Debugging Event-B Models using the ProB Disprover Plug-in [⋆]

Olivier Ligot[1], Jens Bendisposto[2] and Michael Leuschel[2]

[1] Facultés Universitaires Notre-Dame de la Paix Namur
Namur, Belgium
olivier.ligot@student.fundp.ac.be

[2] Softwaretechnik und Programmiersprachen
Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany
{bendisposto,leuschel}@cs.uni-duesseldorf.de

Abstract. The B-method, as well as its offspring Event-B, are both tool-supported formal methods used for the development of computer systems whose correctness is formally proven. However, the more complex the specification becomes, the more proof obligations need to be discharged. While many proof obligations can be discharged automatically by recent tools such as the RODIN platform, a considerable number still have to be proven interactively. This can be either because the required proof is too complicated or because the B model is erroneous. In this paper we describe a disprover plugin for RODIN that utilizes the ProB animator and model checker to automatically find counterexamples for a given problematic proof obligation. In case the disprover finds a counterexample, the user can directly investigate the source of the problem (as pinpointed by the counterexample) and she should not attempt to prove the proof obligation. We also discuss under which circumstances our plug-in can be used as a prover, i.e., when the absence of a counterexample actually is a proof of the proof obligation.

Keywords: RODIN, ProB, Event-B, B-Method, Autoprover.

1 Introduction

The B-method introduced by J.-R. Abrial [1] is a theory and methodology for formal development of computer systems which is based on the notion of *abstract machines* and *refinement*. B is used in industry, mainly for safety critical applications. It is supported by several industrial strength tools such as AtelierB [21], the B Toolkit [5] or B4Free for proving correctness and code generation and tools for animation, modelchecking and model based testing such as ProB [12] or the BZTT [4].

However, classical B is lacking certain dynamic constraints (temporal logic constraints, liveness constraints) that can be used to model how a system can

[⋆] This research is being carried out as part of the EU funded research projects: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

evolve. This shortcoming was one of the reasons to extend B to Event-B [2, 3] which enables the specification of reactive systems without abandoning the notion of refinement.

An Event-B specification is written as an abstract machine that consists of variables which define its state and some events. An event decomposes into a predicate, the guard, that specifies under which circumstances it might occur and some generalized substitutions called actions. For instance, if the state s of an abstract machine is $(x = 2, y = 7)$ and there is an event e with the guard $true$ and the action $x := y$, then a successor state of s might be $(x = 7, y = 7)$.

A notable recent development is the EU funded research project IST 511599 RODIN, which aims to develop an open tool platform based on Eclipse that supports Event-B. The objective of RODIN is to create a unified methodology and supporting tools for cost-effective, rigorous development of software systems.

A rigorous software development requires to reason about the correctness of the formal specification. For example, one should verify that an Event-B model does not violate its invariant. Other correctness conditions are related to refinement or the properties associated with constants. The proof obligations that need to be discharged in order to establish correctness can be mechanically extracted from an Event-B model. For instance, one proof obligation will stipulate that the initialisation of an Event-B model must establish the invariant. The RODIN platform comes with a tool, the proof obligation generator, that extracts proof obligations from a model (see Figure 1).

