Using B and ProB for Data Validation Projects

Dominik Hansen, David Schneider, and Michael Leuschel(✉)

Institut Für Informatik, Heinrich-Heine-Universität Düsseldorf, Universitätsstr. 1, 40225 Düsseldorf, Germany
{dominik.hansen,david.schneider,michael.leuschel}@hhu.de

Abstract. Constraint satisfaction and data validation problems can be expressed very elegantly in state-based formal methods such as B. However, is B suited for developing larger applications and are there existing tools that scale for these projects? In this paper, we present our experiences on two real-world data validation projects from different domains which are based on the B language and use ProB as the central validation tool. The first project is the validation of university timetables, and the second project is the validation of railway topologies. Based on these two projects, we present a general structure of a data validation project in B and outline common challenges along with various solutions. We also discuss possible evolutions of the B language to make it (even) more suitable for such projects.

Keywords: B method · Constraint programming · Timetabling · Scheduling · Railway

1 Introduction

Data validation[1] ensures that software operates on correct, clean data and is typically done by checking validation rules or constraints. We have previously argued that B [3] is a very expressive language to encode constraint satisfaction problems [20,24], and many data validation problems can be expressed as such. Other works have demonstrated that B is useful to express properties about data and to validate them using ProB [17], particularly in the railway domain [2,4–6,15,19].

In this paper we report on our experiences using B in combination with ProB to create tools for data validation. We have used the B language to express parts of our program's domain logic and the rules to validate data, and embedded these B models into running applications by executing the formal models with ProB without relying on code generation. It would also be possible to express these kinds of validation problems in other formal languages such as Alloy [13] and TLA$^+$ [14]. Based on our experiences with these languages and the corresponding tools, we believe that the combination of B and ProB best meets the requirements for the data validation task. Our explicit goal is to explore

[1] http://www.data-validation.fr.

M. Butler et al. (Eds.): ABZ 2016, LNCS 9675, pp. 167–182, 2016.
DOI: 10.1007/978-3-319-33600-8_10

the applicability and scalability of this combination for projects of industrial strengths.

Based on two projects, described below, we will discuss different aspects of using B within such an application and discuss the approaches taken as well as the limitations encountered, i.e. where we had to depart from or extend the language to suit our needs.

Curriculum Validation is a project [24] in which we are creating an interactive tool to validate timetables and curricula for various faculties and courses at our university. Curriculum validation is related to timetabling [8–10,22,23]. Timetable validation differs in the sense that we are interested in the feasibility of studying an entire curriculum, spanning several semesters instead of planning out time slots for classes within one semester. The central task is to detect whether it is possible for a student to attend all classes required for a degree in the manner described by the curriculum, by a suitable choice of alternatives. In case a course contains feasibility conflicts, the tool provides assistance to detect one of the potentially many sources of the conflict by computing a unsatisfiable core of the data with respect to the validation rules; additionally we provide support in finding alternative time slots which solve such conflicts. The largest dataset provided by one of the participating faculties currently consists of 31 courses with 1343 classes and 1578 scheduled events in these classes.

Validation of Railway Topologies is the second project discussed in this article and part of a collaborative research project with Thales Transportation Systems GmbH on applying formal methods for the software development process of the Radio Block Centre (RBC). The RBC is a communication unit of the European Train Control System (ETCS) exchanging messages with trains and interlockings. One of our challenges in this context is to validate the so-called engineering rules over concrete track data. The track data is a representation of the real railway infrastructure and signalling system. Engineering rules are implementation-related rules which result from the concrete RBC implementation. This means, that the concrete RBC implementation is guaranteed to work correctly only if the concrete track data satisfy the engineering rules. For example, a simplified engineering rule requires that two signals for the same direction should not be located at the same position. The modelled engineering rules are validated on different track topologies. The biggest topology contains 1362 track segments, 457 points, 1089 balise groups and 445 signals.

Both projects rely on ProB as the tool to evaluate the models. ProB is an animator and model-checker for the B method with support for validation and proofs. ProB also is a constraint solver for the B language, which is required in an animation and model-checking scenario to efficiently find values for constants, guards and parameters of operations.

The idea of using formal method languages and tools to perform data validation has been explored in the past, e.g. by Abo and Voisin [2] or Lecomte et al. [15] among others. Our intention here is to outline the common structure

and challenging aspects of data validation projects based on what we have identified in the aforementioned projects. The domains and requirements of these two projects are quite different, and all work has been done independently (i.e. by different people). Still, similar challenges were faced during the modelling process.

