Towards Infinite-State Symbolic Model Checking for B and Event-B

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Sebastian Krings
aus Düsseldorf

Düsseldorf, August 2017

aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent:	Prof. Dr. Michael Leuschel Heinrich-Heine-Universität Düsseldorf
Korreferent:	Dr. Stephan Merz Inria Nancy

Tag der mündlichen Prüfung: 29.07.2017

Abstract

The idea of verifying the correctness of software has been brought up in the early days of computing, for example by Alan Turing in 1949 or Robert W. Floyd in 1967. James C. King extended upon the idea of manual verification by suggesting and working towards an implementation of a „verifying compiler“. King's idea has later been stated as a „Grand Challenge for Computing Research“ by Tony Hoare in 2005.

Idealism however is not the only reason for software verification. According to a study performed in 2002 by the U.S. National Institute of Standards & Technology, the U.S. economy suffers a loss of close to $60 billion per year due to software errors. Furthermore, errors have directly led to deaths or serious injuries, as well as failure or loss of equipment.

Within typical software engineering approaches, testing and static analysis tools are used to limit the possibility of software errors. Several holistic approaches to software development have been designed in order to keep bug counts low throughout the whole development cycle.

This thesis focuses on a more rigorous approach to the development of software: the usage of formal methods. In particular, it is concerned with improving the ProB model checker by augmenting its original explicit state model checker with a symbolic counterpart. In this way we broaden the scope of problems to which ProB can be applied.

As symbolic model checking makes heavy use of constraint solving, one aspect of this thesis is to implement methods to increase the performance of ProB's constraint solving kernel. This will be achieved by two means. First, by improving the constraint solver itself, mainly by lifting it from finite to infinite-state problems. Second, we developed an integration of ProB with the SMT solvers CVC4 and Z3, combining them into a single procedure.

Following, this thesis will present different symbolic model checking algorithms and evaluate them regarding their applicability to B and Event-B. Some suitable algorithms, namely Bounded Model Checking, k-Induction and IC3 were implemented inside ProB.

Both the improved constraint solver alone and the symbolic model checking algorithms will be evaluated on different examples ranging from solving B predicates, proving, disproving and SMT solving to model checking of academic and industrial specifications.

Zusammenfassung

Die Idee die Korrektheit von Software zu beweisen kam bereits zu Beginn des Computerzeitalters auf. Sie findet sich beispielsweise 1949 bei Alan Turing oder 1967 bei Robert W. Floyd. James C. King erweiterte die Idee durch seinen Vorschlag statt manueller Verifikation einen so genannten „verifying compiler“, also eine automatische Software, zur Verifikation zu verwenden. Seine Idee der automatischen Verifikation wurde 2005 von Tony Hoare in seine Liste der „Grand Challenges for Computing Research“ aufgenommen.

Idealismus ist aber nicht der einzige Grund für die Verifikation von Software. Laut einer Studie die 2002 vom U.S. National Institute of Standards & Technology durchgeführt wurde erleidet allein die Wirtschaft der USA einen Verlust von fast 60 Milliarden Dollar pro Jahr auf Grund von Softwarefehlern. Softwarefehler haben bereits zu Todesfällen oder Verletzungen geführt, zu Versagen oder Verlust von Equipment beigetragen oder diese direkt verursacht.

Häufig werden Tests oder statische Analysen verwendet um Fehler zu entdecken oder ihr Auftreten unwahrscheinlich zu machen. Ausserdem werden Entwicklungsmodelle verwendet, die die Anzahl von Softwarefehlern gering halten sollen.

In dieser Arbeit geht es um einen strikteren Ansatz zur Entwicklung von korrekter Software: Formale Methoden. Insbesondere beschäftigt sich diese Arbeit mit der Verbesserung des Model-Checkers ProB. Ein Ziel ist es, den bestehenden Model-Checking-Algorithmus durch einen symbolischen Algorithmus zu ergänzen. So kann ProB auf eine größere Klasse von Problemen angewendet werden.

