Int J Softw Tools Technol Transfer (2008) 10:185–203
DOI 10.1007/s10009-007-0063-9

REGULAR CONTRIBUTION

ProB: an automated analysis toolset for the B method

Michael Leuschel · Michael Butler

Published online: 8 January 2008

Abstract We present ProB, a validation toolset for the B method. ProB's automated animation facilities allow users to gain confidence in their specifications. ProB also contains a model checker and a refinement checker, both of which can be used to detect various errors in B specifications. We describe the underlying methodology of ProB, and present the important aspects of the implementation. We also present empirical evaluations as well as several case studies, highlighting that ProB enables users to uncover errors that are not easily discovered by existing tools.

1 Introduction

The B method, originally devised by J.-R. Abrial [2], is a theory and methodology for formal development of computer systems. It is used by industries in a range of critical domains, most notably railway control. The B Method is intended to support a verification by construction approach to system development. This involves a formal framework in which models are constructed at multiple levels of abstraction and related by *refinement*. The highest levels of abstraction are used to express the required behaviour in terms of the problem domain. The closer it is to the problem domain, the easier it is to validate against the informal requirements, i.e., ensure that it is the right specification. The lowest level of abstraction corresponds to an implementation.

Models at any abstraction level are represented in B as *machines*. A machine essentially consists of state variables, a state invariant and operations on the variables. The variables of a machine are typed using set theoretic constructs such as sets, relations and functions. Typically these are constructed from basic types such as integers and given types from the problem domain (e.g., *Name, User, Session*, etc.). The invariant of a machine is specified using predicate logic. Operations of a machine are specified as *generalised substitutions*, which allow deterministic and nondeterministic state transitions to be specified. There are two main proof activities in B: *consistency checking*, which is used to show that the operations of a machine preserve the invariant, and *refinement checking*, which is used to show that one machine is a valid refinement of another. These activities are supported by industrial strength tools, such as Atelier-B [21] and the B-toolkit [8]. Significant recent developments are Event-B [1], an evolution of B to support reactive system development, and the Rodin platform [4], an open tool platform to support Event-B. In this paper, we focus on classical B, though ProB is being extended and ported to the Rodin platform to support Event-B.

In this paper we give an overview of the ProB tool which we developed to complement the existing tools for the B Method. ProB is an animation and model checking tool for the B method. ProB's animation facilities allow users to gain confidence in their specifications. ProB supports automated consistency checking which can be used to detect various errors in B specifications. ProB also supports automated refinement checking between B specifications. We describe

This research was carried out as part of the EU research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

M. Leuschel · M. Butler (✉)
School of Electronics and Computer Science,
University of Southampton, Highfield,
Southampton SO17 1BJ, UK
e-mail: mjb@ecs.soton.ac.uk; mal@ecs.soton.ac.uk

M. Leuschel
Institut für Informatik, Heinrich-Heine Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
e-mail: leuschel@cs.uni-duesseldorf.de

the functionality provided to users of the tool through some simple examples. We then describe key elements of the implementation of ProB. We also outline results from applying the tool to a range of industry-based case studies.

Some of the functionality of ProB was previously introduced in [42,43]. Here we provide a more comprehensive and up-to-date presentation of the tool. All of the functionality of ProB is presented in a more coherent way and the algorithms and implementation are described in more detail. We also present more experimental results and outline some of the case studies in which the tool has been used.

1.1 Animation and exhaustive exploration

Based on Prolog, the ProB tool supports automated consistency checking of B machines via *model checking* [20]. For exhaustive model checking, the given sets of a machine must be restricted to small finite sets, and integer variables must be restricted to small numeric ranges. This makes it possible to determine the enabled operations and allows the checking to traverse all the reachable states of the machine with finite types. ProB will generate and graphically display counter-examples when it discovers a violation of the invariant. ProB can also be used as an animator of a B specification. Due to a mixed depth-first/breadth-first strategy, ProB's model checking facilities are also useful for infinite state machines, not as a verification tool, but as a sophisticated debugging and testing tool.

The interactive proof process with Atelier-B or the B-Toolkit can be quite time consuming. A development can involve going through several levels of refinement to code generation *before* attempting any interactive proof [39]. This is to avoid the expense of reproving POs as the specification and refinements change in order to arrive at a satisfactory implementation. We see one of the main uses of ProB as a complement to interactive proof in that errors that result in counterexamples should be eliminated before attempting interactive proof. For finite state B machines it may be possible to use ProB for proving consistency without user intervention. We also believe that ProB can be very useful in teaching B, making it accessible to new users. Finally, even for experienced B users ProB will often unveil problems in a specification that are not easily discovered by existing tools.

1.2 Refinement checking

Refinement is a key concept in the B-method. It allows one to start from a high-level specification and then gradually refine it into an implementation, which can then be automatically translated into executable code. While there is tool support for proving refinement via semi-automatic proof (within Atelier-B [21], the B-Toolkit [8], and now also Click'n'Prove [5]), there has been up to now no automatic refinement checker in the style of FDR [27] for CSP (Communicating Sequential Processes) [32,55]. The proof-based approach to refinement checking requires that a gluing invariant be provided. In contrast, with our automatic approach no gluing invariant needs to be provided. The proof based approach to refinement is a labour intensive activity. Indeed, when a refinement does not hold it may take a while for a B user to realise that the proof obligations cannot be proven, resulting in a lot of wasted effort. In this paper we wish to speed up B development time by providing an automatic refinement checker that can be used to locate errors before any formal refinement proof is attempted. In some cases the refinement checker can actually be used as an alternative to the prover,[1] but in general the method presented in this paper is complementary to the traditional B tools.

1.3 Distinctive features and aspects of ProB

Below we summarise some of the distinctive features and aspects of ProB:

1. B is a high-level modelling language, making strong use of set theory. Compared to mainstream model checkers such as Spin [33] or SMV [15,49] the the difficulty lies in computing the individual states and possible outgoing transitions as well as finding suitable values for the constants and initial values for the variables.
2. A large part of the rich B language is covered by ProB including set comprehensions, lambda abstractions, record types and multiple machines. This is important in order to be able to deal with existing real-life specifications from industry.[2] The Event-B syntax, as introduced by AtelierB and B4Free, is also supported.
3. ProB provides support for integration of B with other formalisms. So far the integration with CSP has been implemented (described later in Sect. 8); but in principle one can link B-machines with StAC (Structured Activity Compensation) [26] (a version of CSP with a compensation mechanism to model business processes and transactions) or Object Petri nets [25]. (The ProB toolset can already be used to animate and model check StAC and Object Petri net models in isolation.)
4. ProB has been applied to industrial specifications, e.g., Volvo Vehicle Function, Mechanical Press, USB Controller, Mobile Internet Framework.

[1] Namely when all sets and integer ranges are already finite and do not have to be reduced to make animation by ProB feasible.

[2] A faster version of ProB that would only support a subset of B and thus require major rewriting of industrial specifications, would in our view not be that useful.

5. ProB provides both automated consistency checking and refinement checking. The consistency model checker uses a mixed depth-first/breadth-first heuristic providing good usability, whereas the refinement checker obtains good performance by employing on-the-fly normalisation. Recently symmetry reduction has been added, which can considerably speed up the consistency checking [44,61].
6. ProB can be used in conjunction with the existing proof-based tools for B.

2 Using ProB

In this section we introduce the functionality of ProB though some example specifications and refinements.

2.1 Automatic consistency checking

Figure 1 presents a B specification (*Scheduler0*) of a system for scheduling processes on a single shared resource. In this model, each process has a state which is either $idle$, *ready* to become active or *active* whereby it controls the resource. The current set of processes is modelled by the variable $proc$ and the *pst* variable maps each current process to a state. There is a further invariant stating that there should be no more than one active process ($pst^{-1}[\{active\}]$, the image of $\{active\}$ under the inverse of *pst*, represents the set of active processes). *Scheduler0* contains events for creating new processes, making a process ready, allowing a process to take control of the resource (*enter*) and allowing a process to relinquish control (*leave*). Each of these events is appropriately guarded by a WHEN clause.[3] In particular, the *enter* event is enabled for a process p when p is ready and no other process is active.

The definitions in the *DEFINITIONS* clause of the scheduler are used to limit the size of the given set *PROC*. Normally, definitions are used to provide macros that can be included at several places within a machine. Since the definition of *scope_PROC* is not used elsewhere in the machine, it does not affect the meaning of the specification as far as Atelier-B or the B-Toolkit are concerned. However, the definition acts as a pragma for the ProB tool. In this case ProB will automatically enumerate the given set *PROC* with the symbolic values $\{p1, p2, p3\}$. This has the effect of making the state space finite for the purposes of model checking.

Figure 2 presents a counterexample resulting from performing an automatic consistency check on a modified version of the scheduler specification using ProB. The modification involves removing $pst^{-1}[\{active\}] = \{\}$ from the guard of the *enter* operation. This trace of operations shown

```
MACHINE  Scheduler0
SETS
  PROC ;
  STATE = {idle,ready, active}
VARIABLES  proc,pst
DEFINITIONS
  scope_PROC == {p1, p2, p3}
INVARIANT
  proc ∈ ℙ(PROC) ∧
  pst ∈ proc → STATE ∧
  card(pst⁻¹[{active}]) ≤ 1

INITIALISATION
  proc; pst := {}, {}

OPERATIONS
new(p : PROC) ≙
  WHEN
    p ∈ PROC \ proc
  THEN
    pst(p) := idle ||
    proc := proc ∪ {p}
  END;

ready(p : PROC) ≙
  WHEN
    pst(p) = idle
  THEN
    pst(p) := ready
  END;

enter(p : PROC) ≙
  WHEN
    pst(p) = ready ∧
    pst⁻¹[{active}] = {}
  THEN
    pst(p) := active
  END;

leave(p : PROC) ≙
  WHEN
    pst(p) = active
  THEN
    pst(p) := idle
  END
```

Fig. 1 Scheduler specification

in the counterexample leads to a state in which both $p1$ and $p2$ are active. This state clearly violates the invariant $card(pst^{-1}[\{active\}]) \leq 1$. Although not shown in Fig. 2, ProB can pinpoint for the user the invariant clauses which are violated by a reachable state. Re-instating the condition in the guard of the *enter* operation results in a successful consistency check, i.e., an exhaustive search of the reachable states for a system of three processes finds no states that violate the invariants.

