ProB: A Model Checker for B

Michael Leuschel and Michael Butler

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{mal,mjb}@ecs.soton.ac.uk

Abstract. We present ProB, an animation and model checking tool for the B method. ProB's animation facilities allow users to gain confidence in their specifications, and unlike the animator provided by the B-Toolkit, the user does not have to guess the right values for the operation arguments or choice variables. ProB contains a model checker and a constraint-based checker, both of which can be used to detect various errors in B specifications. We present our first experiences in using ProB on several case studies, highlighting that ProB enables users to uncover errors that are not easily discovered by existing tools.
Keywords: B-Method, Tool Support, Model Checking, Animation,Logic Programming, Constraints.

1 Introduction

The B-method, originally devised by J.-R. Abrial [1], 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.

B is based on the notion of *abstract machine* and the notion of *refinement*. The variables of an abstract 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 assignments to be specified. In B refinement, a machine may be refined by another machine in which the state is represented by data structures that are more concrete and/or in which operations are more deterministic and imperative.

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. A refinement that is at a sufficiently low level can be translated into code. These activities are supported by industrial strength tools, such as Atelier-B [33] and the B-toolkit [4]. A B-tool generates a list of predicate logic proof obligations (POs). If each of these POs is proved, then the machine is consistent (or a correct refinement in the case of refinement checking). The B-tools have an automatic prover and an interactive prover. Typically the more complex POs are not proved automatically and need to be proved interactively. The tools also provide automatic translation of low level B specifications into executable code.

The ProB tool introduced in this paper currently supports automated consistency checking of B machines via *model checking* [12]. For exhaustive model checking, the given sets must be restricted to small finite sets, and integer variables must be restricted to small numeric ranges. This allows the checking to traverse all the reachable states of the machine. ProB can also be used non-exhaustively to explore the state space and find potential problems. The user can set an upper bound on the number of states to be traversed or can interrupt the checking at any stage. ProB will generate and graphically display counter-examples when it discovers a violation of the invariant. ProB detects attempts to evaluate undefined expressions, such as the application of a partial function to arguments outside its domain. ProB can also be used as an animator of B specifications. So, the model checking facilities are still useful for infinite state machines, not for verification, but for sophisticated debugging and testing.

ProB also offers an alternative checking method, inspired by the ALLOY [18, 19] analyzer. In this mode of operation, ProB does not explore the reachable states starting from the initial state(s), but checks whether applying an individual operation can result in an invariant violation, independently of the particular initialization of the B machine. This is done by symbolic constraint solving, and we call this approach *constraint-based checking* (another sensible name would be *model finding*).

Possible applications of ProB: For finite state B machines it may be possible to use ProB for proving consistency without user intervention (cf. our case study in Sect. 8). However, we believe that ProB will be more useful as a complement to the current tools. Indeed, the interactive proof process with Atelier-B or the B-Toolkit can be quite time consuming: a typical development involves going through several levels of refinement to code generation *before* attempting any interactive proof [22]. This is to avoid the expense of reproving POs as the specification and refinements change in order to arrive at a satisfactory implementation. We therefore see one of the main uses of ProB as a complement to interactive proof, in that some errors will be discovered earlier in the development cycle and also that there will be less effort wasted by users trying to prove incorrect POs. We also believe that ProB will be very useful in teaching B, and making B more accessible to new users. Finally, even for experienced B users ProB may unveil problems in a specification that are not easily discovered by existing tools.

We proceed with an illustration of the use of ProB before continuing to describe its design.

2 Using ProB

ProB provides two ways of discovering whether a machine violates its invariant:

1. it can find a sequence of operations that, starting from a valid initial state of the machine, navigates the machine into a state in which the invariant

```
MACHINE Lift
VARIABLES  floor
INVARIANT  floor : 0..99
INITIALISATION floor := 4
OPERATIONS
     inc = PRE floor<99 THEN floor := floor + 1 END ;
     dec = BEGIN floor := floor - 1 END
END
```

Fig. 1. Lift example in B

is violated. Trying to find such a sequence of operations is the task of the ProB *(temporal) model checker*.

2. it can construct a state of the machine which satisfies the invariant, but from which we can apply a *single* operation to reach a state which violates the invariant. Finding such states is the task of the ProB *constraint-based checker*.

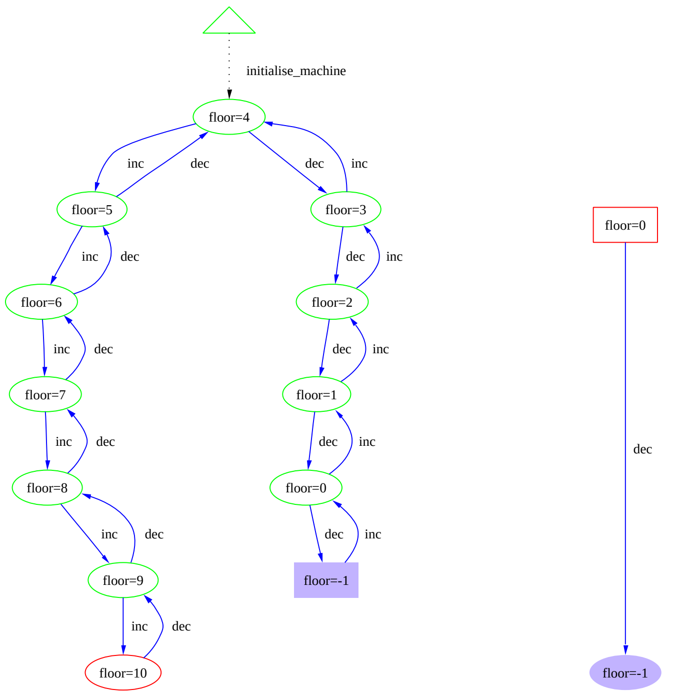
Let us examine how these approaches work on a simple example. Figure 1 presents a very simple B specification of a lift, which has an operation `inc` to go up one floor, and an operation `dec` to go down one floor.