Da die Effizienz von symbolischen Model-Checking-Algorithmen direkt von der Effizienz des verwendeten Solvers abhängt ist die Verbesserung des Kernels von ProB ein weiterer wichtiger Aspekt dieser Arbeit. Hier wurden zwei wesentliche Dinge umgesetzt. Zum einen wurde ProB's interner Constraint Solver erweitert, so dass er mit Problemen auf unbegrenzten Mengen umgehen kann. Weiterhin wurde eine Integration zwischen ProB und SMT Solvern entwickelt.

Anschließend werden verschiedene symbolische Model-Checking-Algorithmen vorgestellt und dann auf ihre Anwendbarkeit auf B und Event-B hin untersucht. Mehrere geeignete Algorithmen, wie Bounded Model Checking, k-Induction und IC3 wurden in ProB implementiert.

Sowohl der Kernel als auch der symbolische Model-Checker werden auf verschiedene Arten evaluiert. Dazu gehört das Auswerten von Prädikaten, das Finden von Gegenbeispielen und das Beweisen von Theoremen. Darüber hinaus wird ProB auf SMT Probleme angewendet. Zuletzt wird die Performance der Model-Checking-Algorithmen auf Basis verschiedener akademischer und industrieller Beispiele ausgewertet.

Acknowledgments

Writing this thesis would not have been possible without the constant help and support granted to me by numerous people. First, I would like to thank everybody working at the chair of software engineering and programming languages at the Heinrich-Heine-University. During the years I spent here, I always felt welcome and as part of a team.

In particular, I would like to thank my supervisor Michael Leuschel for supporting me and my research from bachelor's thesis to dissertation. Additionally, I would like to thank Claudia Kiometzis for having my back whenever I had to deal with the pitfalls of bureaucracy.

As customary, this thesis is based on papers published during my PhD studies. Writing those would have been impossible without the support of my coauthors Jens Bendisposto and Michael Leuschel. Both helped me get into publishing my research by improving my understanding of how to write scientifically and enjoyable at the same time.

My research has also benefited from various developers providing us with Event-B models and case studies as well as discussions and feedback for which I am grateful. Additionally, I would like to express my gratitude for all the other researchers and PhD students I got into contact with at conferences, workshops and summer schools. Without lively discussions, convincing arguments and the occasional outright disagreement, this work would not be what it is today.

Furthermore, all the articles incorporated in this thesis have benefited from suggestions and constructive criticism given by reviewers. For their time and work I would like to thank the anonymous reviewers of SETS 2014, SEFM 2015, ABZ 2016 and WLP 2016 as well as the Science of Computer Programming journal.

The dissertation itself has been proofread by Jens Bendisposto, Philipp Körner and Patricia Krings. Your reviews helped me fix many errors. Thank you!

Last, my heartfelt thanks go to my wife Marion, my sister Patricia, my parents, my family and all my friends for their constant support and encouragement. They all helped me get back on track every time I just wanted to print, rumple and throw away my half-written dissertation. I am grateful for everybody who put up with the annoyances my commitment to this work used to cause at times.

Contents

Part I

Introduction

1

Motivation and Goals

The idea of verifying the correctness of software has been brought up in the early days of computing, for example by Alan Turing in 1949 [180, 150] or Robert W. Floyd in 1967 [79]. James C. King extended upon the idea of manual verification by suggesting and working towards an implementation of a "verifying compiler" [109].

> One can dream of routinely using a verifying compiler as an everyday tool. In the context of this idea our work has been extremely modest and must be considered as a small first step. We only hope that, indeed, this has been a first step of a progression which will allow this dream to come to fruition. (James C. King [109, p. 3])

King's idea has later been stated as a "Grand Challenge for Computing Research" by Tony Hoare [100] in 2005.