2.2 Refinement checking

Figure 3 presents a B refinement called *Scheduler1*. The refines clause indicates that *Scheduler1* is intended to be a refinement of *Scheduler0*. In this refinement, instead of mapping each current process to a state, we have a pool of idle processes, $idleset$, and a queue of ready processes, *readyq*. We also have a flag indicating whether or not there is a process currently active (*activef*). When *activef* is true, the identity of the currently active process is stored in *activep*. The queue of ready processes means that processes will become active in the order in which they became ready.[4] Now the *enter* event is enabled for process p when p is the first element in the queue and there is no active process.

We expect that *Scheduler1* is a valid refinement of the machine *Scheduler0* since any sequence of operations in *Scheduler1* should also be possible in *Scheduler0*.

[3] WHEN is the the Event-B syntax for the SELECT clause of classical B.

[4] In the *ready* event, $readyq \leftarrow p$ represents the appending of p to the end of *readyq*.

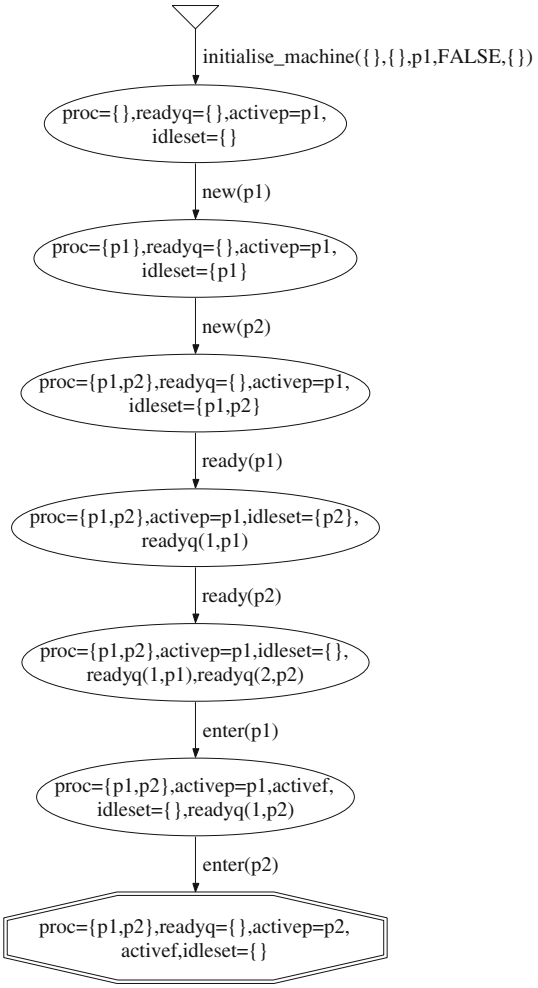

Fig. 2 Consistency counter examples for modified *Scheduler0*

Refinement checking of *Scheduler1* against *Scheduler0* with our tool for a maximum of three processes ($PROC = \{p1, p2, p3\}$) finds no counterexamples. If we were to weaken the guard of the refined *enter* event, removing the clause $activef = FALSE$, this weaker refinement would allow more than one process to take control of the single resource. In terms of operation sequences, it would allow sequences in the refinement in which, for example, $enter(p1)$ is followed by $enter(p2)$ without $leave(p1)$ occurring in between. Such sequences are not possible in *Scheduler0* and *Scheduler1* would thus be an incorrect refinement. The following counterexample is generated by ProB for the incorrect refinement: $new(p1)$, $new(p2)$, $ready(p1)$, $ready(p2)$, $enter(p1)$, $enter(p2)$. This counterexample discovered by ProB is a trace allowed by the incorrect refinement that is not a trace of the specification *Scheduler0*. This counterexample is the same as the counterexample shown in Fig. 2 generated when performing the automatic consistency checking on the incorrect version of *Scheduler0*. It is important to remember though that refinement checking is a different form of analysis

REFINEMENT *Scheduler* 1
REFINES *Scheduler* 0
VARIABLES
$proc; idleset; readyq;$
$activep, activef$
INVARIANT
$idleset \in \mathbb{P}(PROC) \wedge$
$readyq \in seq(PROC) \wedge$
$activep \in PROC \wedge$
$activef \in BOOL$

INITIALISATION
$proc := \{\} \parallel$
$readyq := [\,] \parallel$
$activep :\in PROC \parallel$
$activef := FALSE \parallel$
$idleset := \{\}$

OPERATIONS

$new(p : PROC) \triangleq$
WHEN
$p \in PROC \setminus proc$
THEN
$idleset := idleset \cup \{p\} \parallel$
$proc := proc \cup \{p\}$
END;

$ready(p : PROC) \triangleq$
WHEN
$p \in idleset$
THEN
$readyq := readyq \leftarrow p \parallel$
$idleset := idleset \setminus \{p\}$
END;

$enter(p : PROC) \triangleq$
WHEN
$readyq \neq [\,] \wedge$
$p = first(readyq) \wedge$
$activef = FALSE$
THEN
$activep := p \parallel$
$readyq := tail(readyq) \parallel$
$activef := TRUE$
END;

$leave(p : PROC) \triangleq$
WHEN
$activef = TRUE \wedge$
$p = activep$
THEN
$idleset := idleset \cup \{p\}$ k
$activef := FALSE$
END

Fig. 3 Refinement of the scheduler

to consistency checking. A consistency checking counterexample is a sequence of operation calls that leads to a violation of an invariant in a single machine. A refinement counterexample is a sequence of operation calls that is allowed in a refined machine but is not allowed in its intended abstraction.

3 The challenges of animating B

Let us first clarify some of the issues that an animator for B has to address:

1. It has to be able to find values for the constants of the machine that satisfy the PROPERTIES clause. The machines in Figs. 1 and 3 do not have constants, so this issue does not arise there. But many machines have constants, often with complicated properties. An interesting use of the animator is thus to check whether there actually exist values for the constants that satisfy the properties of the machine.
2. The animator has to find values for the variables that satisfy the INITIALISATION clause. Sometimes this is relatively straightforward—like the initialisation in Fig. 1—but in many cases the initialisation is more complicated. For example, a common initialisation clause consists of $v_1, \ldots, v_n : (INV)$ where $v_1, \ldots, v_n$ are the

variables of the machine and INV is the invariant. This is a nondeterministic assignment with the constraint that the resulting variable values must satisfy the invariant.
3. Given a state of a machine, the animator has to determine whether the INVARIANT clause is violated. Furthermore, in case the invariant is violated, it is of interest to indicate which part of the invariant was violated.
4. Given a state and parameter values, the animator should be able to decide whether a given operation is applicable or not, and, if it is, compute the effect of the operation as well as the return values. Usually, it will also be of interest to let the animator determine the possible parameter values automatically.

Unfortunately, in the general case, all of the above problems are undecidable. Indeed, integer arithmetic can be used and the set of possible states of a B machine is generally infinite, as types can be infinite (or have no fixed cardinality bound). Furthermore, the set of possible values for an individual operation parameter can be infinite. The same is true for existentially or universally quantified variables, as well as various local variables introduced in nondeterministic substitutions. In particular, it is thus undecidable whether an operation can be applied or not, even if the initial state as well as all arguments are completely specified. Preconditions and guards can be arbitrarily complex with existential or universal quantification over infinite sets. Let us examine a simple example to illustrate this point:

```
gold =  WHEN !n.(n:NAT & n>2 =>
    #(x,y).(x<2*n & y<2*n & 2*n=x+y &
     card({z|z:NAT1 & z<x & x mod z = 0})=1 &
     card({z|z:NAT1 & z<y & y mod z = 0})=1))
    THEN  skip  END
```

This is a perfectly legal B operation which can be executed if and only if Goldbach's conjecture (i.e., that every even number greater than 2 can be expressed as the sum of two primes) is true.[5] Similarly, given that deferred sets can be infinite we get undecidability via this route, even in the absence of arithmetic.

3.1 Making things decidable via finite types

Every variable, constant and parameter in B can be given a type. Below, we formally define the set of types that are allowed in B. The full details about type inference can be found in [2].

Recall, that in B there are two ways to introduce sets into a B machine: either as a parameter of the machine (by convention parameters consisting only of upper case letters are sets; the other parameters are integers) or via the SETS clause. Sets introduced in the SETS clause are called *given sets*. Given sets which are explicitly enumerated in the SETS clause are called *enumerated sets*, the other given sets are called *deferred sets*. Other types may be constructed using the Cartesian product ($\times$) and powerset ($\mathbb{P}$) constructors.

Definition 1 *Let M be a B machine with given sets S_M and parameter sets P_M. The basic types BasicType of the machine M are inductively defined as the least set satisfying*:

1. $BOOL \in BasicType$
2. $\mathbb{Z} \in BasicType$
3. $S \in BasicType$ *if* $S \in S_M \cup P_M$
4. $\tau_1 \times \tau_2 \in BasicType$ *if* $\tau_1 \in BasicType \wedge \tau_2 \in BasicType$
5. $\mathbb{P}(\tau) \in BasicType$ *if* $\tau \in BasicType$

One way to make animation decidable is to ensure that via typing, any variable, parameter or constant can only take on finitely many possible values. This can be accomplished by requiring that, at least for the purposes of the animation, all sets in $S_M \cup P_M$ of a B machine be finite. Note that enumerated sets are already finite; hence we just need to fix some finite cardinality for the deferred sets and parameter sets. We also only consider B's *implementable* integers, ranging from *MININT*..*MAXINT* (see, e.g., [21]).[6] Furthermore, for the purposes of animation, we will usually set *MININT* and *MAXINT* to small absolute values, but allow larger values if they are explicitly used or constructed by the machine. (We return to this issue later, as it has some implications for soundness.)

These restrictions turn animation of B into a decidable problem. A naïve solution to the animation problem is thus simply to enumerate all possibilities for the values under consideration; e.g., to find possible values of the constants that satisfy the PROPERTIES clause, we "simply" need to enumerate all possible values for the constants and check whether all PROPERTIES evaluate to true (this in turn can be decided as existential and universal variables also only have finitely many possible values).

3.2 Efficiency

Above we have seen how to make animation decidable by ensuring that every variable, parameter and constant has only finitely many possible values. But obviously the number of possible values will often be of such considerable size so as to make the sketched decision procedure impractical. Take for example the following predicate: `myrels: POW(A<->A)`

[5] While the conjecture may eventually be decided by mathematicians (see, however, [38] where Knuth argues that it may be unprovable), it is clearly outside the range of current automated theorem proving methods to do so.