This B machine does not preserve its invariant and Fig. 2 presents two counter-examples found by ProB. The left one (a) is produced by the model checker and shows that it is possible to reach a state where the invariant is violated, i.e., the `floor` variable becomes negative. (Usually ProB will only display the sequence of operations and states that lead to an invariant violation, but for Fig. 2 we have used ProB to display all the states that were explored until the invariant violation was found.) The right one (b) is produced by the constraint-based checker, which has constructed a before state `floor = 0` and found the operation `dec` which applied to that state yields an invariant violation in the after state. Note that in this case there is no guarantee that the constructed before state is actually reachable from the initial state(s), although in this particular example it is. Also note that the constraint-based checker can determine that the `inc` operation will never introduce an invariant violation on its own. Finally, the constraint-based checker can also be used to find abort conditions, and the model checker can be used to both detect abort conditions and deadlocks.

3 Brief Overview of the System

The ProB system has been developed mainly in SICStus Prolog, with a graphical user interface implemented in Tcl/Tk. An overview of ProB's main components can be found in Fig. 3.

The first implementation problem to be overcome is the translation of specifications written in B abstract machine notation (AMN) [1] into a notation convenient for interpretation within Prolog. ProB uses the JBTools package developed by Tatibouet [34] and the Pillow package [10] to that effect. The JBTools package permits translation of AMN specifications into XML, while



(a) (b)

Fig. 2. Counter-examples for the Lift Machine

the Pillow package allows the conversion of XML files into a Prolog term representation. The ProB front end then postprocesses the general Prolog term tree representation of the Pillow library output into a more structured representation which will serve as the input to the interpreter. The ProB *interpreter* recurses through this structured representation of B machines and makes call to the ProB *kernel*, which provides support for the basic datatypes and operations of the B-language. The ProB kernel itself is written in SICStus Prolog with co-routining and constraints. The ProB animator and the two model checkers all make use of the ProB interpreter in various ways, as will be explained later in the paper.

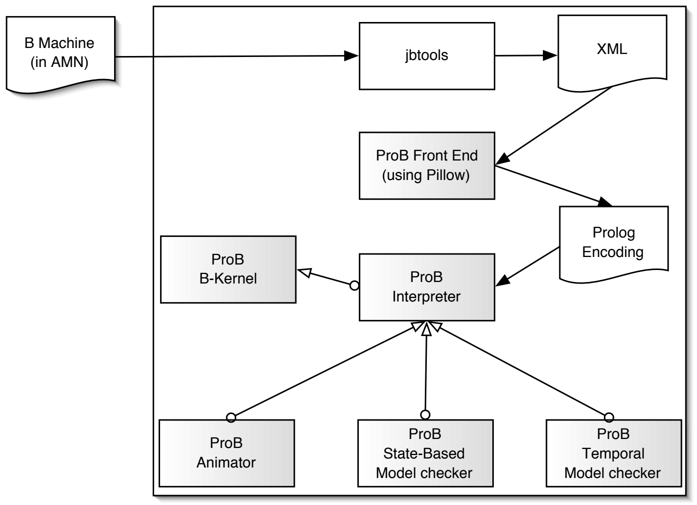

B Machine (in AMN)
jbtools
XML
ProB Front End (using Pillow)
Prolog Encoding
ProB B-Kernel
ProB Interpreter
ProB Animator
ProB State-Based Model checker
ProB Temporal Model checker

Fig. 3. Overview of the ProB System

4 The ProB Kernel and Interpreter

The ProB Interpreter The ProB interpreter is written in a structured operational semantics [28] (SOS) style. More precisely, given a description σ_1 of the state of a B machine, we describe which operations (and with which argument 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. *statements* which modify the variables of a B machine,
2. *expressions* which do not modify the variables but return values, and
3. *boolean expressions*, called predicates in B, which are expressions which return either true or false.

To manipulate these constructs the ProB interpreter provides Prolog predicates[1] to execute statements, compute expressions, and test boolean expressions. Each one of these predicate has access to the global state of the machine (the state of the variables of a machine) and a local state which contains the values for local variables and parameters of operations. In order to manipulate B's basic datastructures and operators, the ProB interpreter calls the ProB kernel, which we discuss later.

[1] Note that there is a potential confusion concerning the use of the word "predicate" in B and in Prolog. From now on, when we use the term predicate, we mean a predicate in our Prolog implementation and not a boolean expression in a B machine.