> A verifying compiler uses mathematical and logical reasoning to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the code of the program. The compiler will work in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components. The capabilities and performance of the verifier will be demonstrated by application to a broad selection of legacy code, chiefly from open sources. (T. Hoare [100, p. 65])

Idealism however is not the only reason for software verification. According to a study [156] performed in 2002 by the U.S. National Institute of Standards & Technology the U.S. economy suffers a loss of close to $60 billion per year due to software errors. Infamous ones include the loss of the first Ariane 5 carrier rocket [67] and the crash of the Mars Climate Orbiter [173]. Beyond loss of equipment or money, software errors have directly led to deaths or serious injuries, as happened with the Therac-25 radiation therapy machine [130].

For general purpose software, reliability is primarily ensured using various testing techniques. For the development of safety critical software, e. g., software

used for medical devices or transportation, one often goes beyond testing and employs formal development techniques. Generally speaking, a formal method for software development takes a thorough and holistic approach towards software correctness.

Instead of testing the software at hand, a formal method relies on specification, mathematical and logical reasoning as well as thorough theoretical analysis. Ultimately, formal methods work towards producing a correctness proof that assures that a piece of software behaves according to its specification.

A technique that is heavily used when employing formal methods is model checking, i. e., the systematic exploration of the state space of some software product. This can be performed by various tools, one of them being ProB, the animator and model checker for the B method developed by the software engineering and programming languages department at the University of Düsseldorf.

ProB can already be used to analyze the state spaces of large software systems thanks to techniques like partial order reduction or symmetry breaking. Furthermore, ProB includes a constraint solver for the B and Event-B languages. The solver is used as a backend for the model checker. Additionally, it has successfully been used for different tasks outside of model checking. Examples include data validation and timetabling. We will give more detail on the features and implementation details of ProB in Section 2.3.

All this machinery is based on models having a finite state space. If necessary, this is enforced by limiting the size of sets or the domain of constants and variables.

In this thesis, ProB should be enabled to successfully model check infinite state spaces. This involves solving constraints involving variables featuring infinite domains, i. e., they have an infinite amount of possible values. This can obviously not be achieved by extending the existing explicit state model checker, as enumerating an infinite amount of states is impossible. Hence, a new symbolic model checker has to be implemented inside ProB based on the existing constraint solving facilities.

Summarizing, the goals of this thesis are as follows:

- Strengthen ProB's constraint solver to enable it to solve infinite domain constraint satisfaction problems. This is crucial in order to use it as one of the solving engines for a symbolic model checker.
- Assess the performance of the extended constraint solver. Based on benchmark data, find ways to improve. Additionally, integrate other solvers or provers if necessary and possible.
- Evaluate and decide on symbolic model checking algorithms that are suited for B and Event-B.

- Implement the selected algorithms in ProB using ProB's strengthened constraint solver. Evaluate whether additional solvers are needed.
- Improve performance by integrating abstraction techniques where needed.
- Perform an empirical evaluation of said algorithms comparing it to ProB's explicit state model checking algorithm.

In the upcoming chapter we will give an overview of formal methods in general and the B method in particular. We will describe model checking in greater detail in Section 2.2. In particular, we will focus on the capabilities of ProB in Section 2.3. Later on we will briefly introduce two further techniques used in this thesis, namely constraint programming in Section 2.4 and SMT solving in Section 2.5.

Each of the following chapters is devoted to one of the goals mentioned above. First, in Chapter 3, several extensions made to the ProB kernel are introduced. Among those are extensions regarding handling of infinite domains, rendering ProB's constraint solver usable as the backend of a symbolic model checking algorithm. The chapter mainly deals with techniques used to strengthen ProB itself. Thus, the techniques presented are beneficial for applications other than symbolic model checking as well.

The following chapters show two applications of the extended constraint solver in greater detail. Chapter 4 highlights how ProB can now be used as a prover owing to the techniques introduced before. The chapter mostly serves as an empirical evaluation of the constraint solver as well, allowing us to decide on further research directions.

A second empirical evaluation of the extended constraint solver is given in Chapter 5. It introduces a translation from SMT-LIB to B, allowing us to evaluate the performance of ProB on problems taken from SMT-LIB's collection of benchmarks. These are fundamentally different from the benchmarks in the former chapter. In consequence, Chapter 5 accounts for an additional analysis of ProB's performance as solver and prover.