[6] Machines using the mathematical set of integers are allowed, but they are treated as implementable integers.

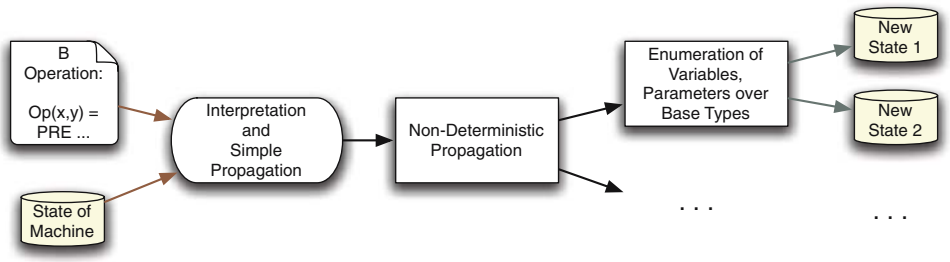

Fig. 4 Basic phases of ProB animation

where `A` is a deferred set. The basic type of the variable `myrels` is `POW(POW(A*A))`. Assuming that we set the cardinality of `A` to 4, the variable `myrels` has $2^{2^{4*4}} = 2^{65536}$ possible values, and even with a cardinality of 3 we still have $2^{512} \approx 1.34 * 10^{154}$ possible values.

This shows that we should avoid or at least delay enumeration as much as possible. In case enumeration is unavoidable, we may have to set the cardinality of the basic sets to small or very small values (e.g., with a cardinality of 2 we only get 65536 possible values for `myrels`).

The former is implemented within ProB, which works in multiple phases as illustrated in Fig. 4.[7] In the first phase, only deterministic propagations are performed (e.g., the predicate `x=1` will be evaluated but the predicates `x:INT` and `y:z` will suspend until they either becomes 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:{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.

In summary, a predicate of the form `x: NAT & x<10 & x=5` will thus result in no enumeration at all: phase 1 will determine that the only possible value for x is 5. Similarly, for `f:A-->A & !x.(x:A => f(x) = x)` no enumeration for f will be required (and the ProB kernel can quite easily handle cardinalities of above 100 for A).

Some of the further challenges of animating B are detailed below:

1. B provides sophisticated data structures, including sets, Cartesian products, relations, sequences, etc. This also means that deciding whether two variables have the same value is a non-trivial task (e.g., $\{a, b\} = \{b, a\}$ or $\{\{a, b\}, \{a\}, \{c, a\}\} = \{\{a\}, \{c, a\}, \{b, a\}\}$). It is thus also non-trivial to decide whether a given state of the machine has already been encountered or not.
2. B provides a large range of operations over the datatypes, ranging from basic set operations such as union or intersection, up to more involved operations such as inverting or computing the transitive closure of a relation. The use of lambda abstractions or set comprehensions are also especially tricky, due to the need to convert arbitrarily complex predicates into sets of values (e.g., a simple example would be: `r={y|y:ran(f) & card(f~[{y}]) = 1}`, where r are all the elements in the range of f which are the image of a single element in the domain of f).

In ProB these issues are dealt with by the ProB-kernel, which treats the basic datatypes of B and their operations. The ProB-kernel is implemented in Prolog with co-routines. This kernel is tailored for extensibility and deals with almost all B operators. It is also capable of dealing with large data values. In order to represent B's data structures we have employed classical Prolog terms, notably representing sets as lists without repetition. In order to avoid multiple representations of the same state, these Prolog representations are normalised.

In the following section we go into more detail about the architecture and implementation of the ProB toolset.

4 The implementation of ProB

4.1 Overview

The overall architecture of ProB is shown in Fig. 5. To read in the AMN (Abstract Machine Notation) syntax we employ the *jbtools* [60] parser by Bruno Tatibouet; a parser written using `javacc`, which we slightly extended to support the application of functions with multiple arguments (allowing `f(a,b)` rather than `f(a|->b)`, for example), as well as various other syntactic extensions employed by AtelierB and B4Free. This parser produces the abstract syntax tree in XML format, which is converted into a Prolog encoding suitable for the ProB interpreter. The ProB interpreter evaluates the B Machine's constructs and calls the ProB kernel to treat the core B datatypes and operators. The ProB interpreter itself is driven by various other components of ProB, one being the ProB animator.

[7] Notice that type inference is not shown in the figure; it is run once when a new machine is loaded for animation. Also, the figure just illustrates the problem of determining the enabled operations. The procedure for finding valid constants and initialisations is similar.

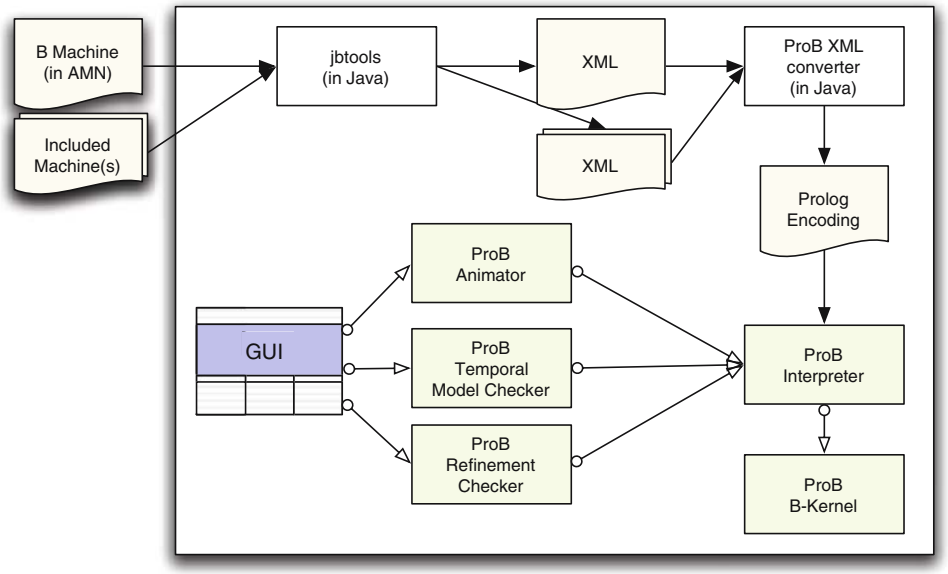

Fig. 5 The main components of ProB

4.2 The ProB interpreter

The ProB interpreter is written in a structured operational semantics [51] (SOS) style. More precisely, given a description σ_1 of the state of a B machine, we describe which operations (with parameter values) can be applied in σ_1 and which new states can be reached by performing those operations. For this, the constructs of B were divided into three main classes:

1. B *substitutions*, which modify the variables of a B machine,
2. *B expressions*, which do not modify the variables but denote values, and
3. B *predicates*, which are either true or false.

To manipulate these constructs, the ProB interpreter contains Prolog predicates[8] to execute statements, compute expressions, and test Boolean expressions. Each of these Prolog predicates has access to the global state of the machine (the state of the variables of a machine) and a local state that contains the values for local variables and parameters of operations. In order to manipulate B's basic data structures and operators, the ProB interpreter calls the ProB kernel, which we discuss later.

Here is a very small part of the interpreter that tests Boolean expressions, responsible for handling the logical connectives "and" and "or":

```
b_test_boolean_expression('And'(LHS,RHS),LocalSt,State):-
      b_test_boolean_expression(LHS,LocalSt,State),
      b_test_boolean_expression(RHS,LocalSt,State).
b_test_boolean_expression('Or'(LHS,RHS),LocalSt,State):-
   b_test_boolean_expression(LHS,LocalSt,State)
        ;  /* or */
   (b_not_test_boolean_expression(LHS,LocalSt,State),
    b_test_boolean_expression(RHS,LocalSt,State)).
```

The first argument of the Prolog predicate is the encoding of the Boolean expression to be tested. The second argument (`LocalSt`) contains the values of all variables local to an operation, i.e., the choice variables from *Any* statements and the operation's arguments. The third argument (`State`) contains the values of all "global" variables and constants of the B machine under consideration. The `b_test_boolean_expression` predicate also has a counterpart, `b_not_test_boolean_expression`, which is used to check whether a Boolean expression evaluates to false.This is required, as Prolog's built-in negation is not sound in general.

For expressions, the corresponding Prolog predicate has an extra argument to return the value of the expression, while for substitutions the corresponding Prolog predicate has an extra argument where it returns the updates (i.e., changed variables with their new values).

While it is non-trivial to cover the vast syntax of B, the code of the ProB interpreter is for the most part relatively simple. The reason is that the ProB kernel is very flexible and "hides" much of the complexity of B from the ProB interpreter. In fact, while the ProB interpreter is written in classical Prolog, the ProB kernel uses the co-routining features of Prolog to provide a robust foundation, which allows the interpreter to be written in a straightforward way. The kernel also provides the various propagation phases shown in Fig. 4. In the next section, we present more details about the ProB

[8] Note that there is a potential confusion concerning the use of the word "predicate" in B and in Prolog.

kernel. Some complicated aspects inside the interpreter are the treatment of set comprehensions, lambda abstractions, as well as universal and existential quantification. Here, co-routining is used to defer the evaluation of such constructs until sufficient information is available. More precisely:

- $\exists x.(x \in XType \ \wedge \ P)$ will suspend until all open variables in P (i.e., free variables of P excluding x) have received a value, at which point the interpreter will try to find a (single) solution for P.
- $\forall x.(x \in XType \Rightarrow P)$ will be expanded out into a large conjunction if possible; otherwise it will suspend in a similar way to the existential quantification. For example, in the context of the machine from Fig. 1, the formula $\forall p.(p \in PROC \Rightarrow pst(p) = s)$ would be expanded out into $pst(p1) = s \wedge pst(p2) = s \wedge pst(p3) = s$.
- $\{x \mid x \in XType \ \wedge \ P\}$ will suspend until all open variables in P have received a value, at which point the set is computed. Lambda abstractions are treated by converting them into a set comprehension with an additional parameter (the return value).[9]

4.3 The ProB kernel

First, let us see how some of B's data structures are actually encoded by the ProB Kernel:

B Type	B value	Prolog encoding
number	5	`int(5)`
boolean	true	`term(bool(1))`
element of set S	C	`fd(3,'S')`
pair	$4 \mapsto 5$	`(int(4),int(5))`
set	$\{4, 5\}$	`[int(4), int(5)]`
relation	$\{4 \mapsto 5\}$	`[ (int(4),int(5)) ]`
sequence	[4, 5]	`[(int(1),int(4)), (int(2),int(5))]`

As can be seen, sets are represented by Prolog lists; the ProB kernel ensures that the same element is not repeated twice within a list. The Prolog term `fd(3,'S')` represents the third element of the given or deferred set S. So if S is defined within the B machine by $S = \{A, B, C, D\}$ then `fd(3,'S')` denotes the constant C. Sequences are encoded in the standard B style, i.e., as a function from $1..size(s)$ to the elements of the sequence.

