Tools for system validation with B abstract machines*

Michael Butler[1] and Michael Leuschel[1,2] and Colin Snook[1]

[1] School of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{mjb,mal,cfs}@ecs.soton.ac.uk
[2] Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
leuschel@cs.uni-duesseldorf.de

Abstract. In this paper we give an overview of some tools that we have developed to support the application of 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 contains a temporal and a state-based model checker, both of which can be used to detect various errors in B specifications. We also overview a recent extension of ProB that supports checking of specifications written in a combination of CSP and B. Finally we describe the UML-B profile and associated U2B tool that allows UML and B to be combined and is intended to make modelling with B more appealing to software engineers.

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 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

* This research was carried out as part of EU research projects: IST-1999-11435 MATISSE (Methodologies and Technologies for Industrial Strength Systems Engineering), IST-2000-30103 PUSSEE (Paradigm Unifying System Specification Environments for proven Electronic design) IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems), and the UK EPSRC funded ABCD (Automated Validation of Business Critical Systems with Component Based Designs).

one machine is a valid refinement of another. These activities are supported by industrial strength tools, such as Atelier-B [16] and the B-toolkit [3].

In this paper we give an overview of some tools that we have developed to support the application of 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 contains a temporal and a state-based model checker, both of which can be used to detect various errors in B specifications. We also overview a recent extension of ProB that supports checking of specifications written in a combination of CSP and B. CSP is a process algebra defined by Hoare [7]. CSP provides operators such as sequential composition, choice and parallel composition of processes, as well as synchronous communication between parallel processes. Finally we describe the UML-B profile and associated U2B tool that allows UML and B to be combined and is intended to make modelling with B more appealing to software engineers.

Section 2 gives an overview of the use of ProB for model checking of B specifications. Section 3 describes the use of ProB for model checking specifications written in a combination of CSP and B. Section 4 describes the UML-B profile and the U2B tool.

2 Overview of ProB

The ProB animator and model checker has been presented in [10]. Based on Prolog, the ProB tool supports automated consistency checking of B machines via *model checking* [5]. 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 to explore the state space non-exhaustively 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 can also be used as an animator of a B specification. So, the model checking facilities are still 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 typical development involves going through several levels of refinement to code generation *before* attempting any interactive proof [8]. 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, and making it 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.

2.1 Using ProB

```
MACHINE phonebook
SETS    Name ; Code
VARIABLES db
DEFINITIONS   scope_Name == 1..3;   scope_Code == 4..6
INVARIANT  db : Name +-> Code
INITIALISATION    db := {}

OPERATIONS
    cc <-- lookup(nn) =  PRE nn : Name THEN cc:=db(nn) END;
    add(nn,cc) =  PRE nn:Name & cc:Code THEN db := db \/ { nn |-> cc} END
END
```

Fig. 1. Phonebook example in B

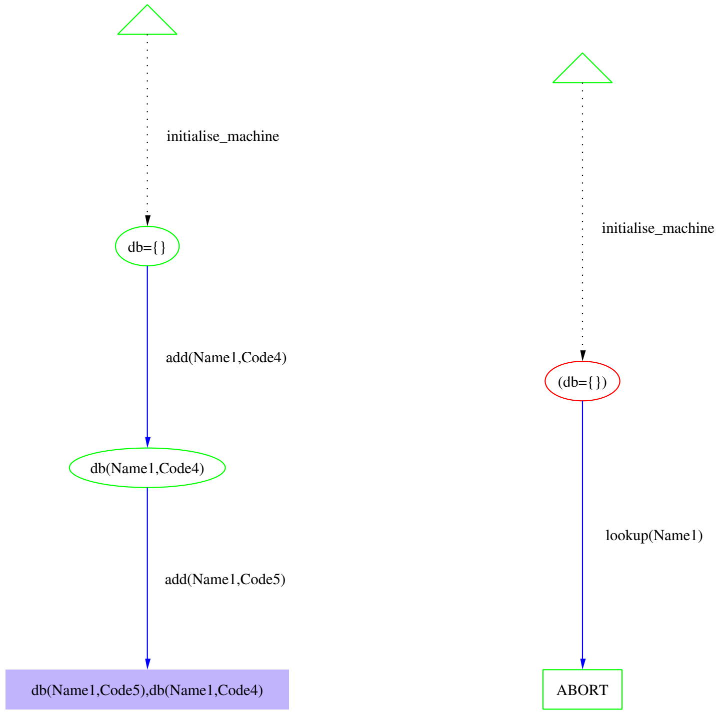
Figure 1 presents a simple B specification of a telephone directory mapping names to phone codes using a partial function *db*. The name associated with a code is retrieved using the *lookup* operation. Here, *nn* is an input parameter representing the name being queried and *cc* is a return value of the operation representing the code associated with *nn*. In the *add* operation, *nn* and *cc* are input parameters and the effect of the operation is to add a mapping from *nn* to *cc* to the directory *db* using set union.

