Validating Z Specifications using the ProB Animator and Model Checker

Daniel Plagge and Michael Leuschel

Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{plagge,leuschel}@cs.uni-duesseldorf.de

Abstract. We present the architecture and implementation of the ProZ tool to validate high-level Z specifications. The tool was integrated into ProB, by providing a translation of Z into B and by extending the kernel of ProB to accommodate some new syntax and data types. We describe the challenge of going from the tool friendly formalism B to the more specification-oriented formalism Z, and show how many Z specifications can be systematically translated into B. We describe the extensions, such as record types and free types, that had to be added to the kernel to support a large subset of Z. As a side-effect, we provide a way to animate and model check records in ProB. By incorporating ProZ into ProB, we have inherited many of the recent extensions developed for B, such as the integration with CSP or the animation of recursive functions. Finally, we present a successful industrial application, which makes use of this fact, and where ProZ was able to discover several errors in Z specifications containing higher-order recursive functions.

1 Introduction

Both B [1] and Z [2, 26] are formal mathematical specification notations, using the same underlying set theory and predicate calculus. Both formalisms are used in industry in a range of critical domains.

The Z notation places the emphasis on human-readability of specifications. Z specifications are often documents where ambiguities in the description of the system are avoided by supporting the prose with formal statements in Z. LaTeX packages such as *f*uzz [25] exists to support type setting and checking those documents. The formal part of a specification mainly consists of schemas which describe different aspects of a system using set theory and predicate logic. The schema calculus—a distinct feature of Z—enables system engineers to specify complex systems by combining those schemas.

B was derived from Z by Jean-Raymond Abrial (also the progenitor of Z) with the aim of enabling tool support. In the process, some aspects of Z were removed and replaced, while new features were added (notably the ASCII Abstract Machine Notation). We will discuss some of the differences later in depth. In a nutshell, B is more aimed at refinement and code generation, while Z is a

more high-level formalism aimed for specification. This is, arguably, why B has industrial strength tools, such as Atelier-B [27] and the B-toolkit [5]. Recently the PROB model checker [19] and refinement checker [20] have been added to B's list of tools. Similar tools are lacking for Z, even though there are recent efforts to provide better tool support for Z [24].

In this paper we describe the challenge of developing a Z version of PROB, capable of animating and model checking realistic Z specifications. We believe an animator and model checker is a very important ingredient for formal methods; especially if we do not formally derive code from the specification (as is common in Z [13, 12]). This fact is also increasingly being realised by industrial users of formal methods.

At the heart of our approach lies a translation of Z specifications into B, with the aim of providing an integrated tool that is capable to validate both Z and B specifications, as well as inheriting from recent refinements developed for B (such as the integration with CSP [8]). One motivation for our work comes from an industrial example, which we also describe in the paper.

2 Specifications in Z

First we give a brief introduction to the Z notation. We want to describe the structure of Z specifications, especially how this differs from specifications in B as supported by PROB. The interested reader can find a tutorial introduction to Z inside the Z reference manual [26]. A more comprehensive introduction with many examples is [16].

2.1 A brief description of Z

Usually, a specification in Z consists of informal prose together with formal statements. In a real-life applications, the prose part is at least as important as the formal part, as a specification has to be read by humans as well as computers.

Usually, one describes state machines in Z, i.e., one defines possible states as well as operations that can change the state. The Z syntax can be split into two: a notation for discrete mathematics (set theory and predicate calculus) and a notation for describing and combining *schemas*, called the schema calculus.

For illustration, we use the simple database of birthdays (Fig. 1) from [26]. The first line in the example is a declaration $[NAME, DATE]$ which simply introduces $NAME$ and $DATE$ as new basic types, without providing more information about their attributes (like generics in some programming languages). We can also see three boxes, each with a name on the upper border and a horizontal line dividing it into two parts. These boxes define the so-called schemas. Above the dividing line is the declaration part, where variables and their types are introduced, and below a list of predicates can be stated.

Without additional description, the purpose of each schema in the example is not directly apparent. We use the first schema *BirthdayBook* to define the state space of our system. *Init* defines a valid initial state and the schema *AddBirthday*

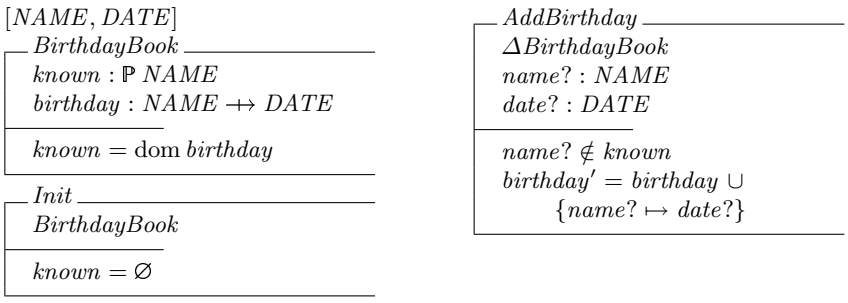

Fig. 1. The birthday book example

is the description of an operation that inserts a new name and birthday into the database.

We describe the schemas in more detail. In *BirthdayBook* we have two variables: *known* is a set of names and *birthday* is a partial function that maps names to a date. The predicate states that *known* is the domain of the partial function, i.e., the set of names that have an entry in the function. A possible state of our system consists of values for these two variables which satisfy the predicate.

The declaration part of the *Init* schema contains a reference to the *BirthdayBook* schema. This imports all variable declarations and predicates of *BirthdayBook* into *Init*. The predicate says that *known* is empty. Together with the predicate of *BirthdayBook* this implicitly states that the domain of *birthday* is empty, resulting in an empty function.