As the constraints occurring in symbolic model checking techniques are quite involved, ProB's constraint solver alone turned out to be insufficient. Hence, we developed an integration between ProB and the SMT solver Z3. Chapter 6 outlines how the integrated solver works and evaluates it on several proof obligations.

Afterwards, Chapter 7 will discuss different symbolic model checking algorithms. It will highlight the advantages and disadvantages of the algorithms and justify the decision to focus on bounded model checking, k-Induction and the IC3 algorithm.

Following, Chapters 8 and 9 explain how infinite-state model checking of B and Event-B models can be done using said algorithms. The chapters heavily rely on the results obtained in the previous ones, incorporating all the improved solving techniques into a symbolic model checking engine for B and Event-B. Again, an empirical evaluation on different models has been performed.

We study the applicability of our results to other formalisms such as TLA^+ or Z in Chapter 10. Overall conclusions and future work will be given in Chapter 11.

An investment in knowledge pays the best interest.
Benjamin Franklin

2 Background

This chapter will give the necessary background information and introduce the key technologies used in this thesis. First, Section 2.1 will introduce formal methods for software development in general. Two methods, namely B and its successor Event-B will be introduced in Section 2.1.1 and Section 2.1.2. Section 2.2 is dedicated to model checking and how it can be used to verify software and systems. One model checker, ProB will be introduced in Section 2.3.

As the algorithms developed in this thesis heavily rely on constraint solving using Prolog and its CLP(FD) systems, Section 2.4 will outline how constraint logic programming works. An alternative approach to constraint satisfaction, namely SMT solving, is the topic of Section 2.5.

2.1 Formal Methods

Generally speaking, formal methods are mathematical approaches towards software and systems development. While there is a multitude of different formal methods, they are all based on a common idea: thorough theoretical analysis throughout requirements engineering, specification, development and verification should lead to better software. Depending on the method and tools used, techniques could involve static analysis or simulation and model checking. Other methods focus on mathematical proof, either interactive or fully automatic.

Some formal methods only try to avoid specific errors or analyze certain parts of a system. Others try to integrate the whole development process, leading

to software that is correct by construction. One such method is the B Method introduced in Section 2.1.1. It is presented alongside its successor Event-B, with differences being discussed in Section 2.1.2.

2.1.1 The B Method

The B method [1] is a state-based formal method for the development of software following the correct-by-construction approach. It mainly consists of two components:

- The B language, used to write specifications, and
- The B method, the formal development approach using the B language.

Central to the B language is the *abstract machine*, holding the specification itself. It consists of the following components:

1. A number of *sets* which are treated as user-defined types. B features two kinds of sets: *Deferred sets* are sets which are not given upfront by enumerating their elements. If the elements are given, the set is called an *enumerated set*.
2. A number of *constants* together with a predicate called *properties* that constrains possible values, i. e., the constants are fixed to some values rendering the properties true. Afterwards, they do not change anymore.
3. A number of *variables* that represent the state of a software or system. Their initial values are given using the *initialisation*.
4. An *invariant* that specifies which states are considered safe. Usually, it assigns a type to each variable and gives accepted ranges for variable values.
5. *Operations* that define the behavior of a system. They can feature guarded execution, stating under which conditions an operation can be executed. Upon execution, an operation can change the system variables in several ways. These include assignments, if-then-else constructs and several other so-called *generalized substitutions*. See [1] for details.

The constants and variables together with their values form the state a system is in. Operations are used to define possible transitions between these states.

B has a rich type system as shown in Fig. 2.1[1]. Expressions can consist of arithmetic, set theory, sequences, trees and arbitrarily nested types. In addition to the given types for booleans, strings and integers, users can provide further types by defining disjoint enumerated or deferred sets. On top of the type system,

[1] Taken from my article [114].