The definitions in the *DEFINITIONS* clause of the phone book are used to limit the size of the given sets *Name* and *Code*. Normally B definitions are used to provide macros that can be included in several places in a machine. Since the definitions *scope_Name* and *scope_Code* are not used elsewhere in the machine, they do not affect the meaning of the specification as far as Atelier-B or the B-Toolkit are concerned. However, they are accessed by the ProB tool, acting as pragmas. In this case ProB will automatically enumerate the given set *Name* with the symbolic values $\{Name1, Name2, Name3\}$ and *Code* with the symbolic values $\{Code4, Code5, Code6\}$. This has the effect of making the state space finite for the purposes of model checking.

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 is violated. Trying to find such a sequence of operations is the task of the ProB *temporal model checker*.
2. it can find 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 *state-based model checker*.

Figure 2 presents two counterexamples resulting from performing a temporal check on the phone book specification using ProB. Figure 2(a) is a counterexample that results in two different mappings from $Name1$ being present in *db*.



(a) Invariant Violation (b) Abort conditon

Fig. 2. Temporal counter examples for the Phonebook Machine

This violates the invariant which specifies that db is a partial function. In the diagram, the transitions are operation occurrences, the nodes are states and the final rectangular state is the state that violates the invariant. This error can be fixed by strengthening the precondition of the *add* operation to say $nn \notin dom(db)$ or by using the function over-ride operator instead of set union.

Figure 2(b) is a counterexample which arises from trying to do a lookup when db is empty. This causes an attempt to evaluate an undefined expression which leads to a special state called $ABORT$. This error can be fixed by strengthening the precondition of the *lookup* operation to say $nn \in dom(db)$. Although not shown in the diagram, ProB will show which expression evaluation causes the abort through the interactive interface, i.e., it will show that there was an attempt to evaluate $\{\}(nn)$.

3 Combining B and CSP in ProB

In the Event B approach [2], 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 at modelling sequential activity. Typically one has to introduce an abstract 'program counter' to order the execution of actions. This can be a lot less transparent than the way in which one orders action execution in process algebras such as CSP [7]. CSP provides operators such as sequential composition, choice and parallel composition of processes, as well as synchronous communication between parallel processes.

Our 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 [9] 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 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. We envisage two main uses of the combined tool. Firstly it can be used to animate and model check specifications which are a combination of B and CSP. We illustrate this below. The second use of the tool 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 [4].

3.1 Specifying using B and CSP

In this section we illustrate the use of a combination of B and CSP to specify a system. The example we use to illustrate this concerns a service for distributing tokens to customers via offices and is based on [6]. The B part of our specification models a database mapping customers to the number of available tokens (Figure 3). It provides operations for creating and deleting customers which add or remove mappings for a customer to or from the database. There are operations for allocating a token to a customer as well as operations for requesting tokens and collecting tokens. Requesting tokens has no effect on the database. If there is more than one token available for a customer, the number of tokens returned is nondeterministically chosen to be less than or equal to the number of tokens available for that customer.

```
MACHINE Tokens
SETS
  OFFICE = {o1, o2};
  CUST = {c1, c2, c3}
CONSTANTS mx
PROPERTIES mx ∈ ℕ ∧ mx = 3
VARIABLES tokens
INVARIANT tokens ∈ CUST ⇸ (0..mx)

INITIALISATION tokens := {}

OPERATIONS

AddCust(cc) ≙
  PRE cc ∈ CUST ∧ cc ∉ dom(tokens)
  THEN tokens := tokens ∪ {cc ↦ 0}
  END;

RemCust(cc) =
  PRE cc ∈ CUST
  THEN tokens := {cc} ⩤ tokens
  END;

AllocToken(cc) =
  PRE cc ∈ CUST ∧ cc ∈ dom(tokens)
  SELECT tokens(cc) < mx THEN
    tokens(cc) := tokens(cc) + 1 END
  END;

ReqToken(cc, pp) =
  PRE cc ∈ CUST ∧ pp ∈ OFFICE
  THEN skip
  END;

toks ⟵ CollectToken(cc, pp) =
  PRE cc ∈ CUST ∧ pp ∈ OFFICE ∧
      cc ∈ dom(tokens)
  THEN
    IF tokens(cc) = 0
    THEN toks := 0
    ELSE
      ANY nn WHERE nn : ℕ∧
        1 ≤ nn ∧ nn ≤ tokens(cc)
      THEN toks := nn ||
        tokens(cc) := tokens(cc) − nn
      END END END
```

Fig. 3. Tokens B machine

The finiteness of the sets $OFFICE$ and $CUST$ in Figure 3 is required for exhaustive model checking. Finiteness is also impose by restricting the maximum number of tokens allocated to a customer using the constant mx. The $AllocToken$ operation is guarded to ensure that this allocation is never exceeded.