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]), LocalState,State) :-
        b_test_boolean_expression(LHS,LocalState,State),
        b_test_boolean_expression(RHS,LocalState,State).
b_test_boolean_expression('Or'(_,[LHS,RHS]), LocalState,State) :-
        (b_test_boolean_expression(LHS,LocalState,State)  ;  /* or */
         (b_not_test_boolean_expression(LHS,LocalState,State),
          b_test_boolean_expression(RHS,LocalState,State)))
```

The first argument of the predicate is the boolean expression to be tested.[2] The second argument (`LocalState`) of the predicate `b_test_boolean_expression` 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 of the B machine under consideration. The `b_test_boolean_expression` predicate also has a counterpart, which is called `b_not_test_boolean_expression` and 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.

While it is clearly difficult to cover the vast syntax of B, the code of the ProB interpreter is relatively simple. The reason is that the ProB kernel is very flexible and "hides" a lot of the complexities of B from the ProB interpreter. In fact, while the ProB interpreter is written in classical Prolog the ProB kernel uses advanced features of Prolog, such as co-routining and constraints. We discuss the ProB kernel in more detail below. Also, because of Prolog's support for non-determinism it was not too difficult to support non-deterministic operations.

The ProB **Kernel** First, let us see how some of B's datastructures are actually encoded by the ProB Kernel:

B Type	B value	Prolog encoding
number	5	`int(5)`
boolean	true	`term(true)`
element of a finite set S	C	`fd(3,'S')`
pair	(2,5)	`(int(2),int(5))`
sequence	$[2,5]$	`cons(int(2),cons(int(5),nil))`
set	$\{2,5\}$	`[int(2), int(5)]`

As you can see, sets are represented by Prolog lists.[3] The Prolog term `fd(3,'S')` means that we are dealing with the third constant of the given 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. There is also a special symbol `abort` reserved to indicate that an abort condition occurred while trying to compute this value. Also note

[2] This expression also contains some XML layout information, e.g., in the first argument of the 'And' symbol, which is ignored by the interpreter.

[3] To be able to detect typing errors, B lists are not represented as standard Prolog lists but use the special `cons` symbol instead.

that the above types can in principle be arbitrarily nested, e.g., we may have sets of sets of sequences of pairs.

There is nothing special about this mapping from B to Prolog, and one may wonder where the difficulties in writing an interpreter lie. To highlight these difficulties, let us examine the following non-deterministic B operation, that finds a symmetric partial function on a finite set `Name`:

```
cc <-- symmetric = ANY nn WHERE nn = nn~ & nn: Name +-> Name
                       THEN cc := nn    END;
```

Now, when the ProB interpreter reaches the construct `nn = nn~` it does not yet have any information about `nn`. This will only be provided "later" when interpreting the construct `nn: Name +> Name`. When writing an interpreter for a "classical" programming language things are much simpler: within a statement we typically would know the type and value of any variable. But in B this is not the case, and the value of a variable might depend on many constraints, some of which may not yet have been encountered by the ProB interpreter. We have overcome this problem by using the co-routining facilities of SICStus Prolog, which allow one to suspend goals until sufficient information is available to evaluate them. For example, the inversion operator `~` is implemented by a binary Prolog predicate `invert_relation`, which will automatically suspend until enough information is available. This is done as follows:

```
invert_relation(R,IR) :- when((nonvar(R);nonvar(IR)),inv_rel2(R,IR)).
inv_rel2([],[]).
inv_rel2([(X,Y)|T],[(Y,X)|IT]) :-
               when((nonvar(T);nonvar(IT)),inv_rel2(T,IT)).
```

The `when` co-routining predicate will suspend until its first argument becomes true, in which case it will then 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.

The co-routining has made the `invert_relation` much more robust and usable. As the following two queries show, it can now handle information that is incrementally provided about its arguments (both of these queries would have looped without the when declarations):

```
| ?- invert_relation(R,R), R=[(int(1),int(2))].
no
| ?- findall(R,(invert_relation(R,R), R=[(X,int(2))]),Answers).
Answers = [[(int(2),int(2))]] ?
yes
```

When the ProB interpreter encounters the expression `nn = nn~` in the above B operation `symmetric`, it would basically call the ProB kernel as follows: `invert_relation(NN,InvN),equal_object(NN,InvN)`.[4] Now, the ProB kernel will

[4] One has to use ProB's `equal_object` predicate instead of Prolog unification because the same B set can be represented by different Prolog lists.

not compute any value for the variable `NN`, it will simply suspend and wait for `NN` to be further instantiated. So, how does ProB get concrete values for the `ANY` statement?

To understand this, we have to examine how the kernel treats the expression `nn: Name +> Name`. In fact, any global set of the B machine, such as `Name`, will be mapped to a finite domain within SICStus Prolog's CLP(FD) constraint solver [11]. A finite domain within CLP(FD) is a finite set of integers, typically an interval. CLP(FD) provides a wide variety of constraints that can be expressed on such domains, and it provides a way of enumerating all concrete solutions (called *labeling*).

For example, supposing that `Name` is mapped to the finite domain $\{1, 2\}$, the expression `n: Name` will be mapped to the SICStus Prolog code `N = fd(C,'Name')`, `C in 1..2`, where `C in 1..2` is a CLP(FD) constraint.

To force the enumeration of concrete values and thus force the execution of suspended goals we use CLP(FD)'s labeling operation.[5] Note that this enumeration is only used as a last resort: sometimes operations can be fully evaluated without having to enumerate at all.

Note, for our approach to work we have to be sure that we will not generate infinitely many solutions or candidate solutions for an `ANY` statement. This is achieved by requiring that every choice variable of an `ANY` statement is given a finite type. For example, `ANY x WHERE x:NAT THEN` is not supported by ProB. However, the above operation `symmetric` is supported by ProB, and all possible symmetric partial functions over `Name` will be generated by the interpreter.

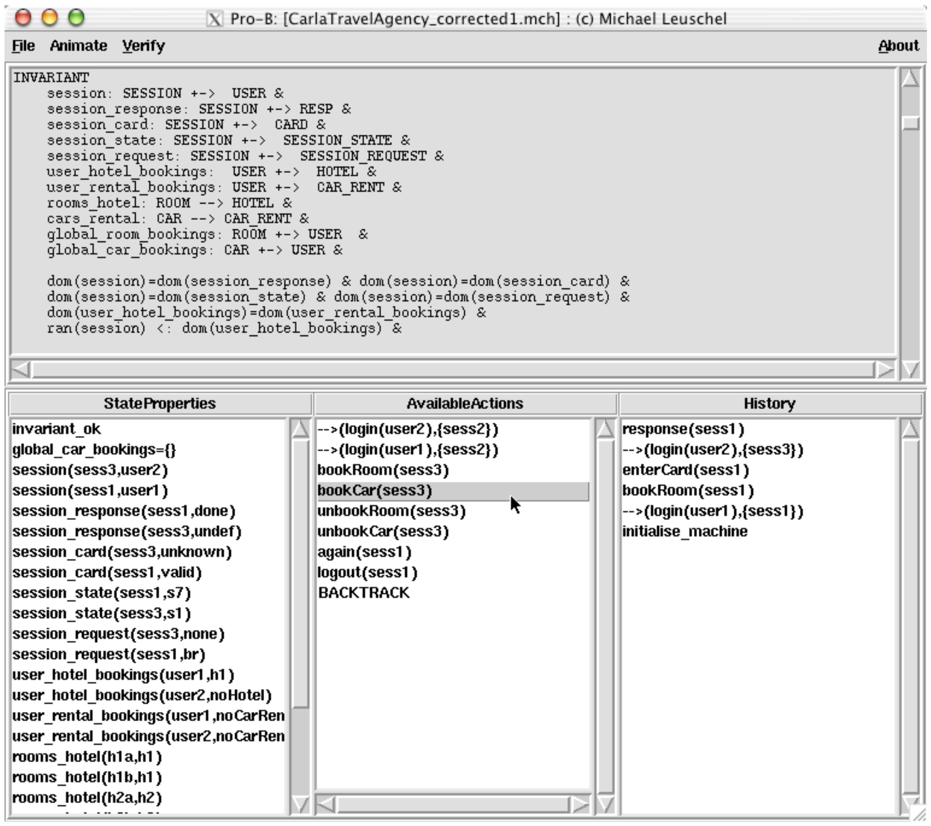
5 The ProB Animator

The ProB animator was developed using the Tcl/Tk library of SICStus Prolog 3.10. The user interface was inspired by the ARC tool [17] for system level architecture modelling and builds upon our earlier animator for CSP [23].

Our animator supports (backtrackable) step-by-step animation of B-machines, and supports non-deterministic operations. As can be seen in Fig. 4 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 `ANY` statements, the user does not have to find values which satisfy the `ANY` statement. If the number of enabled operations becomes larger, one could envisage a more refined interface were not all options are immediately displayed to the user.

To extract all possible values for operation inputs and choice variables from the ProB interpreter, the ProB animator uses Prolog's `findall` construct together with the CLP(FD) labelling operation. For this to work properly, we re-

[5] For more complicated types we may actually have to use the *hypercall* primitive discussed later in Sect. 6.


Pro-B: [CarlaTravelAgency_corrected1.mch] : (c) Michael Leuschel
File
Animate
Verify
About
StateProperties
AvailableActions
History