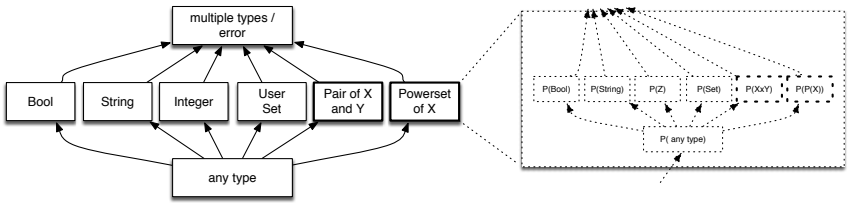


Figure 2.1: Classical B Type System

B features several higher-order constructs such as set comprehensions. Predicates like preconditions are first-order logic formulas over these expressions.

An abstract machine may include other machines in several ways, the main difference being the visibility of constants, variables and operations from the included in the including machine.

To handle complexity, the B method relies on refinement of abstract machines. Broadly speaking, development starts with an abstract machine that does not include many details. During refinement, details are gradually introduced in subsequent machines. For each refinement step the user has to prove that the more concrete machine is indeed a correct refinement of the abstract machine, i. e., properties proven correct on the abstract level hold on the concrete level.

Ultimately, refinement leads to an *implementation machine* that can automatically be translated into a programming language like C or ADA. For this to be possible, it has to include all the necessary details and it has to avoid data structures and constructs that are too high-level, such as sets. Refining and refined machine are connected by a *gluing invariant* that defines the relationship between state variables on different refinement levels.

Consistency and correctness of B machines is ensured by several proof obligations. Most of them make use of the weakest precondition calculus:

Definition 2.1.1 (Weakest Precondition)**.** For P a predicate and S an abstract machine statement $[S]P$ denotes the weakest precondition a state has to fulfill, such that the execution of S on that state is guaranteed to lead to a successor for which P holds. Note that the weakest precondition also has to ensure termination in presence of loops.

Instead of an abstract machine statement S, we can use the belonging transition predicate T.

Using this definition, we can give a proof obligation stating that statement S guarded by a precondition P may not lead to a violation of the invariant I when executed from a valid state:

$$I \wedge P \Rightarrow [S]I.$$

There are several other proof obligations that can occur at various stages of development, e. g., for proving well-definedness or the correctness of refinements. See [1] for details.

2.1.2 Event-B and Rodin

Event-B [2] is a successor of B, tailored towards system level design rather than software level development. In other words, Event-B is designed to model full systems, including hardware, software and environment. An integrated development environment for Event-B is provided on top of Eclipse by the Rodin project [3].

The key differences between classical B and Event-B are:

- Sequences and trees have been removed leading to a much simpler type system. Both can still be expressed using the remaining data structures together with additional axioms.

- With a recent addition, it has been made possible to extend both Event-B and Rodin by user-defined theories [135]. There is no real support for user-defined extensions to classical B.

- Instead of operations, Event-B relies on events. These are considerably simpler than operations as they omit the involved classical B substitutions like `case` statements or `if-then-else` constructs. An Event-B event only consists of guards and (simple) actions.

- The static part of a model has been split from the machine into a so-called *context*. A machine can include contexts, while a context can extend other contexts.

- Machines cannot be included anymore. An Event-B machine can only refine another machine.

- Unlike preconditions, the guards of an event can be strengthened during refinement. This allows removing behavior that was present in a more abstract machine. In consequence, deadlocks might be introduced.

Most of these changes contribute to the fact that Event-B has considerable simpler proof obligations. Overall this leads to proofs that are easier to produce and to comprehend.

2.2 Model Checking

Together with mathematical proof, model checking [53, 13] is one of the key techniques used in formal software development. Usually, the model checking problem is defined on transition systems.

Definition 2.2.1 (Transition System)**.** Following the definition of [13], a *transition system TS* is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

1. S is the set of states.
2. Act is the set of actions.
3. $\rightarrow \subseteq S \times Act \times S$ is the transition relation. It connects states to their successors by "executing" an action. Instead of the tuple $(s_1, a, s_2) \in \rightarrow$ one writes $s_1 \xrightarrow{a} s_2$.
4. $I \subseteq S$ is the set of initial states.
5. AP is a set of atomic propositions.
6. $L : S \rightarrow \mathbb{P}(AP)$ is the labeling function. It relates each state s to the set of atomic propositions evaluating to true over s.

