioned in 2011), Sao Paolo metro line 4 (Brazil, commissioned in 2010) and Charles de Gaulle Airport Shuttle line 1 (France, commissioned in 2005), among others.

Relationship to DEPLOY While the work described in this chapter was carried out within the context of DEPLOY, Siemens uses the classical B rather than Event-B in their current development process. Indeed, neither Event-B nor the Rodin tool is at the time of writing ready for software development at the industrial level. However, the classical B and Event-B clearly 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 unfortunately is 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 subsections, each of which is controlled by safety-critical software called ZC (Zone Controller). A Zone Controller contributes to the realisation of the ATP (Automatic Train Protection) and ATO (Automatic Train Operation) functions of CBTC systems for a portion of the track; these include train location, train anticollision, overspeed prevention, train movement regulation and train and platform door management, among others. In order to avoid multiple development, each ZC is made from a generic B-model and data parameters that are specific to a subsection and a particular deployment (cf. Fig. 4.1).

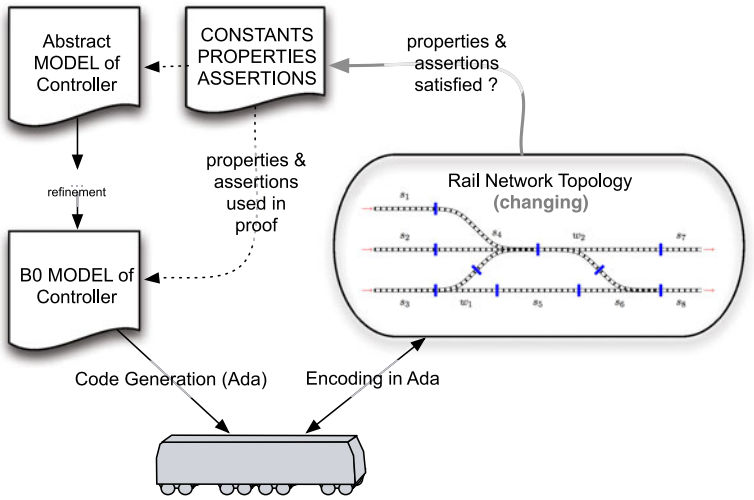

Fig. 4.1 A need for ZC data validation

These data parameters take the form of B functions describing, for example, tracks, switches, traffic lights, electrical connections and possible routes. These B functions are typically regrouped in basic invariant machines in which assumptions about topological properties are defined in the PROPERTIES clause. The proofs of the generic B-model rely on assumptions about data parameters, for instance, those about the topological properties of the track. We therefore have to make sure that parameters used for each subsection and deployment actually satisfy 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 make the assumption that the slope is between -5 % and $+5$ %, and this 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 $+5$ % in the track data (and, of course, that the track data is correct regarding the actual track). More generally, data validation is needed when generic software is used with several sets of specific data.

4.2 An Approach to Validating ZC Data Using Atelier B

Siemens has developed the following approach to validating ZC data:

1. The parameters and topology are extracted from concrete Ada programs (each of which corresponds to an invariant basic machine in the B model; in B, a basic machine does not have a refinement or an implementation but 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 wants to check not only 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 Ada programs rather than directly from a 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 for the concrete topology and parameters represented in basic machines are translated into B assertions in the relevant assertion machines. The macros derived from the actual data extracted from Ada programs are used to define the constant values and these definitions are realised as properties in the assertion machines.

   In B, assertions are predicates that should follow from the properties or invariants, and that 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 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. The assertions for which the proof is unsuccessful are investigated manually.

Problems with the Existing Process This approach initially worked quite well for ZC data, but then it ran into considerable problems:

- First, if the proof of a property fails, the feedback from the prover is not very useful in locating the problem (and it may be unclear whether there is in fact a problem with the topology or "simply" with the power of the prover).
- Second, and more importantly, the constants are currently 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 the maximum memory possible 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. For example, for the San Juan development, 80 properties (out of 300) could not be checked by Atelier B either automatically or interactively (within reasonable time; sometimes just loading the proof obligation results in failure 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 for the compatibility constraints of all possible itineraries on paper), which is very costly and arguably less reliable than automated checking. For the San Juan development, this meant about one man-month of effort; this is likely to be more for larger developments, such as the Canarsie line project [3].