In the following sections we will name these challenges, discuss different language constructs of B and argue how they can be applied in modelling data validation problems. Moreover, we will outline areas where we have extended the B language to overcome some limitations we faced evaluating the models with ProB.

2 The Big Picture

Before describing the details of the data validation process we will discuss the big picture, outlining the design and architecture that emerged from both projects mentioned in the previous section.

```
!signal1, signal2.(
  signal1 : Signals & signal2 : Signals & signal1 /= signal2
  & Signal_Direction(signal1) = Signal_Direction(signal2)
   => not(Signal_TrackSegment(signal1) = Signal_TrackSegment(signal2)
          & Signal_Position(signal1) = Signal_Position(signal2)))
```

Fig. 1. Modelling of an engineering rule as a validation predicate

The general idea is to create B models that define validation predicates which are evaluated against the state of the model. The variables and constants are derived from external data we want to validate. Figure 1 shows the formalisation of the validation rule mentioned in the introduction Section where two signals should not be located at same position if they are valid for the same direction.

The projects discussed in this paper follow the general architecture shown in Fig. 2. By building data validations tools based on the B language we have identified the following concerns: The first is getting the external data from a given source into a B model which is discussed in Sect. 3. Choosing a way to represent the data is a further concern, where it is important to choose a representation and B data types suited for the validation process while keeping the import process as simple as possible; this is discussed in Sect. 4. Some validation rules rely on derived data (e.g., signals reachable from a point) which has to be computed from the imported data. In Sect. 5 we present different approaches to structure derived data in B. One purpose of the B method is to model algorithms and prove their correctness. However, are these models suitable for use by ProB to calculate results? Section 6 describes different approaches to model an algorithm in B such that it can be efficiently evaluated by ProB. Another concern is how to control the validation process from an external application.

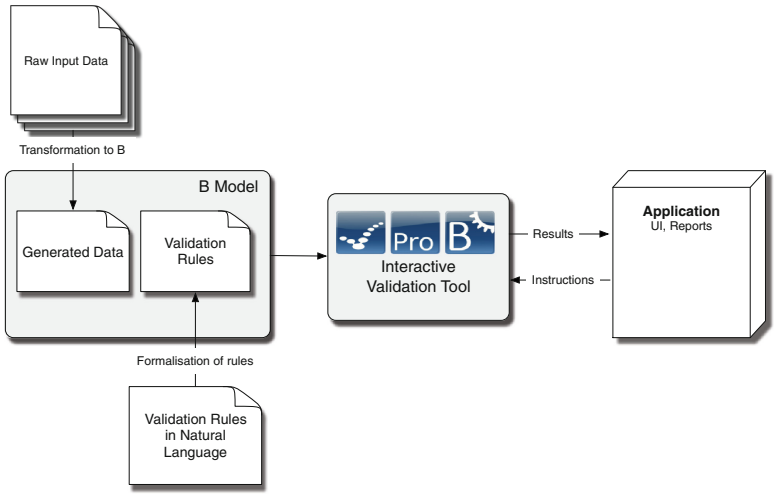

Fig. 2. Generalised architecture of ProB based data validation project

In Sect. 7 we discuss different ways to interact with the model. Finally, in Sect. 8 we briefly discuss how to reuse an existing validation model in similar projects.

3 Preparing Data for Use with a B Model

When used for data validation, our tools obviously depend on externally provided data [2,16], which has to be converted to B format, in order to be validated with ProB. Raw input data is provided in a variety of formats as used in the different domains such as Excel, CSV or XML documents.

In both projects we have opted to create tools that read and parse the externally provided data and generate a text file containing a B model of the data. The data will be accessible as a series of constants in the model.

Having an external tool keeps any knowledge about the raw data format out of the B models; but of course it raises a series of concerns. One is having to maintain an additional tool which has to generate valid B. Also the chosen data representation has to be kept in sync between the import and the validation tools.

Another concern is that, in a safety critical environment, the import tool itself has to be validated. The topology validation project takes a direct approach by avoiding putting too much knowledge into the transformation step, keeping it as simple as possible. In this approach the transformation maps the input structure of the data (XML) to B data structures and copies the values of attributes as uninterpreted strings. To ensure that all data from the input document is represented in the B model we use a back-translation (from the B model to XML) and compare the generated XML document with the source document. The back-translation is done in order to certify the translation tool and ensure that no data has been left out.