The schema defining the operation *AddBirthday* contains two variables with an appended question mark. By convention, variables with a trailing ? (resp. !) describe inputs (resp. outputs) of operations, thus *name*? and *date*? are inputs to the operation. The first line of the schema is $\Delta BirthdayBook$. This includes all declarations and predicates of *BirthdayBook*, as previously seen in *Init*. Additionally the variable declarations are included with a prime appended to their name, representing the state after the execution of the operation. The predicates are also included a second time where all occurring variables have a prime appended. To clarify this, we show the *expanded* schema:

AddBirthday

$$
\begin{array}{l}
known, known' : \mathbb{P}\ NAME \\
birthday, birthday' : NAME \pfun DATE \\
name? : NAME \\
date? : DATE \\
\hline
known = \mathrm{dom}\ birthday \land known' = \mathrm{dom}\ birthday' \\
name? \notin known \\
birthday' = birthday \cup \{name? \mapsto date?\}
\end{array}
$$

The schema thus defines the relation between the state before and after executing the operation *AddBirthday*. Accordingly the unprimed variables refer

```
MACHINE BirthdayBook
SETS NAME;DATE
VARIABLES known,birthday
INVARIANT
 known:POW(NAME) & birthday:NAME+->DATE & known=dom(birthday)
INITIALISATION known,birthday := {},{}
OPERATIONS
  AddBirthday(name,date) = PRE name:NAME & date:DATE & name/:known THEN
    birthday(name) := date || known := known \/ {name}
  END
END
```

Fig. 2. The birthday book example in B

to the state before and the primed ones to the state after the execution. The effect of *AddBirthday* is that the function *birthday* has been extended with a new entry. But, together with the predicates from *BirthdayBook*, it is (again implicitly) stated that *name*? should be added to *known*.

Instead of the schema boxes there is also a shorter equivalent syntax. E.g., *Init* can also be defined with $Init \mathrel{\widehat=} [\, BirthdayBook \mid known = \varnothing \,]$. In addition to inclusion, as seen in the example, the schema calculus of Z provides more operators to combine schemas. E.g., the conjunction of two schemas $R \mathrel{\widehat=} S1 \wedge S2$ merges their declaration part in a way that the resulting schema R has the variables of both schemas $S1$ and $S2$, and its predicate is the logical conjunction of both original predicates. The schema calculus is a very important aspect of the Z notation, because it makes Z suitable for describing large systems by handling distinct parts of it and combining them.

2.2 Some differences between Z and B

PROZ is an extension of PROB, a tool that animates specifications in B. To make use of its core functionality, we need to translate a Z specification into PROB's internal representation of a B machine. To illustrate the fundamental issues and problems, we describe some of the major differences between Z and B using our example.

Figure 2 shows the birthday book example as a B machine. Aside from the ASCII notation, one difference is the use of keywords to divide the specification into multiple sections. The `VARIABLES` section defines that `known` and `birthday` are the variables making up the state. There is an explicit initialisation and in the `OPERATIONS` section the operation `AddBirthday` is described. In a Z specification, on the other hand, the purpose of each schema must be explained in the surrounding prose.

If we look closer at the `INITIALISATION` section in the example, we see that both `known` and `birthday` are set to $\varnothing$. This is unlike the Z schema *Init* in Fig. 2, where only $known = \varnothing$ is stated and the value of *birthday* is implicitly defined. Also in the definition of the operation `AddBirthday` both variables are changed explicitly. Generally in B all changes to variables must be stated explicitly via

generalised substitutions. All other variables are not changed, whereas in Z every variable can change, as long its values satisfy the predicates of the operation.

Another noteworthy difference is the declaration of an invariant in the B machine. An invariant in B is a constraint that must hold in every state. To prove that a machine is consistent it has to be proven that the initialisation is valid and that no operation leads to an invalid state if applied to a valid state. In Z the predicate of the state's schema is also called invariant, but unlike B the operations implicitly satisfy it by including the state's schema. Errors in a B specification can lead to a violation of the invariant. A similar error in Z leads to an operation not being enabled, which in turn can lead to deadlocks.

2.3 Translating Z to B

The notation of substitutions often results in specifications that are easier to animate than higher-level Z specifications. Hence, at the heart of PROZ is a systematic translation of Z schemas into B machines.

Figure 3 contains such a B translation of the birthday book Z specification, as computed by our tool (to make the specification more readable we use Z style identifiers, i.e., ending with $'$, ? or !, even though strictly speaking this is not valid B syntax). As can bee seen, we have identified that the variables *birthday* and *known* form part of the state, their types are declared in the invariant. The initialisation part is a translation of the expanded *Init* schema. One operation *AddBirthday* with two arguments *date*? and *name*? has been identified, a translation of the expanded *AddBirthday* schema can be found in the WHERE clause of its ANY statement. There are also several references to a constant *maxentries*. We added it and a constraint $\#\,known \leq maxentries$ to demonstrate the handling of axiomatic definitions (cf. Section 3.1).

The B machine from Figure 3 can be fed directly into PROB, for animation and model checking. However, Z has also two data types, free types and schema types, that have no counterpart in B. This means that some aspects of Z cannot be effectively translated into B machines, and require extensions of PROB. In the next section we present the overall architecture of our approach, as well as a formal explanation of how to derive a B model from a Z specification.

3 Architecture and the PROZ compiler

In the previous section we have examined the basic ingredients of Z specifications, and have highlighted why Z specifications are inherently more difficult to animate and model check than B specifications. In this and the next section we explain how we have overcome those issues; in particular:

- How to analyse the various schemas of Z specification, identifying the state of a Z specification, the state-changing operations and the basic user-defined data types (cf. Section 3.1).
- How to deal with the fact that Z specifications do not specify all changes to variables explicitly.

```
MACHINE z_translation
SETS NAME;DATE
CONSTANTS maxentries
PROPERTIES
    (maxentries:INTEGER) & (maxentries>=5)
VARIABLES birthday, known
INVARIANT
    (birthday:POW(NAME*DATE)) & (known:POW(NAME))
INITIALISATION
  ANY birthday', known'
    WHERE
        (known':POW(NAME)) & (birthday':(NAME+->DATE))
      & (known'=dom(birthday')) & (card(known')<=maxentries)
      & (known'={})
    THEN
      birthday, known := birthday', known'
  END
OPERATIONS
  AddBirthday(date?, name?) =
    PRE (name?:NAME)
      & (date?:DATE)
      THEN
        ANY birthday', known'
          WHERE
              (known:POW(NAME)) & (birthday:(NAME+->DATE))
            & (known=dom(birthday)) & (card(known)<=maxentries)
            & (known':POW(NAME)) & (birthday':(NAME+->DATE))
            & (known'=dom(birthday')) & (card(known')<=maxentries)
            & (name?/:known) & (birthday'=(birthday\/{(name?,date?)}))
          THEN
            birthday, known := birthday', known'
        END
    END
END
```

Fig. 3. The translated birthday book example

– How to deal with the new data types provided by Z.
– How to deal with new operators and constructs.