4.3 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 [5, 6] has capabilities for dealing with B properties in order to animate and model-check B models. The big question was whether the technology would scale to handle 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, on 8 July 2008 Siemens sent the STUPS team at the University of Düsseldorf the models for the San Juan case study. There were 23,000 lines of B spread over 79 files. Of these 79 files, two were to be analysed, containing a simpler and a more elaborate model. It then took STUPS some time to understand the models and get them through the new parser, whose development was being finalised at that time.

On 14 November 2008 STUPS were able to animate and analyse the first model. It 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 8 December 2008 STUPS were finally able to animate and validate the more elaborate model. It revealed four errors. Note that the STUPS team was 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 identical to those previously uncovered by Siemens using manual inspection. The manual inspection of the properties took Siemens several weeks (about a man-month of effort). With ProB 1.3.0, checking the properties takes 4.15 seconds, and the assertions 1017.7 seconds (i.e., roughly 17 minutes) on a MacBook Pro with 2.33 GHz Core2 Duo. More information on this was included in the presentation at FM 2009 [7] and in the ensuing journal paper [8].

4.4 RDV: Railway Data Validator

In the first experiments, ProB was used with great success on the same IVP instead of Atelier B. But the creation of the IVP was still problematic, with little atomisation. As described in Sect. 4.1, each IVP is an encoding of specific wayside configuration data in B; this is required for validating the configuration data against the formal properties in the generic B project.

The goal was to create a tool that could automatically generate B projects (containing assertion machines), run ProB on the created B projects and compile the result in a synthesis report. In addition, this tool should work not only on ZC data, but also on CC (carborne controller) data, which were not formally validated before the use of ProB. Indeed, in CC software development, as shown in Fig. 4.2, the topology data are contained in text files and loaded “on the fly” by the CC software component when needed. Therefore, in order to enable 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 the basic invariant machines to create assertion machines for the segment in question.

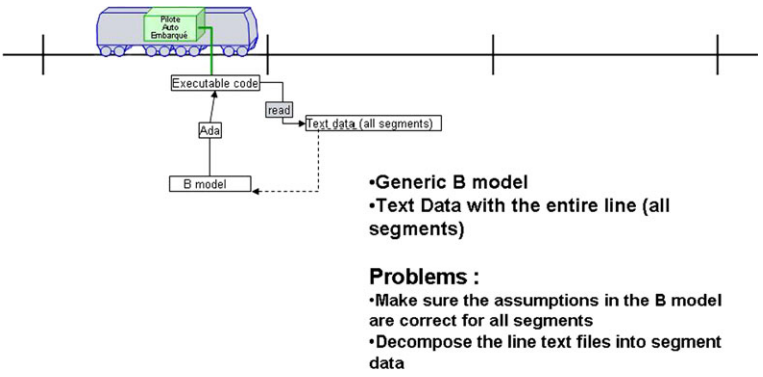

Fig. 4.2 CC data validation

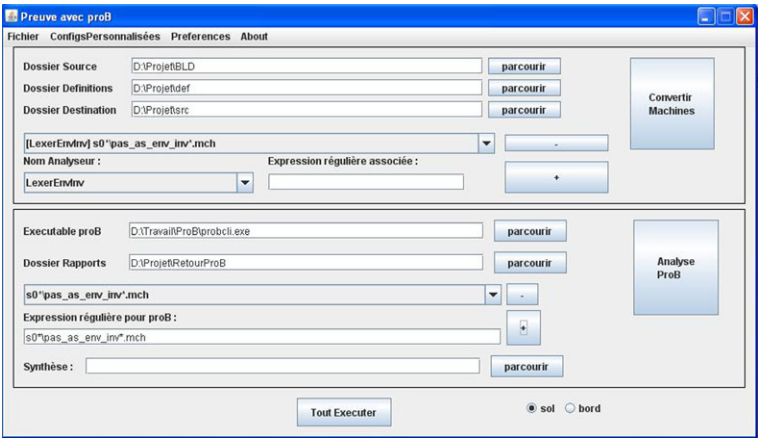

Fig. 4.3 RDV interface

The RDV is a new tool developed by Siemens. Via a graphical interface (cf. Fig. 4.3), the RDV provides the following services:

IVP Generation This function generates an IVP for a subsection in the case of ZC or a segment in the case of CC. The generated IVP is almost ready to be used by ProB or Atelier B. True, we still need some manual modification related to properties of non-function constants; however, in comparison with the tool used previously, manual modification on generated B machines is significantly reduced. In addition, with the file selection function based on regular expressions, the RDV enables the generation of a subset of assertion machines (i.e., only machines with modifications). Moreover, the automation of CC IVP creation is of great help to safety engineers as there are about several hundred IVP to be created (one project per segment).