The kernel then contains Prolog predicates for all of B's operators and mainly uses SICStus Prolog's `when` co-routining predicate to control the enumeration of B values. More precisely, the binary `when` predicate [59] suspends until its first argument becomes true, at which point it will call its second argument. From a logical point of view, the when declarations can be ignored, as they are just annotations guiding the Prolog execution engine: they do not change the logical meaning of a Prolog program. We employ the coroutining to ensure that enumeration will be deferred until either only a single or no possible value remains, or until the ProB kernel has been instructed to move to a more aggressive enumeration phase.

In working with finite base types and enumeration, the ProB kernel has some similarities with the classical finite domain constraint solver CLP(FD) [19]. However, there are also considerable differences:

- Our solver provides multiple phases, but does not yet provide a way to control the enumeration order within a single phase.
- We provide multiple datatypes building upon the finite domain base types.
- We provide many sophisticated operations over the basic finite domains.

4.4 The ProB animator

The first graphical user interface of the ProB animator was developed using the Tcl/Tk library of SICStus Prolog. The user interface was inspired by the ARC tool [31] for system level architecture modelling and builds upon our earlier animator for CSP [40].

Our animator supports (backtrackable) step-by-step animation of the B-machines. As can be seen in Fig. 6 it presents the user with a description of the current state of the machine, the history that has led the user to reach the current state, and a list of all the enabled operations, along with proper argument instantiations. Thus, unlike the animator provided by the B-Toolkit, the user does not have to guess the right values for the operation arguments. The same holds for choice variables in nondeterministic assignments where the user does not have to find values that satisfy the constraint. If the number of enabled operations becomes larger, one could envisage a more refined interface where not all options are immediately displayed to the user. This is being developed in the Rodin platform within Eclipse [4]. The current version already allows the user to set an upper limit on the number of ways the same operation can be executed.

The ProB animator also provides visualisation of the state space that has been explored so far, and provides visual feedback on which states have been fully explored and which ones are still "open." For the visualisation we make use of the dot tool of the *graphviz* package [7], and various ways to visualise larger state spaces compactly have been developed and implemented [47].

5 Exhaustive consistency checking in ProB

In this section we outline the method of exhaustive consistency checking implemented in ProB.

[9] In recent work [45] we have developed a method to keep certain set comprehensions and lambda abstractions symbolic, only evaluating them on demand.

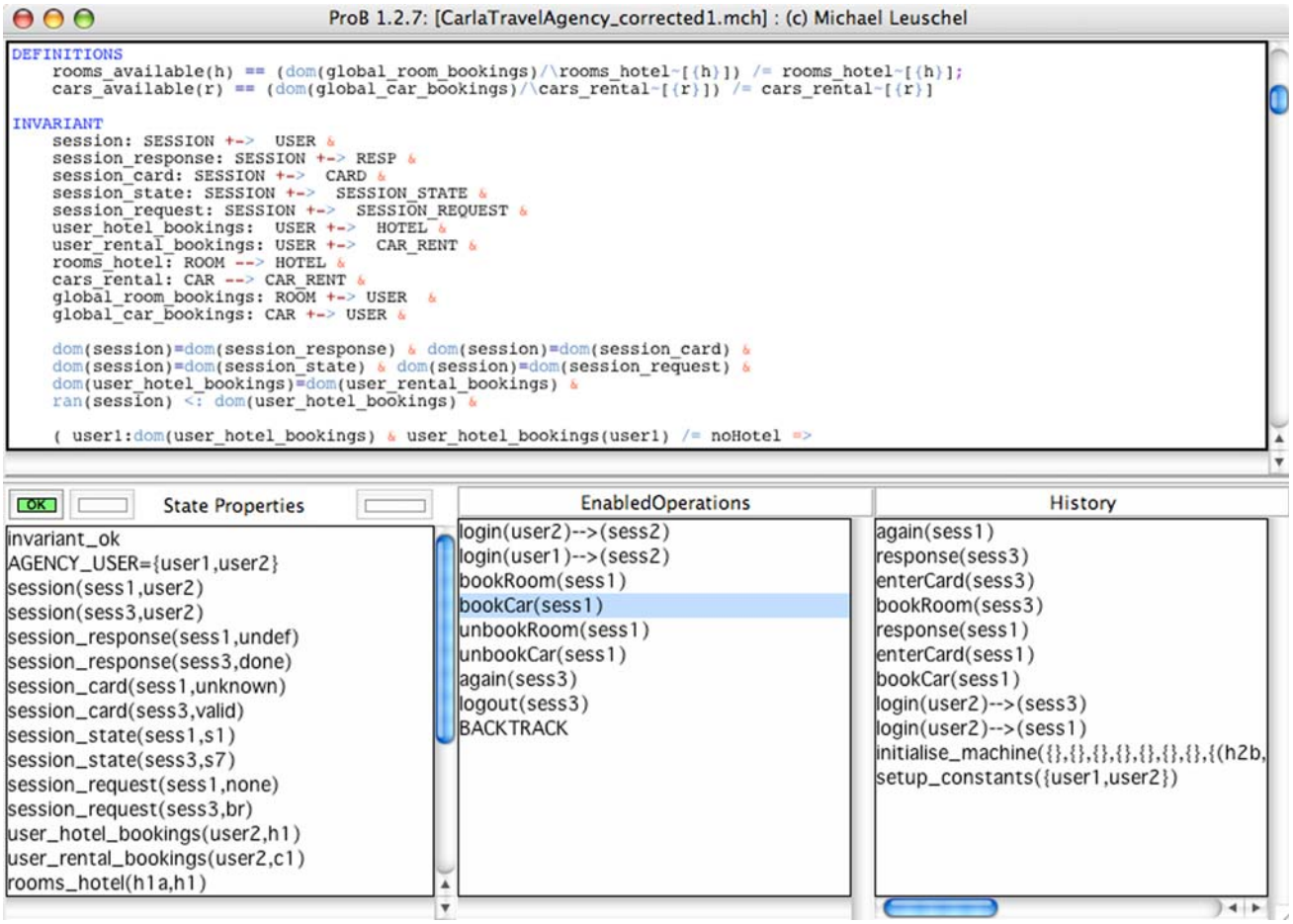

Fig. 6 Animation of the E-travel agency case study (cf. Sect. 7.3.2)

5.1 Overview of the algorithm

By manually exploring a B-machine using the PROB animator, it is possible to discover problems with a machine, such as invariant violations, deadlocks (states where no operation is applicable) or other unexpected behaviour not encoded in the invariant. We have implemented a model checker [20], which will do such an exploration systematically and automatically. It will alert the user as soon as a problem has been found, and will then present the shortest trace (within currently explored states) that leads from an initial state to the error. The model checker will also detect when all states have been explored, and can thus also be used to formally guarantee the absence of errors. This will obviously only happen if the state space is finite (and small enough to fit into memory), but the automatic consistency checker can also be applied to B machines with large or infinite state spaces and will then explore the state space until it finds an error or runs out of memory.

The model checker drives the PROB interpreter in the same way that the PROB animator does. In addition, the model checker needs to keep track of which states have already been explored, and needs to decide which unexplored state to investigate next.

To avoid the same state (e.g., $s_1 = \langle\{a, b\}\rangle$ and $s_2 = \langle\{b, a\}\rangle$) being treated multiple times, the PROB interpreter contains a *normalisation* procedure. Furthermore, to quickly determine whether a particular state of a B machine has already been encountered, state *hashing* is used.

The exploration is derived from the A* algorithm, and can be tuned to perform in the extreme cases as either a depth-first or breadth-first exploration. The default behaviour uses a *mixed depth-first breadth-first strategy*, where a random factor is used to decide whether any given node will be treated in depth-first or breadth-first order. This heuristic has proven itself to be very good in practice. Indeed, in our case studies, at least for the initial machines being developed, errors were easy to find and often fell into one of the following two categories:

1. Systematic errors inside an operation that occur in most states; here it is not important to locate a particular state just to systematically try out all operations for all arguments.

2. Errors that arise when the machine is animated long enough (e.g., deadlock errors); here it is often not important which particular path is taken, just that the machine is animated long enough.

Breadth-first is good at picking out errors of type 1 but may fail to find errors of type 2 (if the state space is too big to be explored exhaustively). Depth-first is good at picking out errors of type 2 but may fail to find errors of type 1. Our heuristic mixed strategy will pick out both types of errors quickly.

The visited states are stored in Prolog's clause database. While this is not as efficient as for example tabling,[10] it allows the model checking state to be easily queried (e.g., for visualisation) and saved to file. For a formalism as rich as B, most of the model checking time is spent evaluating the invariant and computing the enabled operations and the resulting new states. The time needed to look up whether a given state has already been encountered is typically not the bottleneck.

Currently the consistency checker detects the following conditions:

1. Invariant violation errors;
2. Assertion violation errors (assertions are properties of a B machine that should follow from the invariant);
3. Deadlock errors (a deadlocked state is one in which no operation is enabled);
4. When a user-specified goal predicate becomes true.

Full temporal logic model checking was not supported in the early releases of ProB, but it could be achieved by other means (namely by refinement checking with CSP processes, see Sect. 8). In the latest release, a full-blown LTL model checker has been integrated in ProB.

Below we present a formal description ProB's model checking algorithm; experimental Results are presented later in Sect. 7.

5.2 Formalisation

In [2], the semantics of B operations is defined in terms of weakest precondition rules. For the purposes of making the link between B and model-checking we find it convenient to treat B operations as relations on a state space. The state space of a machine is defined as the Cartesian product of the types of each of the machine variables. We represent the machine variables by a vector v. Classical B distinguishes between an enabling condition (guard) and a precondition in operations. The difference between a guard and a precondition is that an operation can never be executed outside its guard while it can be executed outside its precondition but in that case its behaviour is aborting [2].