Overall Architecture PROZ is an extension of PROB that supports Z specifications which can be parsed by the *f*UZZ typechecker. Those specifications are given as a LaTeX file. When the user loads a specification into PROZ, the following steps are performed (see also Figure 4):

1. The specification is typechecked with *f*UZZ. *f*UZZ writes the formal content of the specification into a file which then is parsed by PROZ.
2. The different components of the specification (definition of constants, state, initialisation and operations) are identified.

3. All schemas are expanded and normalised, i.e., all schema inclusions are resolved and the type declarations of variables are strictly separated from constraints on their values.
4. Several rules to simplify expressions are applied.
5. PROZ then translates the specification to an internal representation of a B machine (with some small extensions, which are discussed later in the paper).
6. After the translation process PROB treats the specification the same way as other B machines are treated (with some extensions having been added to the PROB kernel).

Most of the expressions in Z have a direct counterpart in B, for those the translation in point 4 is just a conversion from one syntax into another. Some cases where there is more logic need in the translation process or where we extended the PROB interpreter are presented in Section 3.3. The support of two Z data types as discussed in the next section affects the translation process and requires extensions to the kernel as well.

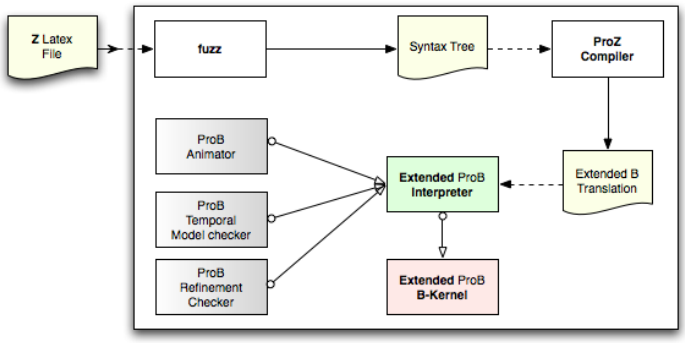

Fig. 4. Overview of PROZ Architecture

3.1 Identifying components of the specification

As we have seen in the previous sections, the purpose of the schemas in a specification is not stated formally. But to interpret a given specification for animation and model checking, we must identify which schemas describe the state space, the initialisation and the operations. To be able to do so, we require that the specification satisfies some rules:

- There must be a schema called *Init* for initialisation.
- *Init* includes exactly one other schema. The included schema will be taken as the description of the state space.
- A schema with all variables of the state and their primed versions, that is not included by any other schema, will be used as an operation.

The rules can be applied to the birthday book example in Fig. 1: There is a schema *Init* which includes *BirthdayBook*. Thus *Init* is the initialisation of the state which consists of *BirthdayBook*'s variables *known* and *birthday*. Expanding *AddBirthday* shows that it has all variables of the state and also the

primed versions *known′* and *birthday′*. It is not included by any other schema. Thus PROZ would identify *AddBirthday* as an operation.

In the next two paragraphs we present two other components of a specification that are used by PROZ and explain how they relate to existing features of PROB.

Invariant As seen in the comparison in Sect. 2.2, the B invariant has no direct counterpart in Z. But it can be useful to search for states that violate a certain property by model checking. To make this feature available for Z specifications, PROZ looks for a schema named *Invariant*. If such an invariant is given, its predicate is checked for every visited state in an animation or in model checking. The predicate is then used analogously to the invariant in B.

Axiomatic definitions In our short introduction to Z we did not describe how global constants can be introduced in Z by *axiomatic definitions*. Like schemas, axiomatic definitions consist also of a declaration and a predicate part, but their declared variables can be used throughout the specification without a schema inclusion. E.g., we can define a constant *maxentries* which value is at least 5 with the axiomatic definition

$$\begin{array}{|l} maxentries : \mathbb{Z} \\ \hline maxentries \geq 5 \end{array}$$

We interpret axiomatic definitions analogously to how PROB interprets CONSTANTS and PROPERTIES in a B machine: The very first step of an animation or model checking—before the initialisation of the state variables—consists in finding values for the constants which satisfy the predicates of the axiomatic definitions. After this step the predicates of the axiomatic definitions can be ignored. To illustrate how the axiomatic definitions are handled, we added the definition above to the birthday book example and appended the predicate $\# known \leq maxentries$ to the schema *BirthdayBook* before translating the specification to the result in Figure 3.

3.2 Translating initialisation and operations from Z to B

The *initialisation* schema *Init* consists of the declaration of all state variables and a predicate I. We annotate T_v as the type of variable v.

$$\begin{array}{|l} \textit{Init} \\ \hline x_1 : T_{x_1}; \ \ldots; x_n : T_{x_n} \\ \hline I \end{array}$$

In B, the initialisation is a generalised substitution to all variables of the abstract machine. We can state "choose any values that satisfy I" with an ANY

statement:

$$
\begin{array}{l}
\text{ANY } x_1', \ldots, x_n' \\
\quad \text{WHERE } x_1' \in T_{x_1} \wedge \ldots \wedge x_n' \in T_{x_n} \wedge \\
\qquad I' \\
\quad \text{THEN } x_1, \ldots, x_n := x_1', \ldots, x_n' \\
\text{END}
\end{array}
$$

Beside the predicate I, the WHERE clause of the ANY contains the type declaration of the variables. The types T_v and the predicate I are translated from Z to B syntax. Most of the types, predicates and expressions in Z have a direct counterpart in B and can be translated directly. In section 3.3 we show how we extended the B interpreter to support other constructs.