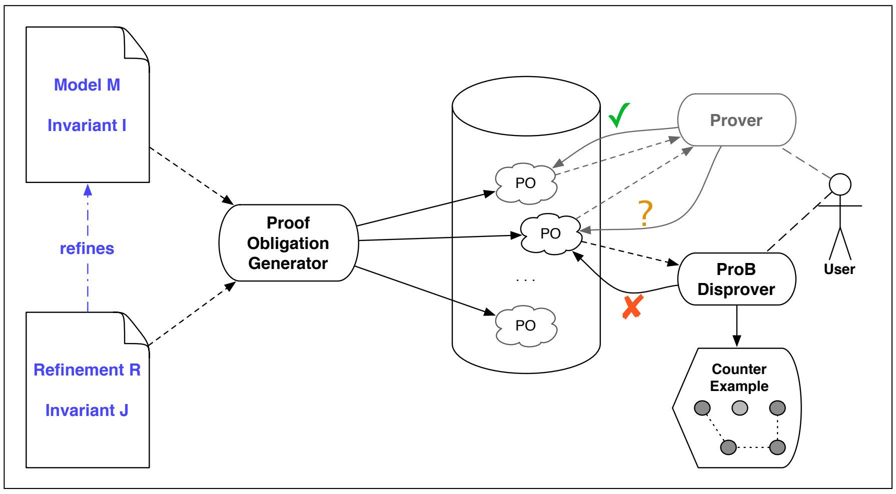

Fig. 1. Overview of proof activities and the role of the disprover

The RODIN platform also comes with some automatic provers, which can discharge a considerable number proof obligations automatically. Obviously, due to incompleteness, not all proofs can be done automatically. In that circumstance the user is left wondering:

– Is the proof obligation valid and the proof simply too complicated for the automated prover? In other words, should I start up the interactive prover and try to prove the goal manually?
– Or is there a problem within the specification and should I spend time looking for the error and then correct it?

Pursuing either path can lead to considerable wasted effort. In this paper we present a tool which helps the user in this common situation: a disprover which tries to find a counterexample for the particular problematic proof obligation (see also Figure 1).

– If the disprover finds a counterexample we know that it is futile to spend time with the interactive prover. Also, the counterexample will give us a handle on the problem and help us find the error in the specification more quickly.
– If the disprover finds no counterexample, we know that—in certain circumstances at least—the proof obligation seems to be valid. Of course, we are still not sure whether the proof obligation is true in all circumstances; but we have at least gained some additional confidence about its validity.

As an example, suppose we want to prove the theorem, that every finite undirected graph has at least two nodes of the same degree.[3] Our Event-B model will contain the basic set $NODES$ and a graph consists of a set $V \subseteq NODES$ of vertices as well as a symmetric binary relation E representing the edges. The degree of a vertex v is simply $card(\{v\} \lhd E) = card(E[\{v\}])$. A sequent representing our theorem might be something like the following:

$$\begin{gathered}V \subseteq NODES \wedge E \in NODES \leftrightarrow NODES \wedge E = E^{-1} \wedge card(V) \in \mathbb{N} \\ \Rightarrow \\ \exists x \exists y : x \in V \wedge y \in V \wedge x \neq y \wedge \\ card(\{x\} \lhd E) = card(\{y\} \lhd E)\end{gathered}$$

However, this theorem is not provable since we made two mistakes in the definition. While it might not be obvious which mistakes we made, the disprover plug-in finds counterexamples that will help us to identify the problems and to correct the theorem. A trivial counterexample the tool finds is the empty graph, another counterexample found by our tool with 5 vertices that contains self loops is shown in Fig. 2. As can be seen, all vertices have a different degree. So we need to strengthen the left side of our implication to disallow self loops and graphs with less than two nodes, after which the disprover can no longer find a counterexample:

[3] This example is inspired by a talk given by Leslie Lamport at B'2007.

$$\begin{gathered} V \subseteq NODES \wedge E \in NODES \leftrightarrow NODES \wedge E = E^{-1} \wedge card(V) \in \mathbb{N} \wedge \\ card(V) > 1 \wedge id(NODES) \cap E = \varnothing \\ \Rightarrow \\ \exists x \exists y : x \in V \wedge y \in V \wedge x \neq y \wedge \\ card(\{x\} \lhd E) = card(\{y\} \lhd E) \end{gathered}$$