The B syntax supported by ProB allows preconditions, but they are treated as guards.[11] If we ignore preconditions but allow for guards, then all B operations have a normal form defined by a characteristic predicate P relating before state v, after state v', inputs x and outputs y as follows [2, Chap. 6]:

$$\text{ANY } v', y' \text{ WHERE } P(x, v, v', y') \text{ THEN } v, y := v', y' \text{ END}$$

This statement nondeterministically assigns values v', y' to v and y such that $P(x, v, v', y')$ holds. Characterising a B operation by a predicate in this way gives rise to a labelled transition relation on states: state s is related to state s' by event $op.a.b$, denoted by $s \rightarrow^{M}_{op.a.b} s'$, when $P(a, s, s', b)$ holds.

The syntactic constraints on initialisation operations in B are such that the outcome of an initialisation will be independent of the initial values of the variables. This means an initialisation has a normal form

$$\text{ANY } v' \text{ WHERE } P(v') \text{ THEN } v := v' \text{ END}$$

In this case, P is used to define a set of initial states for a machine. For convenience we add a special state *root*, where we define $root \rightarrow^{M}_{initialise} s$ if s satisfies the initialisation predicate.

Below we describe ProB's consistency checking algorithm. The algorithm employs a standard queue data structure to store the unexplored nodes. The function *error* determines whether a given state gives rise to an error, e.g., $error(state)$ will check whether $state \notin I$ (invariant violation) and whether no operations can be applied (deadlock).

The key operations are computing "$state \rightarrow^{M}_{op} succ$" and "$state \notin I$" (both of which are achieved by the ProB interpreter, in turn calling the kernel) and determining whether "$succ \notin States$" (which is performed by normalising *succ*, computing the hash value of *succ* and then checking all nodes in *States* with the same hash value for equality with the normal form of *succ*). The algorithm terminates when there are no further queued states to explore or when an error state is discovered.

[10] A tabled logic programming system such as XSB [56] provides very efficient data structures and algorithms to tabulate calls, i.e., it remembers which calls it has already encountered. This can be used to write very efficient model checkers [46,52].

[11] It is possible to set a ProB preference so that preconditions are treated differently from guards. However, we are focussing our efforts on migrating ProB towards supporting Event-B [1] which supports guards but not preconditions.

Algorithm 5.1 [*Consistency Checking*]

Input: An abstract machine M with invariant I
$Queue := \{root\}$; $States := \{root\}$; $Graph := \{\}$
while $Queue$ is not empty **do**
 if random(1) $< \alpha$ **then**
  $state :=$ pop_from_front($Queue$); /* depth-first */
 else
  $state :=$ pop_from_end($Queue$); /* breadth-first */
 end if
 if $error(state)$ **then**
  return counter-example trace in $Graph$
   from $root$ to $state$
 else
  for all $succ$,op **such that** $state \rightarrow^{M}_{op} succ$ **do**
   $Graph := Graph \cup \{state \rightarrow_{op} succ\}$
    if $succ \notin States$ **then**
     add $succ$ to front of $Queue$
     $States := States \cup \{succ\}$

    end if
   end if
  end for
od
return ok

5.3 Relationship with the classical B proof method

In this section we outline how exhaustive consistency checking of a (finite) B machine relates to the standard proof-based approach to consistency in B. For a machine to be consistent, its initialisation must establish the invariant, and each operation must preserve the invariant. Expressed in terms of the relational formulation of B machines outlined above, the consistency obligations for a B machine with invariant I, initialisation $Init$ and operations OP_i are as follows :[12]

$$Init \subseteq I$$
$$s \in I \ \wedge \ s \rightarrow^{M}_{op} s' \ \Rightarrow \ s' \in I, \quad \text{for each operation } op$$

When ProB finds a counterexample, the final transition of the counterexample is from a state satisfying the invariant to a state falsifying the invariant. It is easy to see that such a transition falsifies these consistency conditions.

In the case where ProB finds no counterexamples in an exhaustive check and the machine contains only finite types (i.e., no deferred sets or integers), then consistency can be proven. Let us consider this further. When taking a proof approach to consistency checking in B, it is often the case that the desired invariant is not strong enough to be provable and a stronger invariant I' is required (by adding conjuncts to I). A successful exhaustive consistency check computes the set of reachable states R and will have checked that all of those states satisfy the invariant I. This set of reachable states R corresponds to a stronger invariant since after successful termination of the algorithm we have

$$R \subseteq I$$
$$Init \subseteq R$$
$$s \in R \ \wedge \ s \rightarrow^{M}_{op} s' \ \Rightarrow \ s' \in R, \quad \text{for each operation } op$$

Thus the set of reachable states R is a sufficient invariant to prove consistency w.r.t. the original invariant I in the standard way.

In the case where ProB finds no counterexamples in an exhaustive check and the machine contains deferred sets or integers, then we cannot conclude that the machine with infinite types is consistent. As usual with model checking, we may find a counterexample with larger scopes for types that do not appear with smaller scopes. But lack of counterexamples will at least give us more confidence that the proof will go through.

6 Refinement checking for B

In this section we outline the B notion of refinement. We outline the trace behaviour of B machines and trace refinement for B machines and relate it to standard B refinement. We then explain the automatic refinement checking algorithm implemented in ProB.

B refinement is defined in terms of a gluing invariant which links concrete states to abstract states. In [2], refinement checking checking rules are defined in terms of weakest precondition rules for B operations. As in the previous section, we express the refinement proof obligations in terms of the relational model for B machines. These proof obligations correspond to the standard relational definition of forward simulation. Let R be the gluing relation, AI and CI be the abstract and concrete initial states respectively and aop and cop stand for corresponding abstract and concrete operations. The usual relational definition of forward simulation is as follows [30]:

- Every initial concrete state must be related to some initial abstract state: $c \in CI \ \Rightarrow \ \exists a \in AI \ \cdot \ c \ R \ a$
- If states are linked and the concrete one enables an operation, then the abstract state should enable the corresponding abstract operation and both operations should result in states that are linked: $c \ R \ a \ \wedge \ c \rightarrow^{M}_{cop} c' \Rightarrow \exists a' \cdot a \rightarrow^{M}_{aop} a' \ \wedge \ c' \ R \ a'$

The proof obligations for refinement are automatically generated from the gluing invariant and the definitions of the abstract and concrete operations by, for example, AtelierB or the BToolkit. The user can then try to prove these using the

[12] This is easy to demonstrate by using the normal form for operations characterised by a before-after predicate and the weakest precondition rules for B.

semi-automatic provers of those systems. If the proof obligations are all proven, every execution sequence performed by the refinement machine can be matched by the abstract machine [17]. Automatic refinement checkers work directly on the execution sequences and try to *disprove* refinement by finding traces that can be performed by the refinement machine but not by the specification. For this we need to formalise the notions of execution sequences (traces) for B.

6.1 Traces

The use of event traces to model system behaviour is well-known from process algebra, especially CSP [32]. Although event traces are not part of the standard semantic definitions in B, many authors have made the link between B machines and event traces including [17,24,57]. The ProB animator can also be viewed as a way of computing sample traces of a B machine.

We regard execution of a B operation *op* with input value a resulting in output value b as corresponding to the occurrence of event $op.a.b$. An event trace is a sequence of such events and the behaviour of a system may be defined by a set of event traces. For example, the following is a possible trace of the scheduler specification of Fig. 1:

$$\langle\, new.p1, new.p2, ready.p1, ready.p2, enter.p1, leave.p1 \,\rangle$$

In Sect. 5.2, we have already defined the labelled transition relation $s \rightarrow^M_{op.a.b} s'$, linking two states s and s' via the event $op.a.b$, when the operation *op* can be executed in the state s with parameters a, giving rise to outputs b and the new state s'. This transition relation $\rightarrow^M_e$ is lifted to traces using relational composition:

$$\begin{array}{lcl} \rightarrow^M_{\langle\rangle} & = & ID \\ \rightarrow^M_{\langle e\rangle t} & = & \rightarrow^M_e \;;\; \rightarrow^M_t \end{array}$$

Note that ID is the identity relation over states. Now t is a possible trace of machine M if $\rightarrow^M_t$ relates some initial state to some state reachable through trace t: $t \in traces(M) = \exists c, c' \cdot c \in CI \;\wedge\; c \rightarrow^M_t c'$.

6.2 Trace refinement checking

A machine M is a trace refinement of a machine N if any trace of N is a trace of M, that is, any trace that is possible in the concrete system is also possible in the abstract system. It is straightforward to show by induction over traces that if we can exhibit a forward simulation between M and N with some gluing relation, then M is trace refined by N. It is known that forward simulation is not complete, i.e., there are systems related by trace refinement for which it not possible to find a forward simulation. The related technique of backward simulation together with forward simulation make simulation complete [30]. A backward simulation is defined as follows:

$$\begin{array}{l} c \in CI \;\wedge\; c\; R\; a \;\;\Rightarrow\;\; a \in AI \\ c \rightarrow^M_{cop} c' \;\wedge\; c'\; R\; a' \;\Rightarrow\; \exists a \cdot c\; R\; a \;\wedge\; a \rightarrow^M_{aop} a' \end{array}$$

The B tools produce proof obligations for forward simulation only. There are cases of refinement where, although the trace behaviour of the concrete system is more deterministic, an individual concrete operation is less deterministic than its corresponding abstract operation. Backwards refinement is required in such cases. Typical developments B involve the reduction of nondeterminism in operations so that forward simulation is sufficient in most cases.