An *operation* schema Op declares in addition to the state variables $x_1, \ldots, x_n$ their primed counterparts $x_1', \ldots, x_n'$ and variables for input $i_1?, \ldots, i_k?$ and output $o_1!, \ldots, o_l!$. The predicate P describes the effect of the operation.

$$
\begin{array}{|l}
\hline
Op \\
\hline
x_1 : T_{x_1};\ \ldots;\ x_n : T_{x_n} \\
x_1' : T_{x_1};\ \ldots;\ x_n' : T_{x_n} \\
i_1? : T_{i_1};\ \ldots;\ i_k? : T_{i_k} \\
o_1! : T_{o_1};\ \ldots;\ o_l! : T_{o_l} \\
\hline
P \\
\hline
\end{array}
$$

PROZ translates such a schema to a B operation of the form

$$
\begin{array}{l}
o_1^*!, \ldots, o_l^*! \leftarrow Op(i_1?, \ldots, i_k?) = \\
\quad \text{PRE } i_1 \in T_{i_1} \wedge \ldots \wedge i_k \in T_{i_k} \text{ THEN} \\
\qquad \text{ANY } x_1', \ldots, x_n', o_1!, \ldots, o_l! \\
\qquad\quad \text{WHERE } x_1' \in T_{x_1} \wedge \ldots \wedge x_n' \in T_{x_n} \wedge o_1! \in T_{o_1} \wedge \ldots \wedge o_l! \in T_{o_l} \wedge \\
\qquad\qquad P \\
\qquad\quad \text{THEN } x_1, \ldots, x_n, o_1^*!, \ldots, o_l^*! := x_1', \ldots, x_n', o_1!, \ldots, o_l! \\
\qquad \text{END} \\
\quad \text{END} \\
\text{END}
\end{array}
$$

Like in the initialisation the central part of the operation is an ANY statement with the predicate P, but additionally we have to consider the possible result values. The surrounding PRE statement is just for declaring the types of the operation's arguments.

Often operations change only a subset of the state variables. PROZ checks if terms like $x = x'$ occur in the predicate P. If such a term is found, we know that x does not change and so we can remove the substitution $x := x'$. Also we can replace all occurrences of x' by x in P. Then x' is not used anymore in the statement and can be removed. If parts of the state or the complete state are not modified by an operation, the expression $\theta S = \theta S'$ is often used, where S is

a schema containing all variables that should not change. ΞS is an abbreviation for including ΔS and stating $\theta S = \theta S'$. Those expressions are transformed into $s_1 = s_1' \wedge \ldots \wedge s_n = s_m'$ with $s_1, \ldots, s_m$ as the variables of S. This way the simplification of the ANY statement is also working with θ-expressions.

3.3 New constructs and operators

Some constructs of Z's mathematical language do not have a direct counterpart in B, and below we show how we have treated those.

Translation of Comprehension Sets A comprehension set has the form $\{ \mathit{Decl} \mid \mathit{Pred} \}$ and specifies a set with a declaration of variables and a predicate. E.g., the expression $\{ i : \mathbb{N} \mid i \geq 5 \}$ is the set of all numbers greater or equal to 5. This kind of comprehension sets is also supported by B, but Z has an extended syntax of the form $\{ \mathit{Decl} \mid \mathit{Pred} \bullet \mathit{Expr} \}$. E.g., the set $\{ i : \mathbb{N} \mid i \geq 5 \bullet i * i \}$ is the set of all square numbers greater or equal to 25. We translate such comprehension sets as follows. Let T be the type of *Expr*, then we express $\{ \mathit{Decl} \mid \mathit{Pred} \bullet \mathit{Expr} \}$ by $\{ v : T \mid (\exists \mathit{Decl} \mid v = \mathit{Expr} \wedge \mathit{Pred}) \}$ and translate this into B.

Extensions to the B Interpreter The following expressions are not easily translatable to B (or would entail a considerable efficiency penalty), and hence extensions were made to the PROB interpreter to support an extended B syntax:

- We added an **if**-expression to the standard B syntax. While B contains a substitution IF − THEN − ELSE, it can not be used as an expression that yields a value. The **if** expression of Z resembles to the ternary `?:` operator known in C or Java.
- The **let** in Z can be used as an expression or as a predicate. Both can not be stated directly in B, which again only has the **LET** as a substitution.
- The operations $\upharpoonright$ (extraction) and $\upharpoonright$ (filter) on sequences are defined with the function *squash*. We added the *squash* function to the interpreter.
- We added the definite description quantifier μ.

4 New Types

To deal with the Z specifications we have seen so far, it was sufficient to translate Z to B, possibly with some some syntactic extensions. There are, however, two important features of Z which cannot be effectively dealt with in that way: Z's schema and free types. Supporting those features in an effective manner requires a fundamental addition to the core datatypes of the PROB kernel.

Overview of the PROB*-kernel* The PROB kernel is responsible for storing and finding values for the values of the variables in a specification. In order to avoid naive enumeration of possible values, the PROB kernel is written in Prolog works in multiple phases (controlled by Prolog's `when` co-routining mechanism). In the first phase, only deterministic propagations are performed (e.g., the predicate

$x = 1$ will be evaluated but the predicates $x \in \mathbb{N}$ will suspend until they either become deterministic or until the second phase starts). In the second phase, a restricted class of non-deterministic enumerations will be performed. For example, the predicate $x \in \{a, b\}$ will suspend during the first phase but will lead to two solutions $x = a$ and $x = b$ during the second phase. In the final phase, *all* variables, parameters and constants that are still undetermined (or partially determined) are enumerated.

New Data types Adding a new basic data type to the kernel requires the extension of four Prolog predicates: `equal_object` to check two objects for equality, `not_equal_object` to check two objects for disequality, one predicate to type check an object and one predicate to enumerate all possible values of an object given its type. So far the kernel supported basic user-defined types (defined in B's SET clause), integers, pairs and sets (relations are represented as sets of pairs). Below, we present two new data types, schema types and free types, which are needed for Z.