In the case of the curriculum validation tool the data is not only used for validation purposes but also to populate the application's user interface, hence we have chosen a two step approach that does not directly generate a B machine, but rather import the data into a database. The information in the database is later used to generate the actual B representation of the model at runtime. Additionally, the database is used in the application to persist changes and as the data source for the UI. Since the data is used in multiple places we map the values in the raw data to the most adequate types in the database and later to the corresponding B types.

Are There Any Alternatives? There are many alternative approaches that could be pursued to import data into a B model. E.g. instead of generating a B model with the data as constants, it would be possible to have B operations which incrementally add values to variables containing the data. These operations could be executed in various ways, e.g., using the Java API for ProB. Finally, ProB exposes external functions to B that make it possible to, e.g., load data from CSV files; these features could be extended for additional data sources (see Sect. 6.3).

4 Data Representation

Hand in hand with the decision on how to import data into a B model goes the choice of proper B data-structures to represent the data. This representation should ideally follow the structure of the source data, and additionally lend itself to be used and manipulated in B. Choosing a good representation for the problem is crucial for the complexity and readability of the model. In [11] Hayes et al. discuss some of these issues on the examples of a simple database in VDM and Z. In B, one could encode database records as nested pairs. A quaternary relation over course identifiers, semester, weekday, and starting hour could thus be represented as:

```
db = { (((course1 |-> sem2) |-> monday) |-> 14),
       (((course2 |-> sem1) |-> friday) |-> 9) }
```

In order to access the first and second element of a pair, B provides the prj_1 and prj_2 operators. However, in B accessing a certain field of a nested pair is very cumbersome, as we have to unfold the nested pair until we reach the desired field.[2] Another alternative is to use records with named fields:

```
db = { rec(course_id: course1, semester: sem2, weekday: monday,
    starting_hour: 14), rec(course_id: course2, semester: sem1,
    weekday: friday, starting_hour: 9) }
```

We can easily access a field of a record r by using the quote operator: $r'course\_id$. Compared to the encoding as nested pairs, records are more readable, especially if there are a large number of fields. Otherwise, constructing a

[2] In addition, the types of the arguments have to be provided for prj_1 and prj_2; e.g., $\text{prj}_2((COURSE \times SEMESTER) \times WEEKDAY, \mathbb{Z})(v)$.

record is more verbose than constructing nested pairs. Since, this part of the model is automatically generated, the verbose encoding is not an issue.

A third alternative is to create B functions for each attribute of the data record mapping a unique identifier to the corresponding attribute value. An identifier of a data record could be a unique number generated by the translator or a certain attribute of the data record. In case of our example, we could choose the attribute `course_id` as the unique identifier:

```
course_id__semester = {course1 |-> sem2, course2 |-> sem1}
course_id__weekday = {course1 |-> monday, course2 |-> friday}
course_id__start_hour = {course1 |-> 14, course2 |-> 9}
```

While this approach works well for simple tables such as in Excel or CSV documents, it would become inconvenient for nested data structures, e.g. if a value of a field is itself a set of data records such as a sub-tag of a XML document. In this case, the translation tool first has to transform the nested data structure to a relational database schema. Subsequently, the translator has to create a B function for each attribute of each table of the relational database.

One advantage of the last alternative is the handling of optional fields. Indeed, when no field value is present for a data record, we just omit the corresponding identifier from the domain of the field accessor function (i.e., we use partial functions rather than total functions). For the other two approaches, optional fields pose more of a challenge. Due to the strong and strict typing of B it is not possible to create partial records or to omit a field of a nested pair. One solution is to introduce a special `NULL` value for each B datatype, e.g. the empty string (`" "`) for the `STRING` type. However, we have to ensure that the `NULL` value is not a regular value in the source data. For other data types such as `INTEGER` it is more intricate (which integer to use?) and for the `BOOL` type impossible.

This directly leads to a further aspect of the translation. How to represent the values of the data records? They could be represented either as uninterpreted strings of data copied verbatim from the raw data input in the transformation step. Alternatively the data values could be represented using the most appropriate B data types, e.g. `INTEGER` for numbers, and enumerated sets for values from a set of known values. The first approach has the advantage of a very simple translation process and all the relevant knowledge is encoded in the B model. The drawback is now, however, that the data has to be translated in the B model, which typically requires extensions to the B language which are available in ProB (e.g. transforming a `STRING` value to an `INTEGER` value).

In both projects, we have chosen the record representation for the data. As already mentioned in the previous section, the timetabling tool maps the raw data to the corresponding B data types. In case of the topology validation project, all data values are represented as uninterpreted strings and the processing of these strings is part of the B model.