ProB Launch The RDV enables users to parametrise ProB before launching it. ProB is called on each assertion machine in order to analyse the assertions contained in each machine. There is no need to have B experts for data validation to be carried out. Indeed, B experts are required only in case of a problem, whereas in the previously used process, B experts were required in any case, to carry out long and laborious tasks. In addition, ProB significantly reduces the time for checking data properties: from two or three days with the old tool to two or three hours per project with the RDV. Analysing CC assertion machines with ProB significantly reduces interaction with users as one does not need to launch Atelier B several hundred times on the created CC IVP.

Assertion-Proof Graph Generation This function provides a graphical way of investigating the proof of an assertion. It is based on a service provided by ProB for computing values of B expressions and truth values of B predicates, as well as all sub-expressions and sub-predicates. This is very useful because users often want to know exactly why an assertion fails. This was a problem with the Atelier B approach: when a proof fails, it is very difficult to find out why it did so, especially when large and complicated constants are involved.

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

- Values used when initialising machines;
- Details of proof for each assertion checked;
- Result of the analysis (true, false or time-out).

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;
- Errors occurring when executing ProB (e.g., missing file).

When all report and error files have been generated, an HTML synthesis report is issued. This report gives the result (the number of false/true assertions, time-outs, unknown results) for each sector or segment. For each B component, a hyper-link to the detailed error and report files provides access to the results of the component (to show which assertion is false, for instance).

4.5 ZC Data Validation

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

As shown in Fig. 4.1, the track is decomposed into several sectors (from two to ten); there is one zone controller dedicated to each sector. Ada data is linked with

Ada code transcoded from the generic B model. Below is a B machine with some properties of 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 follows:

```
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;
```

By using the 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 not changed for traceability purposes. The ASSERTIONS clause in the new machine contains predicates which were in the PROPERTIES clause of the original 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_ surcharge, which were derived from the configuration data INV_ZAUM_QUAI_I.
- inv_zaum_troncon_i_indetermine and inv_zaum_troncon_i_surcharge, 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
        }
```

Macros inv_zaum_quai_i_indetermine and inv_zaum_quai_i_ surcharge 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, 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}
```

Macros inv_zaum_troncon_i_indetermine and inv_zaum_troncon_i_surcharge 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 the PROPERTIES clause) and the definition files (in the DEFINITION clause), and displace the assumptions on data in the ASSERTIONS clause. This

modification will lead to some proof obligations "data" = "assumptions" being generated. For example, with inv_zaum_quai_i, we will have to prove the following (simplified) 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
```

4.6 Industrial Projects Validated Using the RDV

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

ALGIERS line 1 (ZC). This is the first driverless metro line in Africa. This line is composed of ten stations over 10 km. The line is divided into two sections, each controlled by a ZC. There were 25,000 lines of B spread over 130 files. ProB was used for the last three versions of this project railway data. Table 4.1 shows the results obtained with ProB on the last version: Each line represents the summary result for one sector. The Predicates column shows the number of assertions to be analysed; the TRUE column represents the number of assertions verified by ProB (no counterexample found by ProB). The FALSE column represents the number of assertions failed by ProB (with a counterexample). The UNKNOWN column represents the number of assertions that ProB dotes not know how to verify. The TIMEOUT column shows the number of assertions that ProB encountered a time-out problem during the analysis.
For each sector, there were two assertions that could not be proved with Atelier B due to their complexity. One is shown below. This property has been proven wrong with ProB using 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 South America. It has 11 stations over 12.8 km and is divided into three sections. The model consisted of 210 files with over 30,000 lines of B. ProB has been used for the last six versions of this project railway data. For the last version, the results are presented in Table 4.2.

Table 4.1 ALGIERS line 1 (ZC)

Sector	Predicates	TRUE	FALSE	UNKNOWN	TIMEOUT
pas_as_inv_s01.html	1174	1164	10	0	0
pas_as_inv_s02.html	1174	1162	12	0	0

Table 4.2 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	1465	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 predicate prover. Here is 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}]))
```

After analysis, we concluded that the problems were not, fortunately, critical. Nonetheless, without ProB, it would have been a lot harder to find them. The assertion-proof graphs helped us better understand the source of the problems.