4.1 Schema Types

In Z each schema defines a new data type, a *schema type* which resembles record types known from other languages. Basically, a record data value $rec(f)$ consists of a list $f = [n_1/v_1, \ldots, n_k/v_k]$ of field names n_i along with values v_i for each field. We require that all field names are sorted alphabetically. Two record values are identical iff they have the exact same field names and all field values are identical. In the kernel this gives rise to two new inference rules:

$$\frac{x_1 = y_1 \quad \ldots \quad x_k = y_k \quad n_1 < n_2 < \ldots < n_k}{rec([n_1/x_1, \ldots, n_k/x_k]) = rec([n_1/y_1, \ldots, n_k/y_k])}$$

$$\frac{x_i \neq y \quad 0 \leq i \leq k \quad n_1 < n_2 < \ldots < n_k}{rec([n_1/x_1, \ldots, n_i/x_i, \ldots, n_k/x_k]) \neq rec([n_1/x_1, \ldots, n_i/y, \ldots, n_k/x_k])}$$

The type of a record contains the name of the fields and the types of each field. This gives rise to two new inference rules for type inference and enumeration, where we use the k-ary type constructor *Record* for records with k-fields:

$$\frac{x_1 : \tau_1 \quad \ldots \quad x_k : \tau_k \quad n_1 < n_2 < \ldots < n_k}{rec([n_1/x_1, \ldots, n_k/x_k]) : Record(n_1/\tau_1, \ldots, n_k/\tau_k)}$$

$$\frac{x_1 \in enum(\tau_1) \quad \ldots \quad x_k \in enum(\tau_k) \quad n_1 < n_2 < \ldots < n_k}{rec([n_1/x_1, \ldots, n_k/x_k]) \in enum(Record(n_1/\tau_1, \ldots, n_k/\tau_k))}$$

Classical B does not have a record type, but a record type extension and syntax has been introduced by the tool Atelier-B [27].[1] In extending the kernel, ProB now also supports those records in B.

[1] See also [11] for a theoretical foundation of records.

Note that in Z, possible instances of a schema type (the *bindings*) can be further constrained by the predicates of the schema. E.g. the schema

$$ExampleRecord \mathrel{\widehat=} [\, x, y : \mathbb{Z} \mid x < y \,]$$

can be used as a record with the constraint $x < y$. The kernel does not support this directly, instead an unconstrained record $[\, x, y : \mathbb{Z} \,]$ can be used. We show how the constraints can be preserved in the translation process by *normalisation*.

PROZ normalises all schemas of a specification, i.e. it strictly separates type information and additional predicates on the instances. E.g. the normalised form of the schema $[\, x : \{1, 2, 3\} \,]$ is $[x : \mathbb{Z} \mid x \in \{1, 2, 3\}]$. Given a normalised schema $A \mathrel{\widehat=} [\, Decl \mid Pred \,]$, we define $A^* \mathrel{\widehat=} [\, Decl \,]$ as the schema with just the type information and without any additional constraints. If A is used as a type for a variable v, in the normalisation process it is split into the type A^* and the additional constraint $v \in \{\, Decl \mid Pred \bullet \theta A\}$. Because the type A^* does not have further constraints, it's supported by the kernel. The constraint had been made explicit by the normalisation and can be translated to B. The used θ-operator creates an instance of type A. We can translate it directly to a record constructor.

4.2 Free Types

Another feature of the Z notation is the definition of *free types*. E.g.,

$$T ::= empty \mid value\langle\!\langle \{1, 2, 3\} \rangle\!\rangle$$

defines a new data type T with a constant value *empty* and a constructor function *value* which maps values from $\{1, 2, 3\}$ to T. Contrary to schema types, free types can also be recursive, as in the following example, defining a binary tree with integers:

$$BinTree ::= empty \mid leaf\langle\!\langle \mathbb{Z} \rangle\!\rangle \mid node\langle\!\langle BinTree \times \mathbb{Z} \times BinTree \rangle\!\rangle$$

In Z, free types are only syntactic sugar and can also be expressed with axiomatic definitions and basic types. But for the purpose of animating the specification it is essential for efficiency to implement this type directly.

There is no counterpart for free types in B, so we extended the PROB core. The representation of data values and the inference rules for equality and typing are similar to record types; one just needs to also store the constructor used (e.g., in the case of T above we need to know whether we are in the case *empty* or in the case *value*). Two free type data values are thus identical iff they have the same constructor and if the values for that constructor are identical.

Free type definitions can be made recursive, so the implementation of enumeration must prevent the generation of infinitely many values. We solved this by introducing a maximum recursion depth when enumerating free types. The maximum is adjustable by the user. The introduction of a maximum recursion depth

has the effect that the model checker might not find all possible solutions (similarly to integer variables whose enumeration is restricted to MININT..MAXINT).

The PROB interpreter is extended by a constructor *FreeConstructor* for creating instance values of free types. The arguments are the free type, the case (*empty* or *value* in the T example) and the tuple containing the arguments to the constructor. Also there is the inverse of the constructor *FreeDestructor*, which takes a free type instance and returns the type, the case and the tuple of arguments. Finally, we have a predicate *FreeCase* that takes the identifier of a case and a free type instance as arguments and evaluates to true if the free type value has the given case.

The kernel itself does not support constraints on the values of a constructor. In the T example above the type of the constructor *value* is $\mathbb{Z}$ but the domain constrained to $\{1, 2, 3\}$. Like with the schema types, the constraints have to be handled separately in the translation. This is done by normalisation as follows.

Given a free type F of the form

$$F ::= c_1 \mid \ldots \mid c_n \mid d_1 \langle\!\langle S_1 \rangle\!\rangle \mid \ldots \mid d_m \langle\!\langle S_m \rangle\!\rangle$$

we define the type F^* which has just the type information of F without other constraints, where T_i is the underlying type of S_i (e.g. $S_i = \{1, 2, 3\} \Rightarrow T_i = \mathbb{Z}$):

$$F^* ::= c_1 \mid \ldots \mid c_n \mid d_1^* \langle\!\langle T_1 \rangle\!\rangle \mid \ldots \mid d_m^* \langle\!\langle T_m \rangle\!\rangle$$

Then we convert F and the constructors d_i, $1 \leq i \leq m$ to

$$F == \{ x : F^* \mid x \in \operatorname{ran} d_1^* \Rightarrow d_1^{*\sim}(x) \in S_1 \wedge \ldots \wedge x \in \operatorname{ran} d_m^* \Rightarrow d_m^{*\sim}(x) \in S_m \}$$
$$d_i == (\lambda x : T_i \mid x \in S_i \bullet d_i^*(x))$$

The schema normalisation transforms a variable v of type F to a variable of type F^* and adds the constraint $v \in F$.