```
INVARIANT
    session: SESSION +->  USER &
    session_response: SESSION +-> RESP &
    session_card: SESSION +->  CARD &
    session_state: SESSION +->  SESSION_STATE &
    session_request: SESSION +->  SESSION_REQUEST &
    user_hotel_bookings:  USER +->  HOTEL &
    user_rental_bookings: USER +->  CAR_RENT &
    rooms_hotel: ROOM --> HOTEL &
    cars_rental: CAR --> CAR_RENT &
    global_room_bookings: ROOM +-> USER  &
    global_car_bookings: CAR +-> USER &

    dom(session)=dom(session_response) & dom(session)=dom(session_card) &
    dom(session)=dom(session_state) & dom(session)=dom(session_request) &
    dom(user_hotel_bookings)=dom(user_rental_bookings) &
    ran(session) <: dom(user_hotel_bookings) &
```

StateProperties	AvailableActions	History
invariant_ok	-->(login(user2),{sess2})	response(sess1)
global_car_bookings={}	-->(login(user1),{sess2})	-->(login(user2),{sess3})
session(sess3,user2)	bookRoom(sess3)	enterCard(sess1)
session(sess1,user1)	bookCar(sess3)	bookRoom(sess1)
session_response(sess1,done)	unbookRoom(sess3)	-->(login(user1),{sess1})
session_response(sess3,undef)	unbookCar(sess3)	initialise_machine
session_card(sess3,unknown)	again(sess1)	
session_card(sess1,valid)	logout(sess1)	
session_state(sess1,s7)	BACKTRACK	
session_state(sess3,s1)		
session_request(sess3,none)		
session_request(sess1,br)		
user_hotel_bookings(user1,h1)		
user_hotel_bookings(user2,noHotel)		
user_rental_bookings(user1,noCarRen		
user_rental_bookings(user2,noCarRen		
rooms_hotel(h1a,h1)		
rooms_hotel(h1b,h1)		
rooms_hotel(h2a,h2)		

Fig. 4. Animation of the E-Travel Agency Case Study (c.f., Sect. 8)

quire that all operation arguments are mapped finite types. For example, while it is admissible to have an operation:

```
add(nn) = PRE nn:0..10  THEN n := n +nn END
```

it is not allowed to have untyped or unbounded operation arguments, such as:

```
addinf(nn) = PRE nn:NAT  THEN n := n +nn END
```

The same holds for set assignments, i.e., it is allowed to use `x:: BOOL` but it is not allowed to use `x::NAT`. However, it would be possible to extend the animator so that it allowed such constructs, but only provided values up to a certain limit.

The ProB animator also provides visualization 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." One can also find the shortest trace (given the state space explored so far) to the current state. For the visualization we make use of the dot tool of the graphviz package [3].

Another noteworthy feature of the animator is its ability to perform *symbolic animation* as well as concrete, ordinary animation. This allows a user to trace a B-machine symbolically, without providing actual values for the parameters; the animator will set up constraints which the parameters have to satisfy (and checks whether concrete values exist which satisfy the constraints). This enables a symbolic exploration of the state space, but the user can at any time force the animator to provide concrete values. In some cases the symbolic exploration will result in a much smaller state space.

6 Consistency Checking in ProB

As we have seen in Sect. 2, ProB provides two ways of consistency checking: 1. a *model checking* which tries to find a sequence of operations that, starting from an initial state, leads to a state which violates the invariant (or exhibits some other error, such as deadlocking or abort conditions); and 2. a *constraint-based* checking, which finds a state of the machine that satisfies the invariant, but where we can apply a single operation to reach a state that violates the invariant (or again exhibits some other error).

Suppose that we have a B-machine with an incorrect invariant. In such a case proving the verification conditions will be impossible, but might not necessarily give the user feedback on why the machine is incorrect; it could even be correct but just very hard to prove.

If the model checker finds a counter-example then there is clearly a problem: a sequence of operations will lead from a valid initial state to an invariant violation, and the B machine has to be corrected. Now, if the constraint-based checker finds a counter-example then, even though the invariant violation may not reachable from a valid initial state, we also have a problem as at least one B verification condition will be wrong (i.e., in logical terms there is a model which makes the formula false), and we will never be able to prove the machine correct using the B proof rules. Take a look at the machine in Fig. 5. There is no sequence of operations that will lead to $n = 2$, but we can find the state $n = 1$ which satisfies the invariant and after applying *inc* we obtain the state $n = 2$ which violates the invariant.

```
MACHINE counter
VARIABLES n
INVARIANT n : 0..10 & n /= 2
INITIALISATION  n := 3
OPERATIONS
      inc = PRE n<10 THEN n := n + 1 END
END
```

Fig. 5. A simple counter machine with an error

From an implementation point of view, the model checking approach is simpler as every single state is clearly determined, and we can use our ProB interpreter to compute all possible successor states of any given state, and then perform a search on the right sequence of operations.

For the constraint-based approach things are more complicated. Indeed, even though we know there is only a single operation to apply, we initially have little information about the state of the B machine under consideration. One could of course try to enumerate all possible states of a B-machine, and then call the ProB interpreter to check whether any given state satisfies the invariant, and if it does call the ProB interpreter to compute and check the successor states. However, this will be highly inefficient and even very small machines will not be checkable in this way. To overcome this problem we have developed a symbolic approach, which makes use of Prolog's co-routining and constraint facilities.

Below, we present these two components of ProB in more detail. In Sect. 8, we will then show how we have successfully applied these components to verify various non-trivial machines.

Temporal Model Checking By manually exploring a B-machine it is possible to discover problems with a machine, such as invariant violations or even deadlocks (states where no operation is applicable). We have implemented a model checker which will do this 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, but the model checker can also be applied to B machines with infinite state spaces and will then explore the state space until it finds an error or runs out of memory.

To detect whether a given state has already been explored, we implemented a normalisation procedure for states. Because the temporal property we need to check (i.e., all reachable states satisfy the invariant) is a safety property [12] a relatively simple, but liberal exploration algorithm can be used. Our exploration is an adaptation of the A* algorithm with cycle detection, and can be tuned to perform in the extreme cases as either a depth-first or breadth-first exploration. In the default setting of ProB, every new node has a 25% chance of being treated in a depth-first manner, which turned out to be a good compromise in our experiments: pure depth-first search employed by many model checkers is often very bad at finding even very short counter-examples (and is not guaranteed to find counter-examples present in infinite state systems), whereas pure breadth-first is bad at finding long counter-examples.

The visited states are stored in Prolog's clause database. While this is not as efficient as for example tabling[6], it allows the model checking state to be easily

[6] A tabled logic programming systems such as XSB [31] provides very efficient datastructures 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 [29, 26].

queried (e.g., for visualization) and saved to file. Anyway, for a formalism as involved as B, most of the model checking time will be spent computing new states and the time needed to look up whether a given state has already been encountered is probably insignificant.

Constraint-Based Checking We achieved the constraint-based checking by delaying the state enumeration as long as possible. The idea is to first set up constraints which assert that the first state of the B-machine under consideration satisfies the invariant. We then apply an operation and set up constraints which assert that the invariant is no longer satisfied in the after state. In ProB this is done by the following code:[7]