We wish to impose a certain coordination protocol on the operations of the system in Figure 3. Operations such as $CollectToken$ and $AllocToken$ should only be available after a customer has been added to the system. Furthermore, before a customer can collect tokens, they must first request those tokens at an office. This coordination is described by the CSP process $MAIN$ of Figure 3. This process consists of three parallel instances of the $Cust$ process, one for each customer. In a $Cust$ process, $AddCust$ is the only operation available initially. Once $AddCust$ has been performed, allocation and collection of tokens can proceed in parallel, modelled by the process $(Collection(C)[|RemCust|]Allocation(C))$. Collection and allocation synchronise on the $RemCust$ event because both are terminated by this event. Collection of tokens by a customer is intended to take place at offices to which customers have access. Before customers can collect tokens from an office, they must first request tokens at that office via a $ReqToken$ operation. Only then can they collect some (or all) of the tokens available for them. The definition of $Collection$ also ensures that a customer cannot be removed in between requesting some tokens and collecting those tokens.

$$
\begin{array}{rcl}
MAIN & = & Cust(c1) \;\;|||\;\; Cust(c2) \;\;|||\;\; Cust(c3) \\[1em]
Cust(C) & = & AddCust.C \rightarrow (Collection(C)[|RemCust|]Allocation(C)) \;;\; Cust(C) \\[1em]
Collection(C) & = & (\; ReqToken.C?O \rightarrow CollectToken.C.O \rightarrow Collection(C) \\
& & \;\;\Box\; RemCust.C \rightarrow SKIP) \\[1em]
Allocation(C) & = & (\; AllocToken.C \rightarrow Allocation(C) \\
& & \;\;\Box\; RemCust.C \rightarrow SKIP)
\end{array}
$$

Fig. 4. Tokens CSP equations

The overall behaviour of the service is determined by the parallel composition of the B and CSP parts. In this case, the CSP specification ensures that the $AddCust$ operation must be invoked before any of the other operations are allowed, and that tokens must be requested before they can be collected. The ProB tool allows the combined specification to be animated so that the overall behaviour can be explored interactively.

Now consider the preconditions of the operations of Figure 3. The $AddCust$ operation has $cc \notin dom(tokens)$ as a precondition, while the $AllocToken$ and $CollectToken$ operations have $cc \in dom(tokens)$ as a precondition. The preconditions represent assumptions about the conditions under which these operations will be invoked but are not enforced by the B machine on its own. Normally, when checking the consistency of a B machine using ProB, operation preconditions are used to restrict the reachable states by treating them in exactly the same way as operation guards. This form of checking detects no errors in the machine of Figure 3. An alternative form of checking can be applied in ProB which treats a violation of a precondition as an error. That is, an error is raised if a machine can reach a state which violates an operation precondition. With this second form of model checking, when the machine of Figure 3 is checked, an error is detected straightaway because the initial state violates the preconditions of $AllocToken$ and $CollectToken$. However, when this form of checking is applied to the combined B and CSP specification, no violation of preconditions is detected by ProB. This is because the CSP enforces an order on the invocation of the operations which guarantees that the preconditions are always satisfied.

4 Combining B and UML

The UML-B [14] is a profile of UML that defines a formal modelling notation. It has a mapping to, and is therefore suitable for translation into, the B language. UML-B consists of class diagrams with attached statecharts, and an integrated constraint and action language, called μB, based on the B AMN notation. UML-B provides a diagrammatic, formal modelling notation based on UML. The popularity of the UML enables UML-B to overcome some of the barriers to the acceptance of formal methods in industry. Its familiar diagrammatic notations make specifications accessible to domain experts who may not be familiar with formal notations. UML-B consists of:

- A subset of the UML - including packages, class diagrams and state charts
- Specialisations of these features via stereotypes and tagged values,
- Structuring mechanisms (systems, components and modules) based on specialisations of UML packages
- UML-B clauses - a set of textual tagged values to define extra modelling features for UML entities,
- μB - an integrated action and constraint language based on B,
- Well-formedness rules