The transformed example would be

$$T^* ::= \mathit{empty} \mid \mathit{value}^* \langle\!\langle \mathbb{Z} \rangle\!\rangle$$
$$T == \{ x : T^* \mid x \in \operatorname{ran} \mathit{value}^* \Rightarrow \mathit{value}^{*\sim}(x) \in \{1, 2, 3\} \}$$
$$\mathit{value} == (\lambda x : \mathbb{Z} \mid x \in \{1, 2, 3\} \bullet \mathit{value}^*(x))$$

Finally, expressions of the form $x \in \operatorname{ran} d_i^*$ are translated to the predicate $\mathit{FreeCase}(F, d_i^*, x)$ and constructor calls of the form $d_i^*(x)$ (resp. the inverse $d_i^{*\sim}(y)$) are translated to $\mathit{FreeConstructor}(F^*, d_i^*, x)$ (resp. $\mathit{FreeDestructor}(F^*, d_i^*, y)$). The result can then be dealt with by the extended PROB interpreter and kernel.

5 Case study

The case study was inspired by a real industrial example. The specifications are very high level and, using the guidelines from [13], were not destined to be

refined into code. These Z specifications thus[2] provide a particular challenge for our tool. Below we present two sub-components of the system, the challenges in animating and validating them, as well as an indication on the errors located by our tool.

5.1 Route calculation

The *route calculation* component is a key component of the overall system, containing several intricate algorithmic aspects. It is important to ascertain the correctness of the algorithms (e.g., before proceeding with an implementation).

This system component calculates routes through a given geometry. The geometry (mainly places and roads) is stored in the system state. The main part of the specification consists of the definition of a function that takes a route as input. The input route is a sequence that starts and ends with a place and between both is a list of places or roads. The result of the function is the *expansion* of the route, i.e. the sequence of all places that lie between the first and the last place. E.g., in the given geometry in figure 5 the expansion of $\langle Bicester, A34, M4, Swindon\rangle$ is $\langle Bicester, Oxford, NewburyRoundabout, Swindon\rangle$. For sake of simplicity we ignore below the connections which are not roads. The

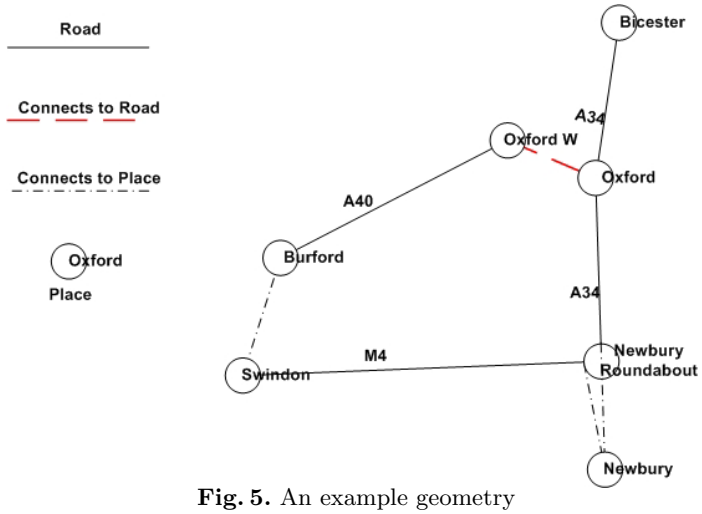

Fig. 5. An example geometry

expansion function is constructed by combining several other functions, which do not work directly on the input route. Instead a record is created that contains the original route, information about which part has already been processed, the result so far, and a set of errors. An error could be "no connection found", for example. A recursive function (Fig. 6) expands every single element in the route

[2] Some of the features of B, such as generalised union, are rarely used in formal refinements as the existing B provers do not support them very well. It is our experience that formal B specifications that are refined to code are easier to animate than more liberal specifications.

until the complete route is expanded or an error is found. In total the specification consists of 8 function definitions which are combined to calculate the result. Most of these functions are defined by comprehension sets.

Due to the complexity of the defined functions, it was not feasible to enumerate them (i.e., to store all possible inputs and outputs). Fortunately, PROB [22] has the ability to compile these kind of definitions into *symbolic* closures, which are evaluated and expanded on demand. For example, given a set comprehension $S = \{x \mid x \in \mathbb{N} \Rightarrow P\}$ and the condition $y \in S$ the Kernel will "only" check that P holds for $x = y$ and *not* compute the entire set S. A similar situation arises for lambda abstractions. Take for example, $f = \lambda\, x.(x \in \mathbb{N} \mid E)$. In that case, to evaluate $f(y)$ the Kernel "only" evaluates E with y substituted for x and *not* the entire function f.[3] The kernel, even supports recursive function definitions, such as the one presented in Fig. 6.

Storing comprehensions sets and λ-expressions symbolically was an essential feature to allow the animation of the specification. By integrating PROZ into PROB we inherit this feature, which allows us to validate this specification.

ExpandElems

$ExpandElem expandElems : Expansion \pfun Expansion$

$$
\begin{array}{l}
expandElems = \{\, \Delta Expansion \mid \\
\quad \theta Expansion' = \mathbf{if}\ error \neq \varnothing \lor currentElem \notin \mathrm{dom}\, proposedRoute \\
\qquad \mathbf{then}\ \theta Expansion \\
\qquad \mathbf{else}\ expandElems(expandElem(\theta Expansion)) \bullet \\
\quad \theta Expansion \mapsto \theta Expansion' \,\}
\end{array}
$$

Fig. 6. Example: The recursive definition of the function *expandElems*.

To make an animation possible, the system was initialised with test data that describes a map with six cities. Running the animation the user can simply click on the *AddElement* operations to construct an input and sees immediately the result. Figure 7 shows a screenshot of the animator after entering the route "Newbury → A34 → Bicester". In the middle the list of enabled operations can be seen where the *Expand* operation contains the solution (which is truncated in the screenshot).

The animation of the specification quickly exhibited one error in the specification. For each road a sequence of places to which it connects is stored in the geometry. When a route contains a road, the entry and exit points are calculated and the section between both is appended to the result. But under certain circumstances the section was appended in the wrong direction so that the route "Newbury → A34 → Bicester" was calculated to "Newbury → Bicester → Oxford → Newbury → Bicester" instead of the much simpler correct solution "Newbury → Oxford → Bicester".