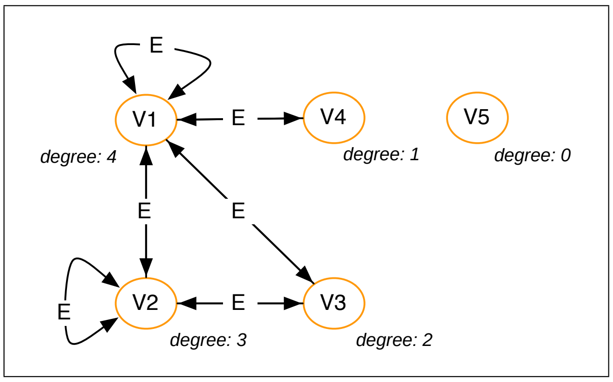

Fig. 2. Counterexample found by the disprover

RODIN can be extended by third parties, in particular it is possible to add external proving tools. We have thus developed a prover plug-in, which works as a disprover. Our plug-in is based on the Prolog based animator and model checker ProB [12]. The ProB animator is fully automatic and does not require the user to guess the right values for the operation arguments or choice variables. The undecidability of animating B is overcome by restricting animation to finite sets and integer ranges, while efficiency is achieved by delaying the enumeration of variables as long as possible. The main idea of our work is to translate an individual proof obligation into a B machine such that the animator can be used to find counterexamples. Of course, one could have used the ProB model checker itself on the whole Event-B model. This is an alternate validation option, but this will "only" find sequences of operations which violate the invariant starting from some valid initialisation; i.e., it will not detect problems if the invariant is too weak (see [12]). Furthermore, by restricting our attention to a single, problematic proof obligation we can increase the likelihood of the disprover finding counterexamples.

The rest of the paper is structured as follows. First we provide some background on proof in Event-B in Section 2. Then we present the underlying methodology of our disprover plug-in in Section 3, before discussing the actual implementation in Section 4. We conclude with remarks on how to use the disprover as a prover and other future work in Section 5.

2 The Event-B proving subsystem

This section gives an introduction to proofs in Event-B. We will also discuss the architecture of the RODIN proving subsystem, its Kernel prover and how external reasoners work.

Proving correctness A proof in Event-B is constructed using (a slight variation of the) sequent calculus [18]. A *sequent* in Event-B is of the form $\Gamma \vdash \Sigma$ where Γ is a finite set of predicates called *hypotheses* and Σ is a single predicate called *goal*. A sequent basically means that the goal should be a logical consequence of the hypotheses Γ. Proofs of sequents are carried out using an *inference rule*. An inference rule contains a finite set of sequents A — the *antecedent* — and a single sequent C — the *consequent*. An inference rule means: if we can prove all sequents within A, then C has also been proven. It is also possible that a rule has the empty set as antecedent, this means that C has been proven. A proof for a sequent can thus be viewed as a finite tree. Each node of this *proof tree* contains a sequent s as well as an inference rule r whose consequent is s. The children of a node are the sequents in the antecedent of its rule r. Leaf nodes are those nodes where the associated inference rule has the empty set as antecedent. To actually discharge a proof obligation *po*, we need to find a finite proof tree whose root node is labelled with *po*.

Note that the antecedent of sequents might contain a subset called *type environment*, where the predicates only carry type information such as *x is an integer* ($x \in \mathbb{Z}$) or *y is a member of a basic set* Y ($y \in Y$). Sequents of the type environment can be statically checked by a type checker and thus have the empty set as antecedent (unless there is a typing error). We call a sequence of inference rules that discharge a certain type of sequents a *proof tactic*. It can be seen as a kind of pattern for proving.

The RODIN proving subsystem In RODIN, a considerable number of proofs can be done automatically by the proving subsystem that consists, as shown in Fig. 3, of the proof obligation generator and the Event-B Kernel Prover [17]. The proof obligation generator extracts all proof obligations from the Event-B model that need to be discharged in order to prove correctness of the model and stores them in a XML database file. After all POs have been generated, the kernel prover tries to discharge valid POs automatically.

As shown in Fig. 3, the Event-B kernel decomposes into the proof manager and a set of prover plugins. While the proof manager is responsible for storage, traversal, composition and reusage of proofs, the prover plugins try to generate valid inferences in order to discharge the proof obligations. The proof manager also maintains the state of current proofs for all proof obligations and decides if they have to be discharged and calls external provers if they are non-interactive. There are also interactive reasoners that require the user to apply them, the ProB disprover plug-in is such an interactive plug-in. In the next section we show the underlying principles of our plug-in, before showing in Section 4 how it was integrated into the RODIN platform.