```
constraint_check(OpName,State,Operation,NewState) :-
   b_extract_types_and_invariant(Variables,VarTypes,Invariant),
   b_set_up_variable_types(Variables,VarTypes,State),
   b_set_up_variable_types(Variables,VarTypes,NewState),
   b_test_boolean_expression(Invariant,[],State),
   b_not_test_boolean_expression(Invariant,[],NewState),
   b_execute_operation(OpName,Operation,State,NewState,_Abort).
```

Calling `constraint_check` will involve no enumeration and thus no search, and may still discard a big part of the search space by partially instantiating variables and arguments. However, calling `constraint_check` on its own will not yet give us any counter-example, as many of the predicates it calls will suspend.

To extract solutions from `constraint_check`, we have developed a new Prolog built-in, the so-called `hypercall` primitive. This primitive takes its argument, executes it until only suspended goals remain, applies the CLP(FD) labeling operation, and then selects one of the suspended goals for a single expansion step. It then repeats the same procedure until there are no more suspended goals. For example, while calling `invert_relation(R,R)` from Sect. 4 will suspend and not give any information about R, `hypercall(invert_relation(R,R))` will enumerate all possible solutions for R. The ProB constraint-based checking is then simply achieved by calling `hypercall(constraint_check(opname,S,Op,NS))`. If we manage to find a solution for this query, we have uncovered a counter-example to the B machine's invariant.

Which checker to use? A user could run either or both of our model checkers before attempting to prove a certain B-machine correct using existing B-tools. If our model checking tool uncovers a counter-example it is clear that proving it will be in vain. Not only have we spared the user a lot of effort, the user also has a counter-example at his or her disposal which will hopefully make it easier to correct the B-machine's specification.

[7] Observe that the variables in the after state `NewState` are given correct types by `b_set_up_variable_types`, ensuring a finite search space. We suppose that invariant violations due to type errors will be caught by a standard B type checker.

An obvious question is, when would one prefer using one approach over the other. In general, the temporal model checking approach will be more efficient, simply due to the fact that the underlying machinery is simpler and the induced overhead is smaller. However, constraint-based checking can still be more efficient for some applications, especially as we can focus on checking a single operation. This can be especially useful in circumstances where one has modified or added a single operation of a (previously verified) big machine. Also, constraint-based checking can proceed even if the initialisation operation is too restrictive or too expensive to perform exhaustively (e.g., if the initialisation chooses any state that satisfies the invariant). Finally, we believe that the constraint-based checking is easier to parallelise; something which we intend to exploit in future research.

7 Relationship with the Classical B Proof Method

In this section we outline how exhaustive temporal checking of a (finite) B machine entails standard B consistency. Consistency in B is defined by treating statements as predicate transformers. For statement S, and postcondition P, $[S]P$ represents the weakest precondition under which S, if it is enabled, is guaranteed to terminate in a state satisfying postcondition P. A machine is said to be consistent wrt an invariant I provided for each operation S_i of the machine[8]:

$$I \implies [S_i]I$$

Often an invariant I' which is stronger than I needs to be found (by adding conjuncts to I) so that

$$I' \implies I \qquad \mathit{and} \qquad I' \implies [S_i]I'$$

A successful exhaustive temporal model 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 I' above.

To show the formal correspondence between the consistency checking and exhaustive temporal checking, we make use of the set theoretic model of B which is defined in [1]. In this model, the state space of a machine is defined as the cartesian product of the types of each of the machine variables. Given a statement S, $\mathit{pre}(S)$ represents the set of states satisfying the precondition of S, while $\mathit{rel}(S)$ is the binary relation corresponding to the statement relating before states to after states. The *set transformer model* of a statement, $\mathit{str}(S)$, is a function from sets of after states to sets of before states, modelling the way in which a predicate transformer maps postconditions to preconditions. Given a set of after states q, we have [1]:

$$\mathit{str}(S)(q) \;=\; \mathit{pre}(S) \cap \overline{\mathit{rel}(S)^{-1}[\overline{q}]}$$

[8] The initialisation of a machine must also establish the invariant. Also, in standard B, the outermost precondition of an operation also appears as an antecedent to the proof obligation. This is equivalent to treating it as a guard rather than a precondition. Here we assume that the outermost precondition is included in S_i as a guard. ProB also treats the outermost precondition as a guard.

Here $R[s]$ is the relation image of set s under relation R, and $\overline{s}$ is the complement of set s. Writing I to represent the set of states satisfying the invariant, the proof obligation for consistency is characterised in the set transformer model as

$$I \subseteq str(S)(I)$$

It is relatively straightforward to show, by structural induction over the statement constructs, that the Prolog interpretation of a statement S used in ProB corresponds to $rel(S)$. A successful exhaustive temporal model check on a machine with invariant (set) I computes a set of reachable states R which is a subset of I, satisfying the following properties for each operation S_i:

$$\begin{aligned} R &\subseteq pre(S_i) \\ rel(S_i)[R] &\subseteq R \end{aligned}$$

From this, it is straightforward to prove that the machine is consistent wrt invariant set R, i.e., that for each operation i:

$$R \subseteq str(S_i)(R)$$

8 Applications and Case Studies

Volvo Vehicle Function We have tried our tool on a case study performed at Volvo on a typical vehicle function. The largest B machine had 15 variables, 550 lines of B specification, and 26 operations. This B specification was developed by Volvo as part of the European Commission IST Project PUSSEE (IST-2000-30103, `http://www.keesda.com/pussee/`).

We have 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 have then used ProB to verify the B-machine using the temporal model checker. ProB managed to explore the entire state space of the B-machine in a few minutes, covering 1360 states and 25696 transitions, thereby proving the absence of invariant violations and deadlocks. However, ProB managed to identify a slight anomaly in the B machines behaviour: a crucial operation was only enabled in 8 of the 1360 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 have also injected a subtle fault into the specification, which both the temporal and the constraint-based checker managed to unveil fully automatically within a couple of minutes.

Discussion on efficiency: Exploring 1360 states and 25696 transitions within a few minutes on a 1Ghz Powerbook G4 might seem slow compared to “classical” model checking using tools such as SMV or SPIN. Note, however, that ProB has not yet been tuned for speed. More importantly, the input language (B) is here of a much higher level, that every state contains information about 15 variables and that computing successor states for B is a quite expensive operation in itself (especially when `ANY` statements are involved, as they were for this example). Indeed, recent experiences show that a model checker in Prolog need not be much slower than tools such as SPIN, while being able to handle problems of similar size and allowing one to more easily check high-level languages [27].