A single complete form of simulation can be defined by enriching the gluing structure. Gardiner and Morgan [28] have developed a single complete simulation rule by using a predicate transformer for the gluing structure. Such a predicate transformer characterises a function from *sets* of abstract states to *sets* of concrete states. Refinement checking in ProB works by constructing a gluing structure between the concrete and abstract states as it traverses the state spaces of both systems. So that we have a complete method of refinement checking, the ProB checking algorithm constructs a gluing structure that relates concrete states with sets of abstract states: $R \;\in\; C \leftrightarrow \mathbb{P}(A)$.On successful completion of an exhaustive refinement checking run the constructed gluing structure R will relate each individual concrete initial state to the set of abstract initial states and for each pair of corresponding concrete and abstract states, the following simulation condition will be satisfied:

$$c\; R\; as \;\wedge\; c \rightarrow^M_{cop} c' \;\Rightarrow\; \exists as' \cdot as \rightarrow^M_{aop} as' \;\wedge\; c'\; R\; as'$$

Here as and as' represent sets of abstract states and $as \rightarrow^M_{aop} as'$ holds when as' is the largest set of states to which to states of as are mapped by $\rightarrow^M_{aop}$. It can be shown by induction over traces that this entails trace refinement, i.e., a successful outcome of the algorithm guarantees trace refinement. Because ProB works on finite state systems, the algorithm always terminates successfully or by detecting a failure. Completeness of the algorithm is proven by demonstrating that whenever the outcome is failure, then there is a violation of trace refinement.

6.3 The refinement checking algorithm

We now present an algorithm to perform refinement checking. The gluing structure discussed in Sect. 6.2. is stored in *Table*, and for every entry (c, as) the algorithm checks whether all operations of the concrete state c can be matched by some abstract state in the set of state as; if not, a counter example has been found, otherwise all concrete successor states are computed and put into relation with the

corresponding abstract successor states. To ensure termination of the algorithm it is crucial to recognise when the same configuration is re-examined. This is done by the check "(c,as) $\notin$ *Table*". If that check fails we know that we can safely stop looking for a counter example. Indeed, if one counter example exists we know that we can find a shorter version starting from the configuration that is already in the table.

Algorithm 6.1 [*Refinement Checking*]

Input: An abstract machine M_A and
a refinement machine M_R
Table := { } ; *Res* := refineCheck(*root*, {*root*}, ⟨⟩);
if *Res* = ⟨⟩ **then println** 'Refinement OK'
else println('Counter Example:',Res)
end if

function refineCheck(*ConcNode*, *AbsNodes*, *Trace*)
if (*ConcNode*, *AbsNodes*) $\notin$ *Table* **then**
Table := *Table* $\cup$ {(*ConcNode*, *AbsNodes*)};
for all *CSucc*, *Op* **such that**
$ConcNode \rightarrow^{M_R}_{Op} CSucc$ **do**
TraceS := **concat**(*Trace*, ⟨(*Op*, *CSucc*)⟩);
$ASucss := \{a' \mid \exists a \in AbsNodes \wedge a \rightarrow^{M_A}_{Op} a'\}$;
if *ASucss* = ∅ **then**
return *TraceS*
else
Res := refineCheck(*CSucc*, *ASucss*, *Trace*);
if *Res* $\neq$ ⟨⟩ **then return** *Res* **end if**
end if
end for
end if
return ⟨⟩
end function

6.4 Implementation

We have developed two implementations of the refinement checking algorithm. The first one is implemented inside the ProB toolset, i.e., using SICStus Prolog. The tabling is done by maintaining a Prolog fact database, which is updated using `assert/1`. For refinement checking, the abstract state space currently has to be computed beforehand (using ProB). To ensure completeness of the refinement checking, it should be fully computed. However, our refinement checker also allows the abstract state space to be only partially computed. In that case, the refinement checker will detect whether enough of the state space has been computed to decide the refinement (and warn the user if not). In the SICStus Prolog implementation the state space of the implementation can be computed beforehand, but does not have to be. In other words, the implementation state space will be expanded *on-the-fly*, depending on how the refinement checking algorithm proceeds. This is of course most useful when counter examples are found quickly, as in those cases only a fraction of the state space will have to be computed. In future work, we plan to enable this on-the-fly expansion also for the abstract state space.

The second implementation has been done in XSB Prolog. The code of the XSB refinement checker is almost identical, but instead of using a Prolog fact database it uses XSB's efficient tabling mechanism [56]. As we will see later, this implementation is faster than the SICStus Prolog one. However XSB Prolog does not support constraint solvers in the same way as SICStus. This means that the abstract and concrete state spaces need to be computed beforehand using the SICStus ProB and then loaded into the XSB version of the refinement checker. The overhead of starting up a new XSB Prolog process and loading the states space is only worth the effort for larger state spaces (and even then only if there are no or difficult-to-find counter examples).

7 Experimental results for consistency and refinement checking

To test the performance of our consistency and refinement checker, we have conducted a series of experiments with various models. As well as using the scheduler example from Sects. 2.1 and 2.2, we have experimented with a much larger development of a mechanical by press by Abrial [3]. The development of the mechanical press started from a very abstract model and went through several refinements. The final model contained "about 20 sensors, 3 actuators, 5 clocks, 7 buttons, 3 operating devices, 5 operating modes, 7 emergency situations, etc." [3]. We were able to apply our model checker and refinement checker to successfully validate consistency as well as various refinement relations. Furthermore, as no finitisation was required for the mechanical press (i.e., all types were already finite from the start), the consistency and refinement checker can actually be used in place of the traditional B provers. In other words, we are thus able to automatically prove consistency and refinement using our tool. To check the ability of our tool to find errors we have also applied it to an erroneous refinement (m2_err.ref), and ProB was able to locate the problem in a few seconds.We have also experimented with a simple example of a server allowing clients to log in (Server.mch and Server.ref). Precise timings and results for these and other experiments are presented in the next subsections.

7.1 Consistency checking

In a first phase we have performed classical consistency and deadlock checking on our examples using ProB's model checker. The results can be found in Table 1, and give an indication of the size of the state space and how expensive it is to compute the reachable state space. The experiments were all run on a PowerPC G5 Dual 2.5 GHz, running

Table 1 ProB consistency checking and size of state space

Machine	Time	States	Transitions
Server.mch	0.013 s	5	9
ServerR.ref	0.05 s	14	39
scheduler0.mch	46 s	55	190
scheduler1.ref	0.93 s	145	447
scheduler0_6.mch	41.37 s	2,188	14,581
scheduler1_6.ref	501.61 s	37,009	145,926
m0.mch	3.19 s	65	9,924
m1.ref	20.38 s	293	47,574
m2.ref	44.29 s	393	59,588
m2_err.ref	31.51 s	405	61,360
m3.ref	364.90 s	2,693	385,496

Mac OS X 10.3.9, SICStus Prolog 3.12.1 and ProB version 1.1.5. Note, while the machine had 4.5 Gigabytes of RAM, only 256 Megabytes are available in SICStus Prolog 3.12 for dynamic data (such as the state space of B machines). *scheduler0.mch* and *scheduler1.ref* are the machines presented above in Sects. 2.1 and 2.2 for 3 processes, while *scheduler0_6.mch* and *scheduler1_6.ref* are the same machines but for 6 processes. The machines *m0.mch*, *m1.ref, m2.ref*, m2*_err.ref*, and *m3.ref* are from the mechanical press example discussed above. *Server.mch* is a simple B machine describing the server example, while *ServerR.ref* is a refinement thereof.

7.2 Refinement checking

Table 2 are the results of performing various refinement checks on these machines. Entries marked with an asterisk mean that no previous consistency checking was performed, that is, the reachable state space of the implementation machine was computed on-the-fly, as driven by the refinement checker. For entries without an asterisk, the experiment was run straight after the consistency checking of Table 1; that is, the reachable state space was already computed and the time is thus of the refinement checking proper. The figures show that our checker was very effective, especially if counter examples existed.

In Table 3 we have conducted some of the experiments where the refinement checker is run as a separate process using XSB Prolog [56], rather than inside ProB under SICStus Prolog. Our experiments confirm that XSB's tabling mechanism leads to a more efficient refinement checking (cf. the third column). However the time to start up XSB and load the state space is not negligible, meaning that the XSB approach does not always pay off. This can be seen in the fourth column, which contains the total time for loading and checking: e.g., the approach pays off for the m2.ref check against m1.ref (overall gain of 30 s) but not for the smaller examples or when a counter example is found quickly.

We have also compared our new refinement checker against a widely known refinement checker, namely FDR [27]. FDR is a commercial tool for the validation of CSP specifications. The results of the experiments can be found in [43]. The conclusion was that our algorithm compares favourably with FDR, and that the on-the-fly normalisation was an important aspect for the examples under consideration.

Table 2 ProB refinement checking and size of refinement relation

Refinement	Specification	Time (in s)	Table size
Successful refinements			
ServerR.ref	Server.mch*	0.05	14
ServerR.ref	Server.mch	0.00	”
scheduler1.ref	scheduler0.mch*	0.73	145
scheduler1.ref	scheduler0.mch	0.00	”
scheduler1_6.ref	scheduler0_6.mch	3.80	37,009
m1.ref	m0.mch*	25.4	585
m1.ref	m0.mch	6.28	”
m2.ref	m0.mch	8.10	785
m2_err.ref	m0.mch	8.13	809
m2.ref	m1.ref	70.57	3,804
m3.ref	m0.mch	51.96	5,345
m3.ref	m1.ref	429.37	24,039
m3.ref	m2.ref	333.85	21,205
Counter examples found			
scheduler1_err.ref	scheduler0.mch*	0.12	19
scheduler1_6_err.ref	scheduler0_6.mch*	1.80	121
m1.ref	m2.ref	0.01	13
m2_err.ref	m1.ref*	4.22	92
m2_err.ref	m1.ref	0.03	”

Table 3 ProB refinement checking using XSB Prolog

Refinement	Specification	Checking time	Total time
Successful refinements			
ServerR.ref	Server.mch	0.00 s	0.06 s
scheduler1.ref	scheduler0.mch	0.00 s	0.11 s
m1.ref	m0.mch	2.85 s	13.76 s
m2.ref	m1.ref	26.66 s	40.24 s
m3.ref	m2.ref	136.12 s	219.03 s
Counter examples found			
m1.ref	m2.ref	0.00 s	22.68 s
m2_err.ref	m1.ref	0.01 s	12.79 s

7.3 Other case studies

7.3.1 Volvo vehicle function

We have tried our tool on a case study performed at Volvo on a typical vehicle function. The B specification machine had 15 variables, 550 lines of B specification, and 26 operations. The invariant consisted of 40 conjuncts. This B specification was developed by Volvo as part of the European Commission IST Project PUSSEE (IST-2000-30103).