The UML-B profile uses stereotypes to specialise the meaning of UML entities, thus enriching the standard UML notation and increasing its correspondence with B concepts. The UML-B profile defines tagged values (UML-B clauses) that may be used to attach details, such as invariants and guards, that are not part of the standard UML. Many of these clauses correspond directly with those of B providing a 'fallback' mechanism for modelling directly in B when UML entities are not suitable. Other clauses, having no direct B equivalent, are provided for adding specific UML-B details to the modelling entities. UML-B provides a combined diagrammatic and textual, formal modelling notation. It has a well defined, formal semantics as a direct result of its mapping to B. UML-B hides

B's infrastructure and packages mathematical constraints and action specifications into small sections each being presented in the context of its owning UML entity.

The U2B [13] translator converts UML-B models into B components (abstract machines and their refinements). Translation from UML-B into B enables the existing B tools to be utilised. In many respects B components resemble an encapsulation and modularisation mechanism suitable for representing classes. A component encapsulates variables that may only be modified by the operations of the component. However, to ensure compositionality of proof, B imposes restrictions on the way variables can be modified by other components (even via local operations). Translating classes into B components imposes corresponding restrictions on the relationships between classes. Therefore we translate a complete UML package (i.e. many classes and their relationships) into a single B machine or refinement. This option allows unconstrained (non-hierarchical) class relationship structures to be modelled. Since the B language is not object-oriented, class instances must be modelled explicitly. Attributes and associations are translated into variables whose type is a function from the class instances to the attribute type or associated class. For example a class A with attribute x of type X would generate the following B:

$$
\begin{array}{l}
\text{SETS } \ A \\
\text{VARIABLES } \ x \\
\text{INVARIANT } \ x \in A \rightarrow \mathbb{P}(X)
\end{array}
$$

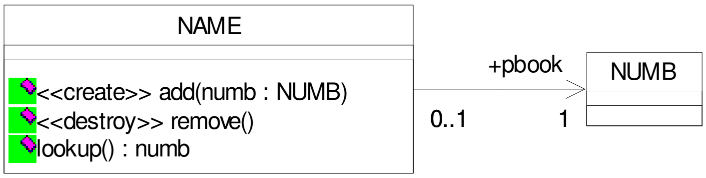
Any reference to x in a μB constraint or action will usually refer to the attribute of some implicit instance. In the translation to B, the implicit instance is made explicit ($thisA$) and a reference to x is replaced by function application $x(thisA)$. The full function representing an attribute or association x may be accessed directly by referring to $\$x$. In the above case, the class A is assumed to have a fixed number of instances (multiplicity n in UML terms). If a class can have a varying number of instances (multiplicity $0..n$ in UML terms), then the translation would be:

$$
\begin{array}{l}
\text{SETS } \ A\_SET \\
\text{VARIABLES } \ A, x \\
\text{INVARIANT } \ A \subseteq A\_SET \ \wedge \ x \in A \rightarrow \mathbb{P}(X)
\end{array}
$$

Here the varying set of instances is modelled by the variable A which is a subset of the larger type A_SET.

Operation behaviour may be represented textually in μB, as a state chart attached to the class, or as a simultaneous combination of both. Further details of UML-B are given in [13]. Examples of previous case studies using UML-B and U2B are given in [11–13, 15].

To give a flavour of UML-B, consider the specification of a telephone book in Figure 5. The classes, $NAME$ and $NUMB$ represent people and telephone numbers respectively. The association role, *pbook*, represents the link from each name to its corresponding telephone number. Multiplicities on this association ensure that each name has exactly one number and each number is associated with, at most, one name (i.e., the relationship is injective). The figure also shows μB conditions and actions for the operations. The *add* operation of class $NAME$ has the stereotype *create* which means that it adds a new name to the class. It takes a parameter *numb*, which must be an instance of the class, $NUMB$, but not already used in a link of the association *pbook* (see μB operation guard), and uses this as the *pbook* link for the new instance (see μB operation action). The figure also shows the multiplicities of the $NAME$ and $NUMB$ classes. The set of instances of $NUMB$ is assumed to be fixed (multiplicity n) while the set of instances of $NAME$ is variable (multiplicity $0..n$). The resulting B generated by U2B is shown in Figure 6. This generated B specification is suitable for animation and model checking by the ProB tool.