E-TravelAgency Within our ABCD[9] project we have 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. 4, has 11 variables with non-trivial types. Attempting to check consistency of the *ETravelAgency* using Atelier-B resulted in a total of 206 POs. 156 of these were proved automatically by the Aterlier-B autoprover (75%), leaving 50 POs to be proved interactively.

ProB was very useful in the development of the specification, and was able to animate all of our models (see Fig. 4). The ProB model checker also discovered several invariant violations, e.g., related to incorrect responses or illegal multiple bookings. It was also able to discover a deadlock in one of the models, which was due to the fact that "session identifiers" were not properly recycled, meaning that after a while no new customers could log into the system. Such an error would have been more difficult to uncover within Atelier-B or the B-Toolkit.

9 Discussions

Scaling We have already applied ProB on reasonably sized, industrial examples. Still, a big question is: how well will ProB scale for even larger specifications. First, concerning the *animation*, we do not believe that the size of the B machine and the number of the B machine's variables are an important factor. We conjecture that ProB should be able to handle B machines with hundreds of operations and thousands of variables. Of course, there will be a user interface issue on how to display a large number of variables and options (e.g., a hierarchical view of enabled operations and of the state space could be added to Fig. 4), but there should not be an intrinsic computational problem. Indeed, to prove this point we have constructed several artificial specifications (the largest one having 240 operations, 80 variables of type partial function, 80 conditions in the invariant) and have been able to successfully animate them.

The limiting factor of animating B, will be more the complexity of the `ANY` statements and the complexity of the datastructures that are passed as operation arguments. The latter will probably be less of a problem, as arguments typically have to be "B0 typed" in order for machines to be implementable (as arguments cannot be refined). However, a single `ANY` statement with a very large domain (e.g., Fermat's last theorem on a large domain) could break the animator. The

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

same would be an `ANY` statement involving say a search over a function satisfying a certain property: `ANY fun WHERE fun = ~fun & fun: 0..100 +-> 0.100 THEN`.

It is well known that due to the state space explosion problem, *model checking* does not scale easily to large systems. Manual abstractions are still the key in many successful applications of model checking to larger examples. The same will be true here, at least if one wants to exhaustively explore the state space. ProB can still be used non-exhaustively to explore the state space and find potential problems. So, the model checking operations can still be useful for very large machines, not as a verification tool but as a sophisticated debugging tool.

Related Work We are not the first to realise the potential of logic programming for animation and/or verification of specifications. See for example [7], where an animator for VERILOG is developed in Prolog, or [5] where Petri nets are mapped to CLP. Also, the model checking system XMC contains an interpreter for value-passing CCS [29, 13]. A logic programming approach to encode denotational semantics specifications was applied in [21] to verify an Ada implementation of the "Bay Area Rapid Transit" controller.

Probably the most strongly related work is [6, 2], which uses a special purpose constraint solver over sets (CLPS) to animate B and Z specifications using the so-called BZ-Testing-Tools. Unfortunately we were not able to get hold of either CLPS or of the B-tool built on top of it, hence we cannot perform a detailed comparison of the animation facilities of ProB and the BZ-Testing-Tools. Indeed, our own B-Kernel, can be viewed as constraint solver over finite sets and sequences (it seems that sequences are not yet supported by [2]). At a higher level, [6, 2] put a lot of stress on animation and test-case generation, but do not seem to cater for model checking nor constraint-based checking. Indeed, to our knowledge we developed the first temporal model checker and the first constraint-based checker for consistency checking in B. Bellegarde et al [15] describes 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. Finally, while [6, 2] can handle Z as well as B specifications, we have interpreters for process languages such as CSP [23, 24] and StAC [16]. These can now be easily coupled with ProB to achieve an integration like [8], 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($\mathcal{SET}$) [14].[10] 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.

Another related work is [35], which presents an animator for Z implemented in Mercury. Mercury lacks the (dynamic) co-routining facilities of SICStus Prolog,

[10] 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, c.f., [14].

and [35] 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, and we believe that such an approach will not be able to cope with constraint-based checking. Another, recent animator for Z is ZANS [20]. It has been developed in C++ and unlike ProB only supports deterministic operations (called explicit in [20]).

Our constraint-based checker is strongly related to the ALLOY analyzer developed by Jackson [18, 19]. ALLOY is a special purpose lightweight object language which does not have the same penetration as B, but is well suited to constraint checking. The tool uses SAT solvers to find counter-examples in which an operation relates a consistent before state to an inconsistent after state. The ProB constraint-based checker has been heavily inspired by ALLOY and it would be interesting to compare the performance of ALLOY's SAT solving approach with ProB's constraint solving technique.

Future Work A lot of avenues can be pinpointed for further work. There are still a few features of B left that we need to support, so as to cover the whole language. For example, currently ProB does not yet support multiple machines or abstract constants of complex type (such as functions). An example of such an abstract function may be found in [9] where a constant *net* is used to model the connectivity between track sections in a railway network: $net \in SECTION \leftrightarrow SECTION$. The specification includes properties restricting the number of sections that can be directly connected. The relation is not given explicitly, so there will be many models for *net*, depending on the size of the given type *SECTION*.

Another obvious step is, in addition to supporting invariant and abort condition model checking, to allow *refinement* checking. Both the temporal and constraint-based checker can in principle be extended to check whether a refinement machine is a proper refinement of a specification machine, much like FDR checks refinement between CSP processes [30].

Also, it is possible to apply the constraint-based checker on B's proof obligations. If one could extract from say Atelier-B, the unproved proof obligations of a B machine, then one could apply ProB to try to find counter-examples for those proof obligations. This would be of great help in assisting the user, and could prevent him from spending a lot of time trying to prove an unprovable proof obligation.