We first used ProB to animate the B machine, which worked very well. The machine was already finite state (apart from an auxiliary natural number variable which was used to make proofs possible). We then used ProB to verify the B-machine using the automatic consistency checker. ProB managed to explore the entire state space of the B-machine in a few minutes, covering 1,360 states and 25,696 transitions, thereby proving the absence of invariant violations and deadlocks. This was achieved in 34.3 s (on a PowerMac G5 Dual 2.7 GHz). However, ProB managed to identify a slight anomaly in the B machine's behaviour: a crucial operation was only enabled in 8 of the 1,360 states. This shows that ProB might be used to identify problems that would otherwise only emerge at implementation time.

To better test the model checkers, we also injected a subtle fault into the specification, which the automatic consistency checker managed to unveil fully automatically within a couple of seconds (on a PowerMac G5 Dual 2.7 GHz).

7.3.2 E-travel agency

Within our ABCD[13] project we developed various B models for a distributed online travel agency, through which users can make hotel and car rental bookings. The models were developed jointly with a Java/JSP implementation. The B model contains about 6 pages of B and, as can be seen in Fig. 6 earlier, has 11 variables of complicated type.

ProB was very useful in the development of the specification, and was able to animate all of our models properly (see Fig. 6) and discover several problems with various versions of our system. For example, it was able to discover an invariant violation, meaning that two cars could be booked in a single transaction, which was not allowed by the invariant of that machine.

7.3.3 Nokia NoTA case study

Within the RODIN Project[14] the ProB tool has been used in conjunction with the AtelierB theorem prover for the validation and verification of Nokia's NoTA hardware platform. This platform is a WebServices/Corba-like interconnect network that allows hardware and software based services to communicate. This case study was highly successful. To quote from a personal communication by Ian Oliver of Nokia:

> ProB also provides a simple way of explaining and demonstrating the mathematical specification to persons who would normally not be able to read such a specification (particularly managers). The ability for the customer of a system to interact with the specification is of enormous value in that the customer can obtain a much clearer understanding of what work is being done, how it is progressing and equally importantly, what the customer really wants.

8 Combining B and CSP in ProB

In the Event-B approach [1], a B machine is viewed as a reactive system that continually executes enabled operations in an interleaved fashion. This allows parallel activity to be easily modelled as an interleaving of operation executions. However, while B machines are good at modelling parallel activity, they can be less convenient for modelling sequential activity. Typically one has to introduce an abstract 'program counter' to order the execution of actions. This can be much less transparent than the way in which one orders action execution in process algebras such as CSP [32]. CSP provides operators such as sequential composition, choice and parallel composition of processes, as well as synchronous communication between parallel processes.

The motivation is to use CSP and B together in a complementary way. B can be used to specify abstract state and can be used to specify operations of a system in terms of their enabling conditions and effect on the abstract state. CSP can be used to give an overall specification of the coordination of operations. To marry the two approaches, we take the view that the execution of an operation in a B machine corresponds to an event in CSP terms. Semantically we view a B machine as a process that can engage in events in the same way that a CSP process can. The meaning of a combined CSP and B specification is the parallel composition of both specifications. The B machine and the CSP process must synchronise on common events, that is, an operation can only happen in the combined system when it is allowed both by the B and the CSP. There is much existing work on combining state based approaches such as B with process algebras such as CSP and we review some of that in a later section.

In [40] we presented the CIA (CSP Interpreter and Animator) tool, a Prolog implementation of CSP. As both ProB and CIA are implemented in Prolog, we were provided with

[13] "Automated validation of Business Critical systems using Component-based Design," EPSRC grant GR/M91013.

[14] http://rodin.cs.ncl.ac.uk/.

a unique opportunity to combine these two to form a tool that supports animation and model checking of specifications written in a combination of CSP and B. The combination of the B and CSP interpreters means we can apply animation, consistency checking and refinement checking to specifications which are a combination of B and CSP. For example, the mutual exclusion property of the scheduler of Sect. 2.1 can be specified as the following CSP process:

$$LOCK = enter?p \rightarrow leave.p \rightarrow LOCK.$$

We can check that both B schedulers (Figs. 1, 3) are trace refinements of the *LOCK* CSP process. We can also check whether a combined B/CSP specification is a refinement of another combined specification.

A further use of the CSP interpreter is to analyse trace properties of a B machine. In this case the behaviour is fully specified in B, but we use CSP to specify some desirable or undesirable behaviours and use ProB to find traces of the B machine that exhibit those behaviours. More details may be found in [16].

9 Related work

We are not the first to realise the potential of logic programming for animation and/or verification of specifications. See for example [14], where an animator for VERILOG is developed in Prolog, or [12] where Petri nets are mapped to CLP. Also, the model checking system XMC contains an interpreter for value-passing CCS [22,52]. A logic programming approach to encode denotational semantics specifications was applied in [37] to verify an Ada implementation of the "Bay Area Rapid Transit" controller.

The most strongly related work is [6,13], which uses a special purpose constraint solver over sets (CLPS) to animate B and Z specifications using the so-called BZ-Testing-Tools. Unfortunately, it is not possible to obtain CLPS hence we cannot perform a detailed comparison of the constraint solving facilities of the ProB kernel with CLPS. Indeed, our own B-Kernel, can be viewed as a constraint solver over finite sets and sequences (it seems that sequences are not yet supported by [6]). At a higher level, [6,13] put a lot of stress on animation and test-case generation, but do not cater for model checking. There are also many features of B that we support (such as set comprehensions, lambda abstractions, multiple machines, refinement) which are not supported by [6,13]. Finally, [6,13] can handle Z as well as B specifications, and ProB has also recently been extended for Z in [50]. In addition, we have interpreters for process languages such as CSP [40,41] and StAC [26]. These can now be easily coupled with ProB to achieve an integration like [18], where B describes the state and operations of a system and where the process language describes the sequencing of the individual operations.

Another constraint solver over sets is CLP($\mathcal{SET}$) [23].[15] While it does not cater for sequences or relations, we plan to investigate whether CLP($\mathcal{SET}$) can be used to simplify the implementation of ProB. Still, it is far from certain whether CLP($\mathcal{SET}$) will be flexible enough for constraint-based checking.

Bellegarde et al. [10] described the use of SPIN to verify that finite B machines satisfy LTL properties (though the translation from B to SPIN does not appear to be automatic). This differs from the ProB approach in that it does not check for standard B invariant violation, rather it checks for satisfaction of LTL properties, which are not part of standard B.

A very recent animator for B is the commercial Brama tool [58] by ClearSy. It provides a very sophisticated interface, along with support for custom Flash animations.[16] However, Brama cannot be used for model checking and it can only animate a restricted subset of B. E.g., a substitution of the form `ANY x where x:NATURAL & x<10 THEN y:=x END` cannot be animated using Brama.

Other related work is [63], which presents an animator for Z implemented in Mercury. Mercury lacks the (dynamic) co-routining facilities of SICStus Prolog, and [63] uses a preliminary mode inference analysis to figure out the proper order in which B-Kernel predicates should be put. It is unclear to us whether such an approach will work for more involved B machines. Another animator for Z is ZANS [35]. It has been developed in C++ and unlike ProB only supports deterministic operations (called explicit in [35]), and has not been updated since 1996 [62].

The Possum [29] tool provides an animator for SUM, an extension of Z. Possum distinguishes between predicates which are "checks" and "chests," where chests can provide values for variables whereas checks can only be true or false. Possum works by simplification of predicates, and attempts to simplify chests with smaller projected sizes. Possum has also been used in [54] to animate refinements. However, the details provided in [29] do not allow for a precise comparison with our approach. It seems that Possum does not yet support set comprehensions and existential variables [62]. Staying with Z, one has to see how the recent Jaza animator [62] and the CZT Z community tools [48] will develop.

The Alloy language and analyzer developed by Jackson [34] provides a powerful framework for system modelling and analysis. Like B, the Alloy language is founded on set theory and logic. Alloy models contain signatures representing state space as well as operations and assertions. Typically

[15] There are many more constraint solvers over sets; but most of them require sets to be fully instantiated or at least have fixed, pre-determined sizes, cf. [23].

[16] Flash animations have also been added to ProB in [11].

assertions state invariant preservation properties. Rather than exploring the reachable states of a model as in ProB, the Alloy analyser uses SAT solvers to find counter-examples to assertions. The analyser uses symmetry breaking techniques to reduce the search required in SAT solving. In the meantime, symmetry reduction techniques have also been added to ProB [44,61], and have turned out to provide big speed improvements.

The idea of using (tabled) logic programming for verification is not new (see, e.g., [53]). The inspiration for the current refinement checker came from the earlier developed CTL model checker presented in [46]. Another related work is [9], which presents a bisimulation checker written in XSB Prolog. Compared to mainstream model checkers such as Spin [33] or SMV [15,49] the difficulty in model checking B actually lies more in checking the invariant and computing the individual enabled operations along with their parameters, results, and effects.

10 Conclusion

We have presented the ProB toolset for animation, consistency checking and refinement for the B method. Our experience is that ProB is a valuable complement to the usual theorem prover based development in B. Wherever possible there is value in applying model checking to a size-restricted version of a B model before attempting semi-automatic deductive proof. While it still remains to be seen how ProB will scale for very large B machines, we have demonstrated its usefulness on medium sized specifications. ProB is being used by our industrial collaborators and we have had positive feedback from them on its value. We also believe that ProB could be a valuable tool to teach beginners the B method, allowing them to play and debug their first specifications. ProB has and is being used at various universities to teach B (e.g., the University of Franche-Comté, Heinrich-Heine Universität Düsseldorf, the University of Southampton, the University of Surrey). ProB's animation facilities have allowed our users to gain confidence in their specifications, and has allowed them to uncover errors that were not easily discovered by Atelier-B. ProB's model checking capabilities have been even more useful, finding non-trivial counter examples and allowing one to quickly converge on a consistent specification.