Note that models written in B or Event-B represent such transition systems. I is given by the initialization, $\rightarrow$ is defined by the operations or events. The set of all states consists of all possible combinations of values of constants and variables. The propositions that should be verified are given for instance in the invariant.

Not every state in a transition system has to be connected to one of the initial states by a sequence of actions. In fact, certain states can exist without any actions related to them. Therefore, we have to distinguish between reachable and unreachable states. Again, the upcoming definitions follow those of [13].

Definition 2.2.2 (Reachability)**.** Given a transition system $(S, Act, \rightarrow, I, AP, L)$, a state $s \in S$ is called *reachable from* s_{start}, if and only if there are states $s_0, \ldots s_n$ and actions $a_1, \ldots a_n$, such that there exists an execution sequence

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$$

where $s_0 = s_{start}$, $s_n = s$ and $\forall_{i=0}^{n-1}\ s_i \xrightarrow{a_{i+1}} s_{i+1} \in \rightarrow$.

Now, we can define model checking itself:

Definition 2.2.3 (Model Checking)**.** The *model checking problem* is to verify whether a specification M, given as transition system, is a model for a formula ϕ. If ϕ is an invariant, this means to verify whether it holds in every state of M that can be reached by the initialization.

In this thesis, model checking always refers to checking of invariants as defined above. There are of course other kinds of formulas, such as liveness properties given in LTL or CTL [75], each with appropriate model checking algorithms.

For certain model checking techniques, we need to limit the number of states that exist in a transition system.

Definition 2.2.4 (Finite Transition System)**.** A transition system is called *finite* if S, Act and AP are finite. Otherwise, the transition system is called *infinite*.

In general, B and Event-B models are represented by infinite transition systems. In the following two sections, two different approaches to model checking techniques and algorithms will be introduced.

2.2.1 Explicit State Model Checking

The idea behind explicit state model checking is to solve the model checking problem by enumerating all reachable states of a system. To do so, an explicit state model checker keeps track of two sets of states:

- The set of visited states. These states have already been verified without detecting an invariant violation.
- A queue of states that have been reached but not yet verified.

An explicit state model checking algorithm for atomic propositions is outlined in Algorithm 2.1. To lift it to other properties, the check in line 6 would need to be replaced appropriately. The algorithm closely resembles the one described in [128] and used in ProB. It proceeds as follows: After the setup, the queue contains all the initial stats while the set of visited states is empty.

As long as there is a state in the queue, model checking has not yet finished. The state is selected from the queue and gets examined. First, the algorithm checks if the current state violates the invariant. If this is the case, an error has been found and can be reported to the user. If not, the algorithm computes all successor states. Those that have not been visited yet are added to the queue. The state itself is added to the set of visited states. Once all states have been traversed, the system is reported as safe.

Explicit state model checking techniques have a major drawback. They suffer from the so called "state space explosion" problem. For large systems with many transitions there is a combinatorial blow up that leads to exponentially growing state spaces. The larger a system becomes, the harder it is to completely traverse all states. For systems with an infinite number of states exhaustive traversal becomes impossible. There are several ways to reduce the number of states

Algorithm 2.1: Explicit State Model Checking

Data: Actions Act, Transition $\rightarrow$, Initial States I,
Labeling $L : S \rightarrow \mathbb{P}(AP)$, Property $P \in AP$
Result: true iff P holds

```
1  procedure boolean explicit_mc(Act, →, I, L, P)
2      queue := I
3      visited := ∅
4      while queue ≠ ∅ do
5          current := select_state_from(queue)
6          if P ∉ L(current) then
7              return false
8          end if
9          visited := visited ∪ {current}
10         foreach action ∈ Act do
11             queue := (queue ∪ {x | (current, action, x) ∈→}) \ visited
12         end foreach
13     end while
14     return true
```

that have to be considered. Those implemented by ProB will be discussed in Section 2.3.

In the following thesis, we will focus on *symbolic model checking* as a means to overcome the state space explosion problem.