Paris line 1 (ZC). This is a project to automate line 1 (25 stations over 16.6 km) of the Paris Métro. The line has been gradually upgraded to driverless train while in operation. The first driverless train was commissioned in November 2011. In February 2012, out of 49 trains 17 were driverless, operating in conjunction with manually driven ones. The line is going to be fully driverless by early 2013. It is divided into six sections. The model to be checked is the same as the SPL4 one. ProB has been used for the last seven versions of railway data. The results for the last version are presented in Table 4.3.

Paris line 1 (PAL). PAL (Pilote Automatique Ligne) is a controller line that implements 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. Overall, 2,024 assertions about concrete data of the PAL needed to be checked. ProB found 12 problems in under five minutes. They were later examined and confirmed by manual inspection at Siemens.

CDGVAL LISA (Charles de Gaulle Véhicle Automatique Léger). This is an extension of CDVVAL which was commissioned in 2005. This line is going to be operational in early 2012. The LISA model consisted of 10,000 line of B over

Table 4.3 Paris line 1 (ZC)

Sector	Predicates	TRUE	FALSE	UNKNOWN	TIMEOUT
pas_as_inv_s011.html	1503	1501	2	0	0
pas_as_inv_s012.html	1503	1498	5	0	0
pas_as_inv_s013.html	1503	1496	7	0	0
pas_as_inv_s014.html	1503	1499	4	0	0
pas_as_inv_s015.html	1503	1498	5	0	0
pas_as_inv_s016.html	1503	1498	5	0	0

Table 4.4 Roissy LISA (ZC)

Sector	Predicates	TRUE	FALSE	UNKNOWN	TIMEOUT
ry_pads_as_inv_pa31.html	1038	1038	0	0	0
ry_pads_as_inv_pa32.html	957	957	0	0	0
ry_pads_as_inv_pagat.html	1038	1038	0	0	0

30 files. This project has three sections. ProB has been used for the last 3 versions of this project railway data. The results for the last version are presented in Table 4.4.

4.7 Tool Issues

A crucial consideration when trying to successfully deploy a tool like ProB in an industrial project is that the changes made to the existing models must be minimal because these 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 support the full language used in industry.

To make ProB fit for use in industry, we had to make significant changes to the parser and type checker even to 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 parameterised DEFINITIONS, for tuples that use commas instead of `|->` or for definition files. We realised that the code base of the existing parser was very difficult to extend and maintain and decided to completely rewrite the parser. We decided to use the parser generator SableCC [4], because it allowed us to write a clean and readable grammar. The following will briefly describe some of the issues we encountered in developing the parser and point out where our parser's behaviour deviates from Atelier B's behaviour.

- Atelier B's definitions provide a macro-like functionality with parameters, i.e., a call to a definition somewhere in the specification is syntactically replaced by the definition. The use of such macros can, however, lead to subtle problems. For example, 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 believed that this could easily lead to errors in the specification and decided to deviate from Atelier B's behaviour in this respect. Thus, we treat `2*sm(1,4)` as `2*(1+4)`, which equals 10 as expected.
- Another problem was 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 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 parentheses for expressions (as in `(x;x)`). Substitutions are never put in parentheses because they are bracketed 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, which is important for identifying the source of any errors in specification. To circumvent this problem, we modified the parser generator itself to support source code positions.

The resulting parser deviates from Atelier B's in just some respects. It is relatively rare for an Atelier B model to need to be rewritten to work with ProB.

The Type Checker We re-implemented our type checker for B specifications because the previous version often needed additional predicates to provide typing information. We implemented a type inference similar to the Hindley-Milner algorithm [9], which 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 by using the information provided by the parser, we can highlight an erroneous expression in the source code. This improves user experience, especially for new users.

Improved Data Structures and Algorithms Originally, ProB represented all sets internally as lists. But with thousands of operations on lists to carry out, it performed very badly. We introduced an alternative representation based on self-balancing binary search trees (AVL trees), which provides much faster access to its elements. In this context we examined the bottlenecks of the kernel in light of the industrial models given to us and implemented specialised versions of various operations. For example, 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 sorts the tree elements and just check if the smallest and largest elements are in the interval.