5 Means of Abstraction – Structuring and Auxiliary Constructs

Abstractions [1] in programs and also models control complexity, encourage reuse and make testing easier. Different parts of the B language offer different ways to abstract and structure models and programs. There are certain concepts that are applicable at the machine and operation level while others are applicable on the predicate and expression level.

Machines and Operations. On the machine level sub-problems can be structured as machines for each sub-aspect which communicate through the execution of operations. The visibility of machines and their variables and operations can be controlled using different machine composition mechanisms such as `SEES`, `USES` and `INSTANCE`.

On the level of a single operation the substitution language provides several expressions that are useful, either if-then-else for control flow or `LET` constructs to introduce scoped variables.

Expressions and Predicates. Within the mathematical language of B, constants can be used to globally save precomputed values whose computation might be expensive and should not be evaluated more than once. Figure 3 shows the calculation of the conflict relations of two different signals placed at the same position and valid for the same direction. Note, that the calculation corresponds to a SQL statement making a self join on a signal table. Moreover, constants can be used to store certain calculations in the form of lambda functions which can be used in different parts of the model. However, constants are not applicable for intermediate results which can not be precomputed globally because they depend on additional information or parameter values.

```
CONSTANTS
  ConflictRelation
PROPERTIES
  ConflictRealtion =
    UNION(r1,r2).(r1 : SignalRecords & r2 : SignalRecords
      & r1'elementID /= r2'elementID & r1'trackSegment = r2'trackSegment
      & r1'position = r2'position & r1'direction = r2'direction
      | {r1'elementID |-> r2'elementID})
```

Fig. 3. Calculating the conflict relation of two signals placed at the same position.

LET for Predicates and Expressions. In complex expressions or predicates it is often useful to introduce a shorthand for certain values or expressions, B only supports `LET` in the context of substitutions, nonetheless it might be useful for predicates and expressions. For example, an existential quantification (`#x.(x=E & P)`) can be used within predicates to achieve a result similar to a `LET`. ProB

tries to identify existential quantifications that only have a single value and treats them specially. Within set-comprehensions an existential quantification could also be used (`{x| #y.(y=E & P)}`), but the following pattern using the domain of a set of pairs is (generally) more efficient in ProB: `dom({x,y| y=E & P})`. For expressions which denote a set of values, one can use `UNION(y).(y=E| S)`. Ideally, however, rather than using these workarounds, we would argue for adding explicit `LET` constructs to the B language for expressions and predicates.

DEFINITIONS. One of the available methods of decomposing larger predicates or expressions into smaller reusable components are `DEFINITIONS` (comparable to macros). The use of `DEFINITIONS` carries some issues that have to be kept in mind. Although they are textual replacements, ProB requires every definition to be syntactically correct on its own, so certain compositions patterns are not possible. Care is also needed with regard to naming conflicts, quantifications not captured in the `DEFINITION` where variables escape the scope (see, e.g., [12]). Take for example the definition `even(x) == (#y.(y:1..x & 2*y=x))`. Evaluating the predicate `even(4)` yields true. However, if we have a machine variable `y` whose value is 4 and evaluate `even(y)` we obtain false; the definition call was rewritten to `#y.(y:1..y & 2*y=y)`. Another issue is unintended repeated computation of arguments. Indeed, the arguments of a definition may get replaced multiple times and then also executed multiple times by ProB. Take, e.g., the definition `POW3(x)==x*x*x` and the call `POW3(f(1))`. The latter gets transformed into `f(1)*f(1)*f(1)`, resulting in repeated computations of `f(1)`. A pattern we have used to avoid this is to create a variable within each `DEFINITION`, which is assigned with the passed argument and used instead of the original parameter to avoid unintentionally causing repeated computations of the same expression:

```
DEFINITIONS
EXAMPLE(aa, bb) == #(va, vb).(
  va = aa & vb = vb & <predicate over va and vb> );
```

For the reasons described above, `DEFINITIONS`, although they are a useful method to store and structure expressions and predicates, should be used carefully, in particular for big expressions with parameters.

6 Using B to Express Computations

Sometimes data validation relies on complex concepts, which cannot be easily described as B predicates. In those cases it can be more convenient to describe these concepts using recursive rules or as fixpoints of iterative algorithms. In this section we show how this can be achieved in a natural B style, while also ensuring that the resulting algorithms can be executed efficiently.