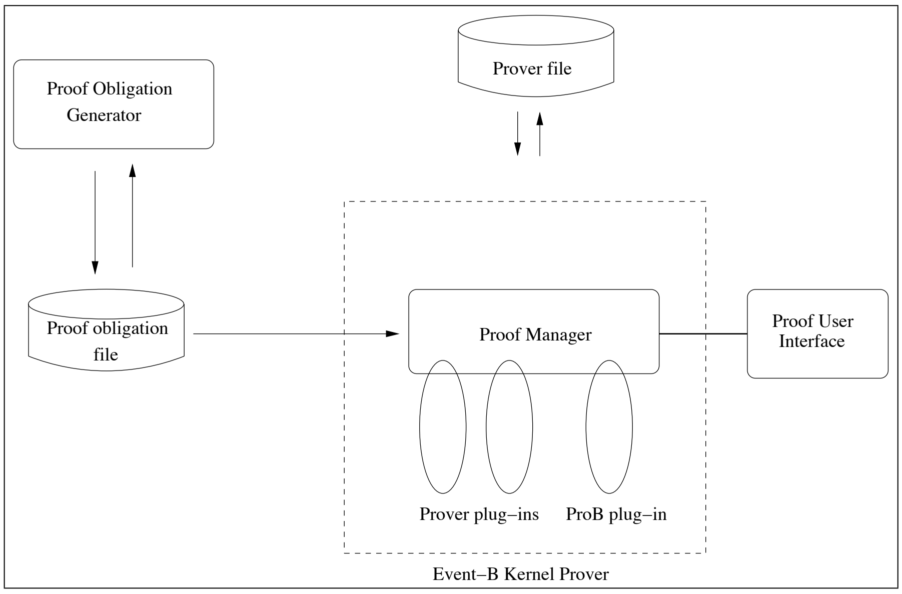

Fig. 3. Architecture of the RODIN proving subsystem

3 The principle of disproving using ProB

In the following, we will explain, how a sequent can be translated into a B machine that can be used with ProB.

Finding counterexamples Let $G(x_1, ..., x_k)$ be the goal of a sequent s and let $H_1(x_1, \dots, x_k), \dots, H_n(x_1, \dots, x_k)$ be the hypotheses. To find a counterexample for s, we need to check if the predicate

$$\exists x_1, ..., x_k : (H_1(x_1, ..., x_k) \wedge ... \wedge H_n(x_1, ..., x_k)) \Longrightarrow \neg G(x_1, ..., x_k) \qquad (1)$$

holds. If it does, then we can extract a concrete counterexample by finding a valuation for $x_1, ..., x_k$ which makes the implication true. Finding values that satisfy a propositional boolean formula is NP complete, and for the first order logic formulas that can occur within sequents, the problem is undecidable. To overcome this difficulty, we have to restrict sets to relatively small, finite domains. As a consequence, we know that in principle it is not possible to guarantee that a disproving algorithm can automatically find a counterexample if one exists. In other words, the absence of a counterexample does not mean in general, that a proof obligation is valid. (There are, however, certain cases where the absence of a counterexample discharges a proof obligation. We discuss these cases in section 5.)

Transforming sequents into B machines ProB can be used to find a counterexample for a given sequent, but it needs a classical B machine that encodes the sequent as its input. Fortunately this encoding is — at least in principle — not difficult to obtain. We create a B machine, that contains an operation $disproveHypotheses$ with the predicate from equation (1) as its guard. The operation is enabled, if and only if ProB can find a counterexample.

In order to construct this stand-alone machine, we need to extract some information from the original Event-B specification such as axioms[4], carrier sets, parameters, variables (including type information) and constants. Furthermore, we need information about the sequent to be (dis-)proved, such as the hypotheses and the goal. The translation of these information is in most cases straightforward, for example we construct the `SETS` clause of the machine by enumerating the set definitions from the original Event-B specification. In some cases the translation is less obvious. For instance, we translate the axioms together with the type information of the constants into the `PROPERTIES` clause. We generate new definitions called `TypeEnvironment` and `Hypotheses` inside the `DEFINITIONS` clause. The `TypeEnvironment` is a subset of the hypotheses that only contains predicates dealing with type information. A schema of the B machine constructed from a given sequent $H_1, H_2, \dots H_n \vdash G$ is shown in Listing 1.1.