[3] Some expressions, however, will require the computation of the entire function (e.g., $dom(f) \subseteq SetA$). In those circumstances the kernel converts the symbolic form into explicit form.

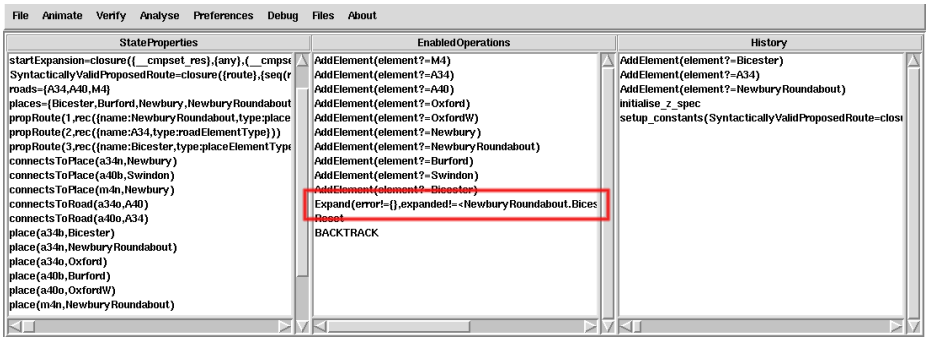

Fig. 7. Animation of the route calculation

Figure 8 shows the function containing the error. The result of the expression (in the third **let** expression)

$$(\textbf{if } entry > exit \textbf{ then } exit \mathinner{..} entry \textbf{ else } entry \mathinner{..} exit) \upharpoonright roadPlaces(r)$$

are all places on the road r that are between *entry* and *exit*. If $roadPlaces(r)$ is the sequence $\langle a, b, c, d, e\rangle$, *entry* is 4 and *exit* is 2, than the result is $\langle b, c, d\rangle$. Although the case $entry > exit$ is covered explicitly in the specification, it has been forgotten to reverse the resulting sequence to $\langle d, c, b\rangle$.

ExpandRoad

$FindConnections$

$expandRoad : Expansion \pfun Expansion$

$$
\begin{array}{l}
expandRoad = \{r : ElementName; ExpansionOp \mid \\
\quad r \in RoadName \land \\
\quad (proposedRoute(currentElem)).type = roadElementType \land \\
\quad (proposedRoute(currentElem)).name = r \land \\
\quad (\textbf{let } entries == findConnections(r, proposedRoute(currentElem - 1)); \\
\qquad exits == findConnections(r, proposedRoute(currentElem + 1)) \bullet \\
\qquad ((entries = \emptyset \lor exits = \emptyset) \\
\qquad\quad \land error' = \{noConnection\} \land expandedRoute' = \langle\rangle) \lor \\
\qquad (\textbf{let } entry == min(entries); exit == min(exits) \bullet \\
\qquad\quad (\textbf{let } placesToAdd == (\textbf{if } entry > exit \textbf{ then } exit \mathinner{..} entry \\
\qquad\qquad \textbf{else } entry \mathinner{..} exit) \upharpoonright roadPlaces(r) \fatsemi place \bullet \\
\qquad\quad expandedRoute' = \textbf{if } last\ expandedRoute = head\ placesToAdd \\
\qquad\qquad \textbf{then } expandedRoute \frown tail\ placesToAdd \\
\qquad\quad \textbf{else } expandedRoute \frown placesToAdd \land \\
\qquad\quad error' = \emptyset))) \bullet \\
\quad \theta Expansion \mapsto \theta Expansion'\}
\end{array}
$$

Fig. 8. The definition of the function *expandRoad* containing an error.

For this application we did not yet use the model checking facilities of ProZ, because we have no further properties about the result of the algorithm (and hence no way to automatically check the correctness of the result). But the

animator alone gives the user a powerful tool to get more insight in the behaviour of a specification, as the quick detection of errors showed.

5.2 Network protocol

A second important component of the overall system implements access control to a shared resource, employing a simple network protocol. A number of workstations are connected via a network and share are critical resource. Whenever a workstation wants to access the resource it has to send a request to the other workstations. The protocol should assure that only one workstation can be in the critical section at the same time.

The specification distinguishes between the state and behaviour of the workstations and the the state and behaviour of the underlying middleware.

The specification of the middleware is the description of an existing system. Its state space consists of a sent and received buffer for each workstation. Messages can be added to a sent buffer, transferred between workstations and removed from a received buffer to deliver it to the workstation.

The specification of the workstation defines their current states (*idle*, *waiting*, *editing* or *failed*) and their operations. They can send requests to the other workstations, read their responses, read other requests and send responses.

The components from both parts of the definitions are combined by using the schema calculus. Especially the pipe operator ($\gg$) was used to connect operations, where the result of one operation serves as the input for another operation. E.g. when a workstation sends a request, the operation describing the workstation behaviour outputs a message that is taken by a middleware operation as input:

$$RequestOK \mathrel{\widehat=} RequestWorkstationOK \gg AcceptMsgMiddleware$$

A screenshot of the animator is shown in Fig. 9. On the left side the current state is displayed. It can be seen that workstation 1 is waiting for a response of workstation 2, workstation 2 is in editing mode and workstation 3 is idle. Also a message is still in the sent buffer of workstation 1.

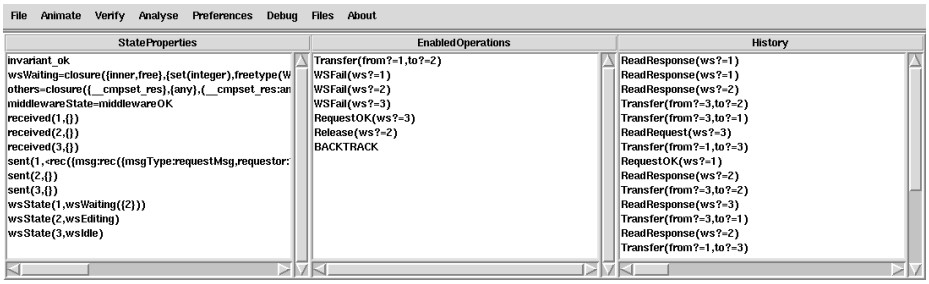

Fig. 9. Animation of the network protocol

Free types are used in the specification for distinguishing the different modes of a workstation. *wsIdle* and *wsEditing* are constants of the free type, whereas

wsWaiting is a constructor, e.g. $wsWaiting(\{1,3\})$ refers to the state "waiting for workstations 1 and 3".