We will discuss different approaches to model an algorithm for sorting a set of numbers into an ordered sequence. Note that the B method does not provide a built-in operator to sort a set. In the particular applications we used the techniques for more complicated constructs, such as a search on a rail way topology with various termination conditions.

6.1 Machines and Operations

First we will discuss the approach of using machines and operations to express the required functionality. Using the machine and substitution semantics of B to express computations has the clear advantage of having all tools and features of the B method at our disposal. Figure 4 shows a stateless query operation calculating the sorted sequence for a given input set.

```
out_sortedSequence <-- Sort_OP(p_set) =
  PRE p_set : POW(INTEGER) THEN
    out_sortedSequence : (
        out_sortedSequence : iseq(p_set)
        & ran(out_sortedSequence) = p_set
        & !i.(i : 1 .. size(out_sortedSequence) - 1
              => out_sortedSequence(i) < out_sortedSequence(i + 1)))
  END
```

Fig. 4. Query operation

However, ProB is not able to evaluate the operation efficiently, i.e. it does not scale for large input sets. Indeed, a naive execution of `Sort_OP` would calculate all possible permutations of the input set to then reject all but one, which is the sorted sequence. ProB's constraint solving can overcome this exponential complexity to some extent,[3] but for larger sequences we are a far cry from the performance of ordinary sorting algorithms. Following the refinement principles of the B method we can implement the abstract operation by a concrete sorting algorithm. Figure 5 shows a selection sort (MinSort) implementation in B. The operation `Sort_OP` exposes the algorithm as a single operation which can be used several times and embedded in different machines.

ProB provides various optimisations for while loops. First, an interesting point is that the variant is evaluated upon entry and gives ProB an upper-bound on the number of iterations.[4] If a certain threshold is exceeded, ProB will pre-compile the body of a while loop, by pre-computing all parts which do not depend on variables modified in the loop. Furthermore, the state of the interpreter is projected onto those variables that are modified.

In our approach, we are not interested in proving the concrete algorithm to be a correct refinement of the abstraction. However, we are interested in the correctness of the sort implementation. Therefore, we use the predicate of the abstract operation as an invariant respectively an assertion on the output of the concrete operation. Note, that in this case ProB is able to check that the predicate holds for a concrete value even for a large input set. Moreover, the termination of the sort algorithm is ensured using a loop variant which

[3] ProB can compute `Sort_OP({3,55,22,44,1,100,20,40,55,88,10,90,200,0,5})` in 0.18 s, despite there being 15!=1,307,674,368,000 permutations.

[4] In many models, the variant actually corresponds exactly to the number of iterations.

```
out_sortedSequence <-- Sort_OP(p_set) =
  PRE p_set : POW(INTEGER) THEN
    VAR v_set, v_seq
    IN
      v_set := p_set; v_seq := [];
      WHILE v_set /= {}
      DO
        v_seq := v_seq <- min(v_set);
        v_set := v_set \ {min(v_set)}
      INVARIANT
        v_set : POW(p_set) & v_seq : iseq(p_set)
        & !i.(i : 1 .. size(v_seq)-1 => v_seq(i) < v_seq(i + 1))
      VARIANT card(v_set)
      END;
      ASSERT ran(v_seq) = p_set THEN out_sortedSequence := v_seq END
    END
  END
```

Fig. 5. Implementation of a sorting algorithm

is observed by ProB. For more complex algorithms such as different search algorithms on railway topologies we have modelled state machines instead of stateless query operations. However, the execution of the these state machines is controlled by a single operation of an additional interface machine.

A small disadvantage of using operations is that the output value of the operation can only be assigned to a variable and the operation can not be used as part of a set comprehension or quantification.

6.2 Recursive Functions

Recursive functions, which are supported by ProB [18], are a very effective way to compactly express certain kinds of algorithms. Figure 6 shows the selection sort algorithm modelled as a recursive function in B. By defining *Recursive_Sort* as an abstract constant we indicate that ProB should handle the function symbolically, i.e. ProB will not try to enumerate all elements of the function. The recursive function itself is composed of two single functions: a function defining the base case and a function defining the recursive case. Note, that the intersection of the domains of these function is empty, and hence, the union is still a function.

However, there are certain constructs that are harder to write (and read) using only the expression language of B, as it has no explicit support for let expressions and if-then-else. Nonetheless it is often easier to express a construct as a recursive function than it is to decompose the steps in order to express it as a machine. In general, the performance of a recursive function is slower compared to the operation/while approach.