operation	guard	action
add	$numb \notin ran(\$pbook)$	$pbook := numb$
remove	-	-
lookup	-	$numb := pbook$

class	multiplicity
NAME	0..n
NUMB	n

Fig. 5. UML-B model of a telephone book

5 Concluding

We have presented the ProB animation and model checking tool for the B method. Our view 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. 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'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.

```
MACHINE Phone
SETS NAME_SET; NUMB
DEFINITIONS
invariant == ( NAME:POW(NAME_SET) &
               pbook : NAME >-> NUMB )

VARIABLES NAME, pbook
INVARIANT invariant
INITIALISATION
    NAME, pbook :(invariant &
        NAME={} & pbook = {} )

OPERATIONS
Return  <-- add (numb) =
PRE numb:NUMB THEN
  ANY thisNAME WHERE
    thisNAME : NAME_SET & thisNAME/:NAME & numb:NUMB & numb/:ran(pbook)
  THEN
    NAME := NAME\/{thisNAME} ||
    Return := thisNAME ||
    pbook(thisNAME):=numb
  END
END ;

remove (thisNAME) =
 PRE thisNAME:NAME
 THEN
    NAME := NAME-{thisNAME} ||
    pbook := {thisNAME} <<| pbook
 END;

numb <--  lookup (thisNAME) =
 PRE thisNAME:NAME
 THEN numb:=pbook(thisNAME)
 END
END
```

Fig. 6. B generated from UML-B model by U2B

The combined model checker for CSP and B as an enhancement of the existing ProB checker allowing for automated consistency checking of specifications written in a combination of CSP and B. We have shown how ProB can now be used to automatically check consistency between B and CSP specifications (i.e., checking that no B preconditions are ever violated). Though not described in this paper, ProB also supports refinement checking between B models and between combinations of CSP and B. Further work is required to enhance the scalability of the model checking approach, especially for refinement checking (although some quite large, realistic specifications have already been successfully verified).

Regarding UML-B and U2B, the emergence of the UML as a de-facto standard for object-oriented modelling has been mirrored by the success of the B method as a practically useful formal modelling technique. The two notations have much to offer each other. The UML provides an accessible visualisation of models facilitating communication of ideas but lacks formal precise semantics. B, on the other hand, has the precision to support animation and rigorous verification but many software engineers find the notation difficult to learn, visualise and communicate. Our experience has been that there is an encouraging level of interest in UML-B from industry. The interest is mainly from companies that are investigating formal methods but not using them. Their reaction is that they would probably not use B in its current (textual) form but they may consider using UML-B as it becomes more mature and usable. They view the UML basis of UML-B as providing a more understandable and visible route into using formal specifications. Several of our industrial contacts are keenly participating in ongoing research into the development of UML-B and U2B. Furthermore the combination of ProB and U2B provides a route to animation and model checking of UML-B models, further enhancing the attraction of UML-B.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *Second International B Conference*, April 1998.
3. B-Core (UK) Limited. B-toolkit manuals. Oxon, UK, www.b-core.com, 1999.
4. M. Butler and M. Leuschel. Combining CSP and B for Specification and Property Verification. Technical report, School of Electronics and Computer Science, University of Southampton, 2005. eprints.ecs.soton.ac.uk/10388/.
5. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
6. P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche, and B. Walters. Questions and answers about ten formal methods. In *Proc. 4th Int. Workshop on Formal Methods for Industrial Critical Systems*, Trento, Italy, Jul 1999. http://www.dsse.ecs.soton.ac.uk/techreports/99-1.html.
7. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.
8. J-L Lanet. The use of B for Smart Card. In *Forum on Design Languages (FDL02)*, September 2002.
9. 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.

10. M. Leuschel and M. Butler. ProB: A Model Checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proceedings FME 2003, Pisa, Italy*, LNCS 2805, pages 855–874. Springer, 2003.
11. C. Snook and M. Butler. Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. In *Proceedings of UML 2000 Workshop, Dynamic Behaviour in UML Models: Semantic Questions*, 2000.
12. C. Snook and M. Butler. Using a graphical design tool for formal specification. In *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, 2001.
13. C. Snook and M. Butler. U2B - A tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*. Springer, 2004.
14. C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. Technical report, School of Electronics and Computer Science, University of Southampton, 2004. eprints.ecs.soton.ac.uk/10169/.
15. C. Snook and K. Sandstrom. Using UML-B and U2B for formal refinement of digital components. In *Proceedings of Forum on specification and design languages (FDL03)*, 2003.
16. Steria. Atelier B, User and Reference Manuals. Aix-en-Provence, France, http://www.atelierb.societe.com/index_uk.html, 1996.