First we used the model checker to find deadlocks in the protocol. It found a deadlock that was caused by an error in the specification. It was possible that a workstation could ignore a rejected request. The same error caused a situation where more than one workstation was in the critical section.

We added an *Invariant* schema to the specification to check automatically if more than one workstation is in the editing mode (*wsState* is a function defined in the schema *Workstations* that maps each workstation to its mode, and the operator $\rhd$ is used to restrict it to all entries which map to *wsEditing*):

$$Invariant \mathrel{\widehat=} [\, Workstations \mid \#(wsState \rhd \{wsEditing\}) \leq 1 \,]$$

The model checker was able to find states where the invariant was violated. Another error was found: Every response to a request was treated as if it was a grant, even rejections.

The model checker was not able to do an exhaustive search of the state space because the message buffers in the model are not limited.

6 Discussion, Related and Future Work

Limitations Z is a very large and extensive formal method, with many features and extension. While we provide a tool that can animate a considerable subset of Z, some of Z's features are obviously not yet supported:

- Bags (multisets) are not supported.
- Some expressions like disjoint and partition are not yet implemented.
- Generic definitions cannot be used in a specification yet. We plan to support them by determining with wich types a generic definition is used, and then creating for each such type a separate axiomatic definition.

Related and Future Work On the theoretical side, there are several works discussing the relationship and translations between Z and B [9, 10] or weakest precondition semantics [4].

On the practical side, several animators for Z exist, such as [29], which presents an animator for Z implemented in Mercury, as well as the Possum animation tool [14]. Another animator for Z is ZANS [17]. It has been developed in C++ and unlike PROB only supports deterministic operations (called explicit in [17]). The more recent Jaza tool by Mark Utting [28] looks very promising. There has also been a recent push [24] to provide more tool support for Z. However, to our knowledge, no existing Z animator can deal with the recursive higher-order functions present in our case study.

The most closely related work on the B side is [6, 3, 18], which uses a special purpose constraint solver over sets (CLPS) to animate B and Z specifications using the so-called BZ-Testing-Tools. However, the focus of these tools is test-case generation and not verification, and the subset of B that is supported is comparatively smaller (e.g., no set comprehensions or lambda abstractions, constants and properties nor multiple machines are supported).

Another very popular tool for validating specifications and models is Alloy [15], which makes use SAT solvers (rather than constraint solving). However, the specification language of Alloy is first-order and thus cannot be applied "out of the box" to our motivating industrial example.

Conclusion In this paper we presented PROZ, a tool for animating and model checking Z specifications. We pursued an approach to translate Z specifications to B, reusing the existing PROB toolset as much as possible. Some extensions to the PROB core were required (e.g., for free types and schema types), after which we have obtained an integrated tool that is now capable to animate and validate Z and B specifications. In principle our tool could now validate combined B/Z specifications,[4] and as a side effect we have added support for B specifications with records. By integrating PROZ with PROB our tool has also inherited from the recent developments and improvements originally devised for B, such as visualisation of large state spaces [23], integration with CSP [8], symmetry reduction [21], and symbolic validation of recursive functions [22].

Our tool was successfully applied to examples which were based on industrial specifications and also revealed several errors. Especially PROZ's ability to store comprehensions sets symbolically was essential to make the animations of those specifications possible.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial, S. A. Schuman, and B. Meyer. Specification language. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs: An Advanced Course*, pages 343–410. Cambridge University Press, 1980.
3. F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002. Technical Report, INRIA.
4. J. W. Ana Cavalcanti. A weakest precondition semantics for z. *The Computer Journal*, 41(1):1–15, 1998.
5. U. B-Core (UK) Limited, Oxon. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.
6. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P.Katoen and P.Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.
7. J. P. Bowen. *Formal Specification and Documentation using Z*. International Thomson Computer Press, 1996.
8. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.
9. A. Diller and R. Docherty. Z and abstract machine notation: A comparison. In *Z User Workshop*, pages 250–263, 1994.

[4] It is not clear to us whether this has any practical benefit

10. S. Dunne. Understanding object-z operations as generalised substitutions. In E. A. Boiten, J. Derrick, and G. Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2004.
11. N. Evans and M. Butler. A proposal for records in event-b. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2006.
12. A. Hall. Correctness by construction: Integrating formality into a commercial development process. In L.-H. Eriksson and P. A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2002.
13. J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.
14. D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. *Automated Software Engineering*, 00:302, 1998.
15. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, 2002.
16. J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.
17. X. Jia. An approach to animating Z specifications. Available at `http://venus.cs.depaul.edu/fm/zans.html`.
18. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.
19. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
20. M. Leuschel and M. Butler. Automatic refinement checking for B. In K.-K. Lau and R. Banach, editors, *Proceedings ICFEM'05*, LNCS 3785, pages 345–359. Springer-Verlag, 2005.
21. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for b by permutation flooding. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
22. M. Leuschel, D. Cansell, and M. Butler. Validating and animating higher-order recursive functions in B. Submitted; preliminary version presented at Dagstuhl Seminar 06191 Rigorous Methods for Software Construction and Analysis, 2006.
23. M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.
24. P. Malik and M. Utting. CZT: A framework for Z tools. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *Proceedings ZB'2005*, volume 3455 of *Lecture Notes in Computer Science*, pages 65–84. Springer, 2005.
25. J. M. Spivey. *The Fuzz Manual*. Available at `http://spivey.oriel.ox.ac.uk/mike/fuzz`.
26. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
27. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com/index_uk.html`.
28. M. Utting. Data structures for Z testing tools. In *FM-TOOLS 2000 conference*, July 2000. in TR 2000-07, Information Faculty, University of Ulm.
29. M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the 21st Australasian Computer Science Conference*, pages 279–294, Perth, Australia, February 1998.