Rather than using an explicit recursive call as in Fig. 6, we can also use B's transitive closure operator to compute the fixpoint of a relation. For our example,

```
ABSTRACT_CONSTANTS Recursive_Sort
PROPERTIES
  Recursive_Sort : POW(INTEGER) <-> POW(INTEGER*INTEGER)
& Recursive_Sort =
    %in.(in : POW(INTEGER) & in = {} | [])
      \/ %in.(in : POW(INTEGER) & in /= {}
                  | min(in) -> Recursive_Sort(in\{min(in)}))
```

Fig. 6. Recursive sort function

let us define the relation `step = %(s,o).(s/={} | (s\{min(s)},o<-min(s)))` which encodes one recursive step of selection sort (s is the set to sort, o is the output sequence so far). For a start set $in = \{4, 5, 2\}$ we can now compute `closure1(step)[{(in,[])}]` resulting in $\{(\{4, 5\} \mapsto [2]), (\{5\} \mapsto [2, 4]), (\varnothing \mapsto [2, 4, 5])\}$. As we can see, the result of sorting a set in can be obtained by calling `closure1(step)[{(in,[])}]({})`.

6.3 External Functions

There are certain concepts that are not part of the B language, e.g. mathematical functions such as *sin*, *cos*, etc. Other computations are difficult or impossible to express using only predicates and expressions, while others might be too slow to evaluate purely in B. ProB offers a mechanism named `external functions` to add and expose new constructs to B. In our sorting example this might look as follows:

```
DEFINITIONS
 SORT(X) == [];
 EXTERNAL_FUNCTION_SORT == (POW(INTEGER)-->seq(INTEGER))
```

The function `SORT` is implemented in Prolog as part of the ProB core and exposed in B as a definition. In order to define a syntactically correct `DEFINITION` we use the empty sequence as a dummy value ensuring type correctness. The second definition tells ProB the type of the external function.

External functions provide the best performance for specific computations, by removing the interpretation overhead but at the same time are opaque to the user and at this point in time need to be integrated explicitly into the ProB Prolog kernel to be available in the language.

6.4 Further Language Extensions

In the topology validation project we have introduced further language constructs that provide a uniform schema to write validation rules. Wherein, the validation predicates are embedded in special `RULE` operations. Figure 7 shows a simplified schema of a `RULES_MACHINE` which contains several `RULE` operations and will be translated to an ordinary B machine. The result of a `RULE`

operation can be stored by using the new RULE_SUCCESS or RULE_FAIL(.) keywords. The argument of the RULE_FAIL(.) keyword is the message reported in case of a rule violation. For each rule operation an ordinary variable is generated in the translated B machine containing the result of the rule evaluation (i.e. ‘‘FAIL’’,‘‘NOT_CHECKED’’ or ‘‘SUCCESS’’). By using additional guards we are able to define dependencies between rules (using the new keyword DEPENDS_ON_RULES) or disable a rule if necessary. The model itself is nondeterministic in the sense that different rules can be executed at the same time if their guards are satisfied. Thus, we are not forced to define an explicit execution order of all rule operations and can use ProB's animation feature to conveniently execute a certain operation. To ease the writing of a rule we developed a new FORALL substitution which can be used to define an error message of a rule by conveniently accessing the variables of a universal quantification.

7 Interaction with the Model

There are several ways the main software can interact with the B validation model. Depending on the kind of application, one could animate or model check the B model, execute a B operation or evaluate B expressions or B predicates (assertions) on a certain state of the model.

The ProB Java API (aka. ProB 2.0[5]) provides facilities to use ProB in applications running on the JVM. Through this API it is possible to access the functionalities mentioned above and to translate B data types to and from appropriate Java types.

In case of the curriculum validation project, the tool itself is a Java application that embeds the model and ProB. We expose all features provided by the model as B operations that represent the public API of the model. These operations are evaluated, using ProB's animation facilities, with externally provided parameters to validate the different curricula. The validation operations return a list of variables that represent one possible choice of subjects to successfully finish a degree. Furthermore, we use the result computed for a feasible curricula to generate a PDF timetable for students with a recommend choice of subjects for their studies.