2.2.2 Symbolic Model Checking

The general idea introduced with symbolic model checking [39] is to avoid the explicit enumeration of the states of a model. Instead, the state space is stored symbolically by using predicates to describe sets of states.

For example, the three states $x = 1$, $x = 2$ and $x = 3$ could be combined to the single state predicate $1 \leq x \leq 3$. Operations on these states can then be performed by modifying the predicate. Performing a simple increment operation would result in the three states $x = 2$, $x = 3$, $x = 4$, or $2 \leq x \leq 4$ if the state space is kept symbolically.

In addition, transitions can be computed symbolically as well by building predicates that logically connect the sets of states before and after execution.

The key idea behind symbolic model checking is to work on equivalence classes of states instead of on the concrete state space. States are combined into classes based on properties such as the status of the invariant in the states or the distance

to the initialization. This reduces the impact of the state space explosion problem on the performance of the model checking algorithm.

Obviously any symbolic algorithm has to keep a balance between abstraction and precision. Instead of $1 \leq x \leq 3$ which is equivalent to $x = 1 \vee x = 2 \vee x = 3$ one could have chosen to abstract away detail by using an approximation like $x \leq 3$.

The easy integration of abstraction and concretization techniques allows for an extension of symbolic model checking techniques to models featuring infinite state spaces.

Chapter 7 will give an overview of different symbolic model checking algorithms for safety properties. In particular, the IC3 algorithm will be explained in-depth in Section 7.3.5, as it is currently the most promising algorithm regarding infinite-state model checking of B and Event-B specifications.

2.3 ProB

ProB [128, 127] is an animator and model checker for the B method introduced in Section 2.1.1. It is developed by the software engineering and programming languages department of the Heinrich-Heine-University in Düsseldorf. The kernel is written in SICStus Prolog, extensions and other supporting software has been written in C, Java and other languages.

ProB has several key features:

- **Animation**, i. e., stepwise execution of a B model. This is especially useful during the development of a formal model, as it allows to quickly check for expected behavior.
- **Verification**, i. e., ProB can be used to detect counterexamples to correctness, refinement relations and other proof obligations. In certain cases, ProB can even be used to prove correctness as we will show in Chapter 4.
- **Model Checking** using an explicit state algorithm as introduced in Section 2.2.1. The model checking algorithm used in ProB is more refined than the one given in Algorithm 2.1, as it features both depth- and breadth-first search as well as heuristics for search direction [21].
- **Visualization**, i. e., ProB can generate several graphical representations of a model, the state space or certain predicates. With BMotionStudio, interactive visualizations for B and Event-B models can be created using ProB [119]. This simplifies understanding a model and provides a comprehensible entryway to the model for non experts.

An important difference between ProB and other animators for the B method is that ProB tries to provide automatic animation as far as possible. To do so, ProB tries to infer feasible values for constants and variables as well as parameters that allow the execution of operations and events. This relieves the user from providing feasible values himself. Furthermore, it avoids possible values being overlooked.

As already stated, ProB's model checking kernel relies on an explicit state model checking algorithm. It features several techniques to reduce the size of the state space and to handle large and complicated models.

Among these are:

- Partial order reduction [85, 157, 181], a technique based on the observation that for certain transitions the order of execution does not matter. If this is the case, the state space can be reduced by only analyzing one order of execution. ProB implements partial order reduction as outlined in [66].
- Some models feature a state space that is highly symmetrical, e. g., several branches of the state space could be merged as they only differ in the selection of a constant. This symmetry can sometimes be detected and broken. Different techniques exist and have been evaluated and implemented for B and Event-B inside ProB [172].
- Lastly, [21] uses parallelization and distributed computing to be able to handle large state spaces.

In addition to classical model checking ProB can perform several so-called "constraint-based checks" [91]. To some extent, these are comparable to symbolic model checking. For instance, the constraint-based invariant check tries to find two states connected by an operation, such that the invariant holds in the first but not in the second state. If this were the case, the model could not be proven correct. There is however a key difference to model checking: We do not know if the first state is reachable from the initialization.