We are also currently working to extract test cases for boundary conditions from within ProB. ProB is already capable of driving a Java implementation in synchrony with the animator. The hope is to develop a system that can generate test cases and verify them directly on an implementation. Another plan for further work is to link ProB with our U2B converter [32] which is a tool that converts UML models to B specifications.

We would also like to extend ProB so that it can check more complicated temporal properties. Indeed, consistency checking basically amounts to checking the temporal logic formula $AlwaysGlobally(\neg invariant\_violated)$, but it may be interesting to check more involved properties, e.g., that whenever one executes an operation *Request* eventually the *Acknowledge* operation will become enabled. Ideally one could also try to port the ProB system to XSB-Prolog, so as to obtain efficient model checking via tabling in the style of [29, 26]. Unfortunately, XSB Prolog does neither support finite domain constraints nor sophisticated co-routining; hence this will be a major undertaking. However, for those cases where ProB can construct the full state space of a B machine it is already possible to use our model checker [26] to verify CTL [12] formulas.

Finally, now that ProB has acquired sufficient functionality to be practically useful, we can focus some of our efforts on improving the performance of ProB. To that end we plan to compile B machines before animation or model checking, using our partial evaluation system LOGEN [25]. We hope that this will yield a substantial performance improvement.

Conclusion We have presented the ProB animation and model checking tool for the B method. We believe that this tool will be of high value to people developing B specifications, and our first case studies confirm this. 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 correct specification.

In one case, the Volvo vehicle function machine, ProB was actually able to prove the absence of errors (no counter-example exists and model checking was performed on the original unsimplified machine) fully automatically. (Note that it was a non-trivial task to prove this machine correct using Atelier-B.) So, one could argue that we have made it possible to use B without proof. In general, however, it will still be necessary to manually prove the B machine using Atelier-B or the B-Toolkit. Nonetheless, after the model checking a lot of errors should have already been found and corrected, and hopefully proof should be successful.

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. We also believe that ProB could be a valuable tool to teach beginners the B method, allowing them to play with and debug their first specifications.

We plan to release the tool later this year, and make it available at the following URL: `http://www.ecs.soton.ac.uk/~mal/systems/prob.html`.

Acknowledgements

We would like to thank Laksono Adhianto, Stefan Gruner, Leonid Mikhailov, and especially Carla Ferreira for their help in implementing and testing ProB. We are very grateful to Andy Gravell, Daniel Jackson, Cliff Jones, Steve Schneider, and to anonymous referees of FM'03 for their valuable feedback.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. 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.
3. AT&T Labs-Research. Graphviz - open source graph drawing software. Obtainable at `http://www.research.att.com/sw/tools/graphviz/`.
4. B-Core (UK) Limited, Oxon, UK. *B-Toolkit, On-line manual.*, 1999. Available at http://www.b-core.com/ONLINEDOC/Contents.html.
5. B. Bérard and L. Fribourg. Reachability analysis of (timed) petri nets using real arithmetic. In *Proceedings of Concur'99*, LNCS 1664, pages 178–193. Springer-Verlag, 1999.
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. Bowen. Animating the semantics of VERILOG using Prolog. Technical Report UNU/IIST Technical Report no. 176, United Nations University, Macau, 1999.
8. M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–198, 2000.
9. M. Butler. A system-based approach to the formal development of embedded controllers for a railway. *Design Automation for Embedded Systems*, 6(4):355–366, 2002,.
10. D. Cabeza and M. Hermenegildo. *The PiLLoW Web Programming Library.* The CLIP Group, School of Computer Science, Technical University of Madrid, 2001. Available at http://www.clip.dia.fi.upm.es/.
11. M. Carlsson and G. Ottosson. An open-ended finite domain constraint solver. In H. G. Glaser, P. H. Hartel, and H. Kuchen, editors, *Proc. Programming Languages: Implementations, Logics, and Programs*, LNCS 1292, pages 191–206. Springer-Verlag, 1997.
12. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
13. B. Cui, Y. Dong, X. Du, N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of ALP/PLILP'98*, LNCS 1490, pages 1–20. Springer-Verlag, 1998.
14. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):861–931, 2000.
15. F.Bellegarde, J. Julliand, and H. Mountassir. Model-Based Verification through Refinement of Finite B Event Systems. In *Formal Methods B Users Group Meeting (FM'99 B UGM Meeting)*, September 1999.
16. C. Ferreira and M. Butler. A process compensation language. In T. Santen and B. Stoddart, editors, *Proceedings Integrated Formal Methods (IFM 2000)*, LNCS 1945, pages 424–435. Springer-Verlag, November 2000.
17. P. Henderson. Modelling architectures for dynamic systems. In A. McIver and C. Morgan, editors, *Programming Methodology.* Springer-Verlag, 2003.
18. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, 2002.

19. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, pages 256–290, September 2001.
20. X. Jia. An approach to animating Z specifications. Available at `http://venus.cs.depaul.edu/fm/zans.html`.
21. L. King, G. Gupta, and E. Pontelli. Verification of a controller for BART. In V. L. Winter and S. Bhattacharya, editors, *High Integrity Software*, pages 265–299. Kluwer Academic Publishers, 2001.
22. J.-L. Lanet. The use of B for Smart Card. In *Forum on Design Languages (FDL02)*, September 2002.
23. M. Leuschel. Design and implementation of the high-level specification language CSP(LP) in Prolog. In I. V. Ramakrishnan, editor, *Proceedings of PADL'01*, LNCS 1990, pages 14–28. Springer-Verlag, March 2001.
24. M. Leuschel, L. Adhianto, M. Butler, C. Ferreira, and L. Mikhailov. Animation and model checking of CSP and B using Prolog technology. In *Proceedings of VCL'2001*, pages 97–109, Florence, Italy, September 2001.
25. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 2004. To appear.
26. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, 2000.
27. M. Leuschel and T. Massart. Logic programming and partial deduction for the verification of reactive systems: An experimental evaluation. In G. Norman, M. Kwiatkowska, and D. Guelev, editors, *Proceedings of AVoCS 2002, Second Workshop on Automated Verification of Critical Systems*, pages 143–149, 2002. Available as Technical Report CSR-02-6, University of Birmingham.
28. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.
29. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Proceedings of the International Conference on Computer-Aided Verification (CAV'97)*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
30. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.
31. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.
32. C. Snook and M. Butler. Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. In *UML 2000 WORKSHOP Dynamic Behaviour in UML Models: Semantic Questions*, October 2000.
33. Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 1996. Available at http://www.atelierb.societe.com/index_uk.html.
34. Tatibouet, Bruno. *The JBTools Package*, 2001. Available at http://lifc.univ-fcomte.fr/PEOPLE/tatibouet/JBTOOLS/BParser_en.html.
35. 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.