Selecting Hypotheses The RODIN proving subsystem allows the user to select a subset of hypotheses that are in the database, these hypotheses are either directly derived from the specification or previously proven. Obviously if a subset of H proves G, then H also proves G.

$$H' \subseteq H \wedge H' \vdash G \Rightarrow H \vdash G$$

Thus a user can restrict the hypotheses in a sequent to an arbitrary subset of so-called *selected hypotheses*, by removing hypotheses that are of not relevant for the proof. By default, a particular set of hypotheses which deals with the involved variables are automatically selected by RODIN. The user can also decide to hide a particular subset of hypotheses, this subset are called *hidden hypotheses*. In fact, there are thus two alternatives:

– run the external disprover with the *selected* hypotheses or
– run it with *all* hypotheses except the hidden ones.

In any case, the user can choose which alternative to apply (our plug-in provides two buttons) and change his mind later.

[4] An axiom is treated like a sequent $true \vdash A$

Listing 1.1. Schema of an abstract machine constructed from a sequent

```
MACHINE Disprove
DEFINITIONS
    TypeEnvironment == $H_1(x_1,...,x_k)$ & ... & $H_i(x_1,...,x_k)$;
    Hypotheses == TypeEnvironment &
        $H_{i+1}(x_1,...,x_k)$ & ... & $H_n(x_1,...,x_k)$;
    Goal == $G(x_1,...,x_k)$
OPERATIONS
    disprove($x_1,...,x_k$) =
        PRE Hypotheses & not(Goal)
        THEN skip
        END
END
```

4 Implementation of the ProB disprover plug-in

In previous work [6], we have developed a version of ProB that integrates with Eclipse. Its main component is the Eclipse ProB plug-in as shown in Fig. 4. It allows third party tools to use ProB for several tasks, thus it can be seen as a Java abstraction layer for the Prolog part of ProB. The disprover uses this core plug-in to find counterexamples. Therefore it creates — when applied to a node of the proof tree — a B machine as described in Section 3 and starts animating this machine. If the operation `disprove` is enabled, we have found a counterexample.

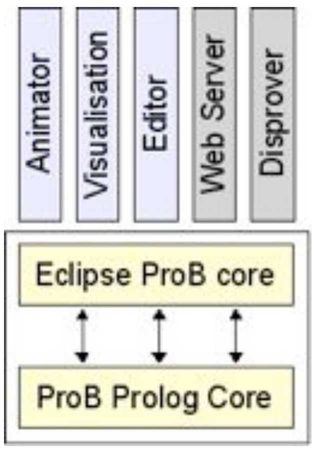

Fig. 4. Architecture of the ProB Eclipse Version

Our disprover plug-in consists of a user interface (UI) that displays the results of a proof and a core component that encapsulates the proof logic. The UI is an extension to the RODIN proving user interface. It allows the user to select a node in the proof tree, that he wants to check with ProB. The core plug-in provides a way to apply the ProB disprover. Its role is to:

- Translate the sequent into a B machine.
- Call ProB through the Eclipse ProB core plug-in.
- Return results to the user interface.
- Handle failures, time outs and user cancel requests.

Displaying counterexamples A first approach was to display counterexamples in a separate window. This solution was however not very useful because there is no connection between the counterexample and the proof obligations. We thus take another approach to resolve this problem by applying a case distinction [1] to the node in the proof tree.[5] As seen in the previous section, a counterexample can be described by a predicate

$$C_p \equiv x_1 = e_1, ..., x_k = e_k$$

Now we apply a case distinction to the node. This results in two child nodes with the sequents

1. $H, C_p \vdash G$
2. $H, \neg C_p \vdash G$

The first sequent is the case where the counterexample was found (C_p makes G false). The second one is the remaining case, where the counterexample is not considered. The user can then repeat the step of applying the disprover plugin to the second predicate to try to find a further counterexample.