Infinite Sets Several industrial specifications make use of definitions of infinite sets such as `INTEGER \ {x}`. ProB now detects certain patterns and keeps them symbolic, i.e., instead of trying to calculate all elements, ProB just stores the conditions that all elements fulfil and uses them for member checks and function applications. Examples of expressions that can be kept symbolic are

- integer sets and intervals,
- complementary sets (as in the example above),
- some lambda expressions and
- unions and intersections of symbolic sets.

4.8 Tool Validation

In order to be able to use ProB without resorting to Atelier B, Siemens asked the Düsseldorf team to validate ProB. There are no general requirements for using a tool within an SIL 4 development chain; the amount of validation depends on how critical the tool is in the development or validation chain. In this case, Siemens requires

- 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 a description of ProB functions, and a classification of functions into critical and non-critical, along with a detailed description of the various techniques used to ensure a 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.

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

Unit Tests ProB contains over 1,000 manually entered unit tests at the Prolog level. These check, for example, the proper functioning of the various core predicates operating on B data structures. In addition, there is an automatic unit test generator, which tests the ProB kernel predicates in many different scenarios and with different set representations. For example, starting from the initial call,

```
union([int(1)],[int(2)],[int(1),int(2)]),
```

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

the test generator will derive 1,358 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 irrespective of the order in which (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. The models are loaded, the saved animation traces replayed and the models run through the model checker. The tests have turned out to be extremely valuable in ensuring that a bug once fixed remains fixed. They are also very effective in uncovering errors in arbitrary parts of the system (e.g., the parser, type checker, the interpreter or the ProB kernel).

Self-model Check with Mathematical Laws This approach allows ProB 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 in 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 of a mathematical law could be found. In other words, 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 as `FIN1`) using the self-model check. Furthermore, we have even uncovered two bugs in the underlying SICStus Prolog compiler using the 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 be easily 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 At the moment we also input 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. In this way we test whether the typing information inferred by our tool is compatible with the Atelier B type checker. (Obviously, 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 be easily 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 the machines is correct. The latest version of ProB has a command for cross-checking the typing of the internal representation with Atelier B in this manner.

With the help of testing we identified 26 errors in the B syntax as described in the Atelier B English reference manuals, upon which our pretty printer and parser were 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 succeeded for a particular B predicate, then a bug in ProB would be uncovered. If both fail, then either the B predicate is undefined or we have a bug in ProB. This validation method can detect errors in the predicate evaluation parts of ProB, i.e., in the treatment of the Boolean connectives $\vee$, $\wedge$, $\Rightarrow$, $\neg$, $\Leftrightarrow$, quantification $\forall$, $\exists$, and the various predicate operators such as $\in$, $\notin$, $=$, $\neq$ and $<$. Redundancy cannot detect bugs inside expressions (e.g., $+$, $-$) or substitutions (but the other validation methods mentioned above can).

4.9 Conclusions

In this chapter we have described a 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 outlined some of the effort and features that were required in moving from a tool capable of dealing with medium-sized examples toward 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 to 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. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)

2. Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project. In: Treharne, H., King, S., Henson, M.-C., Schneider, S.-A. (eds.) Proceedings of ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users. Lecture Notes in Computer Science, vol. 3455, pp. 334–354. Springer, Berlin (2005)
3. Essamé, D., Dollé, D.: B in large-scale projects: The Canarsie line CBTC experience. In: Julliand, J., Kouchnarenko, O. (eds.) Proceedings of B 2007: 7th Int. Conference of B Users. Lecture Notes in Computer Science, vol. 4355, pp. 252–254. Springer, Berlin (2007)
4. Gagnon, E.: SableCC, an object-oriented compiler framework. Master thesis, McGill University, Montreal, Canada (1998). http://www.sablecc.org
5. Leuschel, M., Butler, M.-J.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) Proceedings FME 2003: Formal Methods. Lecture Notes in Computer Science, vol. 2805, pp. 855–874. Springer, Berlin (2003)
6. Leuschel, M., Butler, M.-J.: ProB: An automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
7. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models. In: Cavalcanti, A., Dams, D. (eds.) Proceedings FM 2009. Lecture Notes in Computer Science, vol. 5850, pp. 708–723. Springer, Berlin (2009)
8. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models with ProB. Form. Asp. Comput. **23**(6), 683–709 (2011)
9. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**, 348–375 (1978)
10. Siemens: B method—optimum safety guaranteed. Imagine **10**, 12–13 (2009)
11. Steria, Aix-en-Provence. France. Atelier B, user and reference manuals (2009). http://www.atelierb.eu/