In the topology validation project the model is used as an independent validation tool with the goal to generate validation reports about the input data. Each engineering rule is modelled as one or more RULE operations containing the validation predicates (see Sect. 6.4). By using more than one RULE operation for an engineering rule the complexity of a natural language requirement can be decomposed into several simple and readable validation predicates. The advantage of a RULE operations, compared to a listing of all validation predicates as part of the ASSERTIONS section, is that a B operation defines a clean interface to perform the evaluation of the individual rules and to access result values and counterexamples. In order to generate a complete validation report and to validate all possible rules, we construct a trace of the model using ProB's execute

[5] https://www3.hhu.de/stups/prob/index.php/ProB_Java_API.

```
RULES_MACHINE Rules
SEES Features
OPERATIONS
 RULE rule1 = ...;
 RULE rule2 = ...;
 RULE rule3 =
  SELECT
   DEPENDS_ON_RULES(rule1, rule2)
   & Enabled(feature1) = TRUE
  THEN
   FORALL
    p1, p2
   WHERE
    P(p1,p2)
   EXPECT
    Q(p1,p2)
   THEN
    RULE_SUCCESS
   ELSE
    VAR errorMsg
    IN
     errorMsg := Exp(p1,p2);
     RULE_FAIL(errorMsg)
    END
   END
  END
END
```

```
MACHINE Rules
SEES Features
VARIABLES rule1, rule2, rule3
INVARIANT
 rule1 : {"NOT_CHECKED",
     "FAIL", "SUCCESS"}
 & rule2 : {"NOT_CHECKED",
     "FAIL", "SUCCESS"}
 & rule3 : {"NOT_CHECKED",
     "FAIL", "SUCCESS"}
INITIALISATION
 rule1 := "NOT_CHECKED"
 || rule2 := "NOT_CHECKED"
 || rule3 := "NOT_CHECKED"
OPERATIONS
 res,ce <-- rule1 = ...;
 res,ce <-- rule2 = ...;
 res,ce <-- rule3 =
  SELECT
   rule3 = "NOT_CHECKED"
   & rule1 = "SUCCESS"
   & rule2 = "SUCCESS"
   & Enabled(feature1) = TRUE
  THEN
   IF
    !(p1,p2).(P(p1,p2)
       => Q(p1,p2))
   THEN
    rule3 := "SUCCESS"
    || res := "SUCCESS"
    || ce := ""
   ELSE
    ANY p1,p2
    WHERE
     P(p1,p2) & not(Q(p1,p2))
    THEN
     VAR errorMsg
     IN
      errorMsg := Exp(p1,p2);
      rule3 := "FAIL"
      || res := "FAIL"
      || ce := errorMsg
     END
    END
   END
  END
END
```

Fig. 7. Translation of a `RULES_MACHINE` to an ordinary B machine

command until all operations are covered. By doing this, we eliminate the overhead which would be introduced by performing a complete model checking run on the non-deterministic model (i.e. evaluating an operation several times).

8 Configuration Management

Configuration management, i.e. how to reuse rules and infrastructure for similar or related projects which differ in very specific aspects, is very important in the context of data validation. For example, in the case of curricula validation, there are subtle differences amongst faculties in the overall structure or how the students choose classes. We have explored two different approaches, to tackle this issue.

One approach is that of a Software Product Line (SPL) [7], where the system would create, from a selection of predicates and evaluation rules a machine that composes them according to a provided configuration. A further approach would be to search for and find a data representation and formulation of the validation rules that is general enough to be applied to more than one particular instance. Such a generic model can contain variation points to control specific aspects of the validation process that differ from project to project. For example, the rule in Fig. 7 is only tested when two particular features are selected.

In both projects we have settled for a combination of both approaches, automatically generating certain parts of our models and additionally configuring the generic parts.

9 Conclusion and Future Work

In this paper we have presented two data validation projects where we have expressed the validation rules in B. Based on the experiences gathered and the similarities between the projects, we have discussed different relevant areas and presented our architecture and design decisions as well as possible alternatives.

We have identified the aspects of data validation that can be easily and elegantly expressed in B such as deriving intermediate data structures from the raw data, modelling complex algorithms while ensuring their correctness, and formalising validation predicates which are close to natural language counterparts. Otherwise, we presented the points were we had to diverge from B by either using language extensions supported by ProB or by moving certain features outside of the B models, e.g. the data import. Moreover, we described a way to interact with the formal model and to build various applications on top on ProB.

In both projects ProB satisfies the respective requirements on performance and execution time. For the curriculum validation, ProB is able to detect conflicts among courses in an appropriate time, making interactive use on top of ProB possible. In the topology validation project there are no strict timing constraints. However, our B and ProB based approach is able to compete with a pre-existing validation tool written in an imperative language.

The work on both projects has helped to push the development of ProB forward by highlighting performance bottlenecks that have since been resolved. Moreover, we added support for language constructs such as tree operators.