Figure 5 shows the tool displaying a counterexample. To launch the external disprover, one has to push the green button.[6] The advantages of this approach are the flexibility of the exploration – by exploring the proof deeper if one wants to find other counterexamples – and the connection with the proof itself.

[5] Original idea by Farhad Mehta.

[6] The red button is for all hypotheses, the green for selected hypotheses.

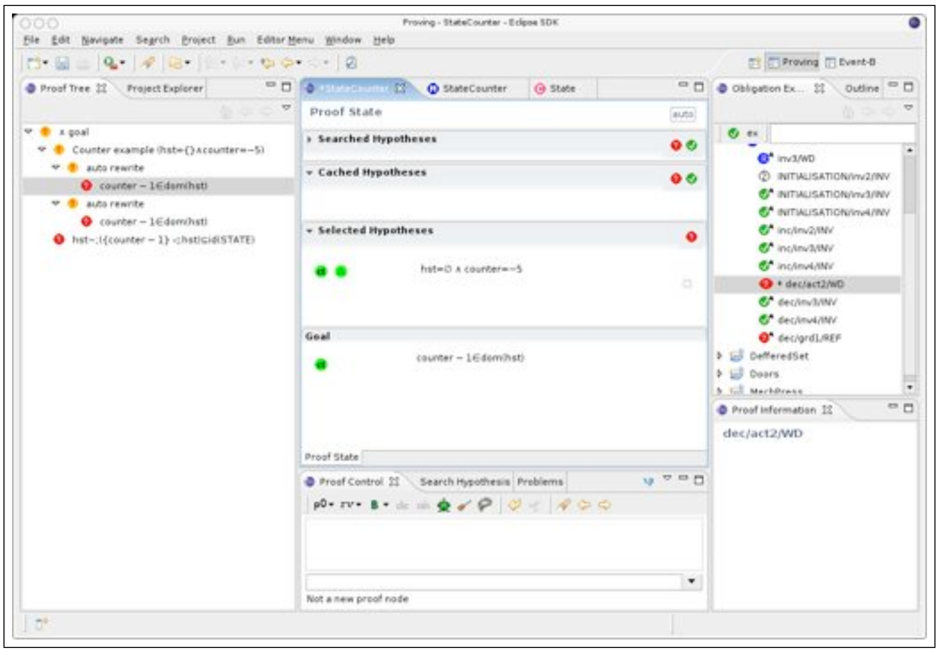

Fig. 5. Screenshot showing the display of a counterexample

5 Future Work and Conclusion

Using ProB as a prover If the ProB disprover fails to find a counterexample for a particular proof obligation, we cannot infer that the proof obligation is true. This is due to two reasons:

– **deferred sets:** If a B machine uses deferred sets (i.e., sets which are not explicitly enumerated in the SETS clause), then the cardinality of those sets is not a priori fixed; the set could even be infinite. ProB, however, will check the proof obligation only for some finite cardinalities of the deferred sets, and thus may fail to find existing counterexamples. For example, ProB will fail to find a counterexample for the formula $\exists n.(n : \mathbb{N} \wedge card(S) < n)$, where S is a deferred set without further restrictions.
– **integers:** If an integer variable occurs inside a proof obligation, whose value is *not* determined by the rest of the proof obligation, ProB will enumerate the variable only within a finite interval (between user determined MININT and MAXINT). Again, ProB may thus fail to find counterexamples for integer values which lie outside of MININT..MAXINT.

However, if a B machine contains neither deferred sets nor integer variables, the ProB disprover can actually also be used as a *prover*. This condition can be easily checked statically, in which case our disprover can inform the Rodin platform that the PO has actually been proven. Some practical B specifications fall into this category. For example, the Volvo vehicle function used in [12]. Another example is a Hamming encoder [8], for which Dominique Cansell has used ProB to prove some essential theorems (which would have been extremely tedious to prove by hand).[7]

In future work, we are planning to implement a static analysis which safely infers intervals for the integer variables. If all variables can be proven to lie within a finite range, ProB could be used as a prover on this larger class of specifications (provided MININT and MAXINT cover all those ranges).