References

1. Abrial, J.-R., Mussat, L.: Introducing dynamic constraints in B. In: Bert, D. (ed.) B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22–24, 1998, Proceedings. LNCS, vol. 1393. Springer, Heidelberg (1998)
2. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
3. Abrial, J.-R.: Case study of a complete reactive system in Event-B: a mechanical press controller. In: Tutorial at ZB'2005, http://www.zb2005.org/ (2005)
4. Abrial, J.-R., Butler, M., Hallerstede, S.: An open extensible tool environment for Event-B. In: ICFEM06. LNCS, vol. 4260. Springer, Heidelberg (2006)
5. Abrial, J.-R., Cansell, D.: Click'n prove: interactive proofs within set theory. In: Basin, D.A., Wolff, B. (eds) TPHOLs. LNCS, vol. 2758, pp. 1–24. Springer, Heidelberg (2003)
6. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Utting, M., Vacelet, N.: 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, August 2002. Technical Report, INRIA, pp. 105–120 (2002)
7. AT&T Labs-Research. Graphviz—open source graph drawing software. http://www.research.att.com/sw/tools/graphviz/
8. UK B-Core (UK) Limited, Oxon: B-Toolkit, On-line manual, 1999. http://www.b-core.com/ONLINEDOC/Contents.html
9. Basu, S., Mukund, M., Ramakrishnan, C.R., Ramakrishnan, I.V., Verma, R.M.: Local and symbolic bisimulation using tabled constraint logic programming. In: International Conference on Logic Programming (ICLP), Paphos, Cyprus, November 2001. LNCS, vol. 2237, pp. 166–180. Springer, Heidelberg (2001)
10. Bellegarde, F., Julliand, J., Mountassir, H.: Model-based verification through refinement of finite B event systems. In: Robinson, K., Bert, D. (eds), Formal Methods'99 B User Group Meeting, Toulouse, France, September 1999. Springer, Heidelberg (1999)
11. Bendisposto, J., Leuschel, M.: A Generic Flash-Based Animation Engine for ProB. In: Julliand and Kouchnarenko [36], pp. 266–269 (1999)
12. Bérard, B., Fribourg, L.: Reachability analysis of (timed) petri nets using real arithmetic. In: Proceedings of Concur'99, LNCS, vol. 1664, pp. 178–193. Springer, Heidelberg (1999)
13. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B — a constraint solver for B. In: Katoen, J.-P., Stevens, P. (eds), Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 2280 , pp. 188–204. Springer, Heidelberg (2002)
14. Bowen, J.: Animating the semantics of VERILOG using Prolog. Technical Report UNU/IIST Technical Report no. 176, United Nations University, Macau (1999)
15. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Informat. Computat. **98**(2), 142–170 (1992)
16. Butler, M., Leuschel, M.: Combining CSP and B for specification and property verification. In: Proceedings of Formal Methods 2005, Newcastle upon Tyne, LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
17. Butler, M.J.: An approach to the design of distributed systems with B AMN. In: Bowen, J.P., Hinchey, M.G., Till, D.,(eds), ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April, 1997, Proceedings, LNCS, vol. 1212, pp. 223–241. Springer, Heidelberg (1997)
18. Butler, M.J.: csp2B: a practical approach to combining CSP and B. Formal Asp. Comput. **12**(3), 182–198 (2000)
19. Carlsson, M., Ottosson, G.: An open-ended finite domain constraint solver. In: Glaser, H.G., Hartel, P.H., Kuchen, H. (eds), Proc. Programming Languages: Implementations, Logics, and Programs, LNCS, vol. 1292 pp. 191–206. Springer, Heidelberg (1997)
20. Clarke, E.M.: Orna Grumberg, and Doron Peled. Model Checking. MIT Press, (1999)
21. ClearSy. Atelier B, User and Reference Manuals, Available at http://www.atelierb.societe.com/index_uk.html

22. Cui, B., Dong, Y., Du, X., Narayan Kumar, C.R., Ramakrishnan, I.V., Ramakrishnan, Roychoudhury, A., Smolka, S.A., Warren, D.S.: Logic programming and model checking. In: Palamidessi, C., Glaser, H., Meinke, K., (eds) Proceedings of ALP/PLILP'98, LNCS, vol. 1490 pp. 1–20. Springer, Heidelberg (1998)
23. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Transactions on Programming Languages and Systems (TOPLAS), **22**(5), 861–931 (2000)
24. Dunne, S., Conroy, S.: Process refinement in B. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds), ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, LNCS, vol. 3455, pp. 45–64. Springer, Heidelberg (2005)
25. Farwer, B., Leuschel, M.: Model checking object Petri nets in Prolog. In PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming, pages 20–31, New York, NY, USA, 2004. ACM Press (2004)
26. Ferreira, C., Butler, M.: A process compensation language. In Santen, T., Stoddart, B. (eds), Proceedings Integrated Formal Methods (IFM 2000), LNCS, vol. 1945 pp. 424–435. Springer, November 2000 (2000)
27. Formal Systems (Europe) Ltd. Failures-Divergence Refinement — FDR2 User Manual. Available at http://www.fsel.com/
28. Gardiner, P.H.B., Morgan, C.: A single complete rule for data refinement. Formal Asp. Comput. **5**(4), 367–382 (1993)
29. Hazel, D., Strooper, P., Traynor, O.: Possum: An animator for the SUM specification language. In Proceedings Asia-Pacific Software Engineering Conference and International Computer Science Conference, pages 42–51. IEEE Computer Society, (1997)
30. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds), ESOP 86, European Symposium on Programming, Saarbrücken, Federal Republic of Germany, March, 1986, Proceedings, LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
31. Henderson, P.: Modelling architectures for dynamic systems. In: McIver, A., Morgan, C. (eds) Programming Methodology. Springer, Heidelberg (2003)
32. Hoare, C.A.R: Communicating Sequential Processes. Prentice–Hall, Englewood Cliffs (1985)
33. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, 2001
34. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, (2006)
35. Jia, X.: An approach to animating Z specifications. Available at http://venus.cs.depaul.edu/fm/zans.html
36. Julliand, J., Kouchnarenko, O. (eds.): B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17–19, 2007, Proceedings, volume 4355 of Lecture Notes in Computer Science. Springer
37. King, L., Gupta, G., Pontelli, E.: Verification of a controller for BART. In: Winter, V.L., Bhattacharya, S. (eds), High Integrity Software, pp. 265–299. Kluwer Academic Publishers, New York (2001)
38. Knuth, D.: All questions answered. Notices of the ACM **49**(3), 318–324 (2003)
39. Lanet, J.-L.: The use of B for Smart Card. In Forum on Design Languages (FDL02) (2002)
40. Leuschel, M.: Design and implementation of the high-level specification language CSP(LP) in Prolog. In: Ramakrishnan, I.V. (ed), Proceedings of PADL 01, LNCS, vol. 1990, pp. 14–28. Springer, Heidelberg (2001)
41. Leuschel, M., Adhianto, L., Butler, M., Ferreira, C., Mikhailov, L.: Animation and model checking of CSP and B using Prolog technology. In Proceedings of VCL 2001, pp. 97–109, Florence, Italy, September (2001)
42. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds), Proceedings FME 2003, Pisa, Italy, LNCS vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
43. Leuschel, M., Butlerm, M.: Automatic refinement checking for B. In: Lau, K.-K., Banach, R. (eds), Proceedings ICFEM'05, LNCS, vol. 3785, pp. 345–359. Springer, Heidelberg (2005)
44. Leuschel, M., Butler, M., Spermann, C., Turner, E.: Symmetry reduction for B by permutation flooding. In: Julliand J., Kouchnarenko, O. (eds) B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17–19, 2007, Lecture Notes in Computer Science, Proceedings, vol. 4355 pp. 79–93. Springer, Heidelberg (2006)
45. Leuschel, M., Cansell, D., Butler, M.: Validating and animating higher-order recursive functions in B. In: Abrial, J.-R., Glässer, U. (eds), Festschrift for Egon Börger, May (2006)
46. Leuschel, M., Massart, T.: Infinite state model checking by abstract interpretation and program specialisation. In: Bossi, A. (ed.) Logic-Based Program Synthesis and Transformation. Proceedings of LOPSTR'99, Venice, Italy, 2000. LNCS, vol. 1817, pp. 63–82. Springer, Heidelberg (2000)
47. Leuschel, M., Turner, E.: Visualizing larger states spaces in ProB. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds), Proceedings ZB'2005, LNCS, vol. 3455, pp. 6–23. Springer, Heidelberg (2005)
48. Malik, P., Utting, M.: CZT: A framework for Z tools. In: Treharne, H., King, S., Henson, M.C., Schneider, S.A. (eds), ZB, LNCS, vol. 3455, pp. 65–84. Springer, Heidelberg (2005)
49. McMillan, K.L.: Symbolic model checking. PhD thesis, CMU (1993)
50. Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In Jim Davies and Jeremy Gibbons, editors, IFM, Lecture Notes in Computer Science, vol. 4591, pp. 480–500. Springer, Heidelberg (2007)
51. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University (1981)
52. Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S.: Efficient model checking using tabled resolution. In: Grumberg, O. (ed), Proceedings of the International Conference on Computer-Aided Verification (CAV'97), LNCS, vol. 1254, pp. 143–154. Springer, Heidelberg (1997)
53. Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.N.: XMC: A logic-programming-based verification toolset. In Proceedings of CAV 2000, pp. 576–580 (2000)
54. Robinson, N.J., Fidge, C.J.: Animation of data refinements. In Asia-Pacific Software Engineering Conference, APSEC 2002, pp. 137–146. IEEE Computer Society (2002)
55. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice–Hall, Englewood Cliffs (1998)
56. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: Proceedings of the ACM SIGMOD International Conference on the Management of Data, Minneapolis, Minnesota, May 1994, pp. 442–453. ACM (1994)
57. Schneider, S., Treharne, H.: Communicating B machines. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds), ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, LNCS, vol. 2272, pp. 416–435. Springer, Heidelberg (2002)
58. Servat T BRAMA: a new graphic animation tool for B models. In: Julliand and Kouchnarenko [36], pp. 274–276
59. Sweden SICS, Kista: SICStus Prolog User's Manual. http://www.sics.se/sicstus
60. Bruno Tatibouet: The jbtools package. http://lifc.univ-fcomte.fr/PEOPLE/tatibouet/JBTOOLS/BParser_en.html, 2001

61. Turner, E., Leuschel, M., Spermann, C., Butler, M.: Symmetry reduced model checking for B. In First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, June 5–8, 2007, Shanghai, China. IEEE Computer Society (2007)
62. Utting, M.: Data structures for Z testing tools. In FM-TOOLS 2000 Conference, July 2000, TR 2000-07, Information Faculty, University of Ulm (2000)
63. Winikoff, M., Dart, P., Kazmierczak, E.: Rapid prototyping using formal specifications. In: Proceedings of the 21st Australasian Computer Science Conference, Perth, Australia, February 1998, pp. 279–294 (1998)