Due to the availability of higher-order data types, B can be used almost like a functional programming language. We have used this in particular to compute derived data. In the paper we have also shown various limitations of B, and have presented some ways to overcome them (e.g., how to encode let constructs). In the future, we would like to be able to use parts of B as a proper functional programming language. In that sense, we are considering adding polymorphic operators, as present in TLA^+, to provide a simpler way to structure predicates and allow the user to define new recursive operators. Moreover, we are pursuing an approach to embed parts of the mathematical B language into the Clojure programming language using native syntax and evaluating it with ProB, an approach comparable to aRby for Alloy [21].

Acknowledgements. We would like to thank Luis-Fernando Mejia from Alstom for pushing (and funding) B and ProB into new directions. We have also grateful to ClearSy and Systerel for exercising ProB, allowing us to discover new aspects of the B language. We are grateful to Thales for funding a collaborative research project and to the Thales team, in particular Nader Nayeri, for all their help and insights. Finally we would like to thank Frank Meier, Tobias Witt, Philip Höfges and the planning teams at the Faculty of Arts and Humanities and the Faculty of Business Administration and Economics at Heinrich Heine University for their contributions to the curriculum validation project.

References

1. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs, 2nd edn. MIT Press, Cambridge (1996)
2. Abo, R., Voisin, L.: Formal implementation of data validation for railway safety-related systems with OVADO. In: Counsell, S., Núñez, M. (eds.) SEFM 2013. LNCS, vol. 8368, pp. 221–236. Springer, Heidelberg (2014)
3. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
4. Ayed, R.B., Collart-Dutilleul, S., Bon, P., Idani, A., Ledru, Y.: *B* formal validation of ERTMS/ETCS railway operating rules. In: Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 124–129. Springer, Heidelberg (2014)
5. Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project: roissy VAL. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
6. Badeau, F., Doche-Petit, M.: Formal data validation with Event-B. In: Proceedings of DS-Event-B 2012, Kyoto. CoRR, abs/1210.7039 (2012)
7. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co. Inc, Boston (2001)
8. Corne, D., Ross, P., Fang, H.-L.: Evolving timetables. In: Practical Handbook of Genetic Algorithms: Applications, vol. 1, pp. 219–276 (1995)
9. Deris, S., Omatu, S., Ohta, H.: Timetable planning using the constraint-based reasoning. Comput. Oper. Res. **27**(9), 819–840 (2000)

10. Gotlieb, C.C.: The construction of class-teacher time-tables. In: IFIP Congress, pp. 73–77 (1962). http://dblp.uni-trier.de/rec/bib/conf/ifip/Gotlieb62, http://dblp.org
11. Hayes, I.J., Jones, C.B., Nicholls, J.E.: Understanding the differences between VDM and Z. ACM SIGSOFT Softw. Eng. Notes **19**(3), 75–81 (1994)
12. Herman, D., Wand, M.: A theory of hygienic macros. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 48–62. Springer, Heidelberg (2008)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2012)
14. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
15. Lecomte, T., Burdy, L., Leuschel, M.: Formally checking large data sets in the railways. In: Proceedings of DS-Event-B 2012, Kyoto. CoRR, abs/1210.6815 (2012)
16. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From animationto data validation: the ProB constraint solver 10 years on. In: Formal Methods Applied to Complex Systems, pp. 427–446 (2014)
17. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
18. Leuschel, M., Cansell, D., Butler, M.: Validating and animating higher-order recursive functions in B. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 78–92. Springer, Heidelberg (2009)
19. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 708–723. Springer, Heidelberg (2009)
20. Leuschel, M., Schneider, D.: Towards B as a high-level constraint modelling language. In: Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 101–116. Springer, Heidelberg (2014)
21. Milicevic, A., Efrati, I., Jackson, D.: αRby—An embedding of Alloy in Ruby. In: Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 56–71. Springer, Heidelberg (2014)
22. Rudová, H., Murray, K.: University course timetabling with soft constraints. In: Burke, E.K., De Causmaecker, P. (eds.) PATAT 2002. LNCS, vol. 2740, pp. 310–328. Springer, Heidelberg (2003)
23. Schimmelpfeng, K., Helber, S.: Application of a real-world university-course timetabling model solved by integer programming. OR Spectr. **29**(4), 783–803 (2006)
24. Schneider, D., Leuschel, M., Witt, T.: Model-based problem solving for university timetable validation and improvement. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 487–495. Springer, Heidelberg (2015)