More future work An empiric evaluation of the use of ProB as a disprover is required if one wants to see if the plug-in is efficient or can be more optimized. A number of tests have already be done but more benchmark tests on several operating systems would be welcome.

When using relations or functions in the Event-B, the possible values for the variables of a sequent grows extremely large. For example, given $r : A \leftrightarrow A$, where A has a cardinality of 4, we have $2^{4*4} = 65536$ possibilities for r. Given $x : \mathbb{P}(A \leftrightarrow A)$ we even have $2^{2^{4*4}} = 2^{65536}$ possibilities for x. ProB has to investigate these possibilities in order to search for a counterexample. Symmetry reduction is one way to ease this task, and we plan to check whether we can make use of ProB's recent developments [13, 14] in that area for our disprover plug-in. Another option is to partion the configuration space into several areas, and let different instances of ProB running in parallel take care of the corresponding exploration.

Related work A very popular tool for validating models and finding counterexamples is Alloy [11], which makes use SAT solvers (rather than constraint solving). However, the specification language of Alloy is first-order and thus cannot be applied "out of the box" to Event-B models.

Earlier related work are the model generators FINDER [19] and MACE [15] which can also be used to find counterexamples. The prover Isabelle now also has a quick check function [7], looking randomly for counterexamples. There are many more related works, such as the more recent [20], and even several CADE and IJCAR workshops on disproving have been organized. There is also considerable work on combining model checking [9] with theorem proving in general (e.g., [16, 10]).

Conclusion In summary, we have presented a method to use the existing model checker ProB as a tool for proof support, by trying to find counterexamples for individual proof obligations. We have also discussed under which circumstances the model checker can be used as a prover. We have presented the implementation within Eclipse, using the Rodin Event-B platform and have shown how

[7] Private communication by Dominique Cansell.

this has enabled to use the model checker in a very targeted and convenient way. We believe that a model checker can provide a very valuable support for the B developer, avoiding unnecessary time spent trying to prove a false proof obligation.

References

1. J.-R. Abrial. *The B book : assigning programs to meanings*. Cambridge University Press, 1996.
2. J.-R. Abrial. Extending B without changing it. (For Distributed System). Proc. of 1st Conf. on B Method. pages 169–191, 1996.
3. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 83–128, London, UK, 1998. Springer-Verlag.
4. 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.
5. U. B-Core (UK) Limited, Oxon. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.
6. J. Bendisposto. Integration of the ProB modelchecker into Eclipse. Bachelor's thesis, July 2006.
7. S. Berghofer and T. Nipkow. Random Testing in Isabelle/HOL. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.
8. D. Cansell, S. Hallerstede, and I. Oliver. UML-B specification and hardware implementation of a hamming coder/decoder. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, Nov 2004. Chapter 16.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. E. L. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Asp. Comput.*, 17(2):201–221, 2005.
11. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, 2002.
12. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
13. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry Reduction for B by Permutation Flooding. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
14. M. Leuschel and T. Massart. Efficient Approximate Verification of B via Symmetry Markers. *Proceedings International Symmetry Conference*, pages –, Januar 2007.
15. W. McCune. MACE 2.0 Reference Manual and Guide. *CoRR*, cs.LO/0106042, 2001.
16. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

17. RODIN Consortium. Rodin deliverable D10 - Specification of basic tools and platform. Technical report, 2005. Available at http://rodin.cs.ncl.ac.uk/deliverables/rodinD10.pdf.
18. RODIN Consortium. Rodin deliverable D7 - Event B language. Technical report, 2005. Available at http://rodin.cs.ncl.ac.uk/deliverables/rodinD7.pdf.
19. J. K. Slaney, E. L. Lusk, and W. McCune. SCOTT: Semantically Constrained Otter System Description. In A. Bundy, editor, *CADE*, volume 814 of *Lecture Notes in Computer Science*, pages 764–768. Springer, 1994.
20. G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. In *Foundations Of Computer Security Workshop*, 2002.
21. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com/index_uk.html`.