There are several versions of ProB available:

- A command line interface that can be included in scripts and is used as a backend for other versions of ProB.
- A graphical user interface written in Tcl/Tk. This has long been the main interface of ProB. Figure 2.2 shows a screenshot.
- An embedded version of ProB for use inside the Rodin platform.
- Recently, a new frontend for ProB has been in development under the name ProB 2. It focuses on scriptability and provides a user interface based on JavaFX.

The B language aside, ProB supports the Event-B language introduced in Section 2.1.2, CSP and CSP ‖ B [159], Z [158] and TLA$^+$ [93].

Of course there are also other model checkers for the B and Event-B languages: Eboc [141], pyB [185] and JeB [186] also rely on explicit state model checking.

All developments presented in this thesis are done within ProB. In particular, ProB's constraint solver will be used as a backend for the symbolic model checking algorithms.

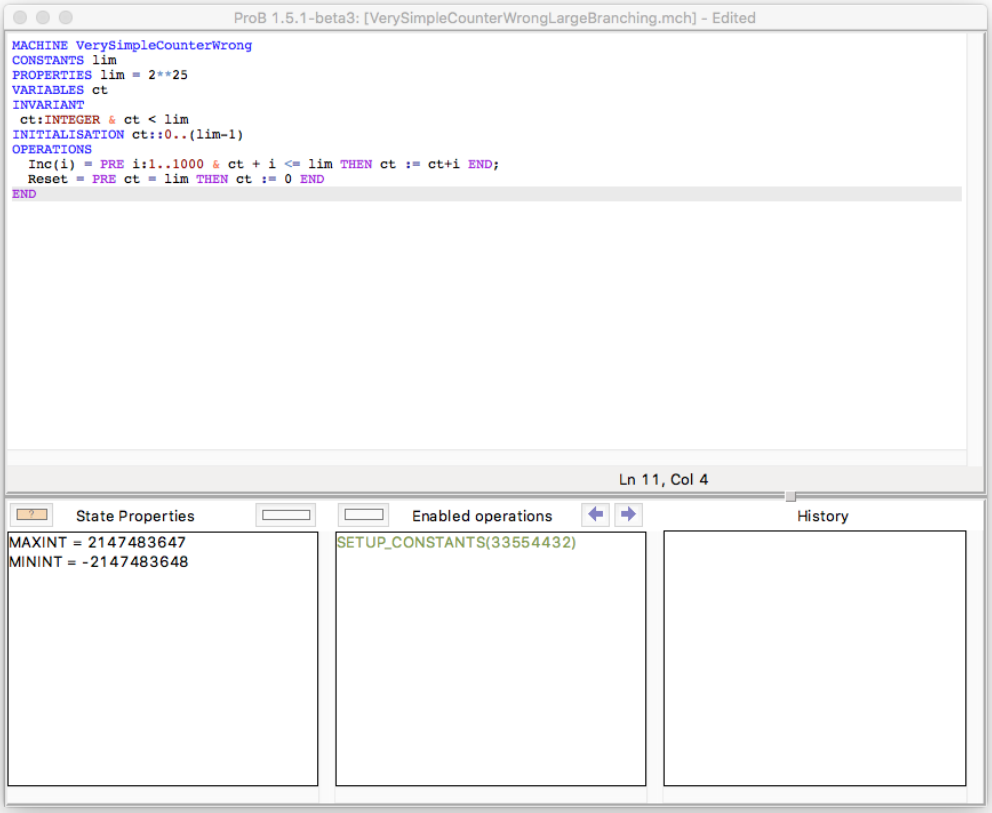


Figure 2.2: ProB Tcl/Tk Interface

2.3.1 Constraint Solving Kernel

The ProB kernel can be viewed as a constraint-solver for the basic datatypes of B and the various operators on it. It heavily relies on the SICStus Prolog CLP(FD) system [42] which follows the general implementation scheme of [104], which we will discuss in Section 2.4. It supports booleans, integers, user-defined base types, pairs, records and inductively: sets, relations, functions and sequences.

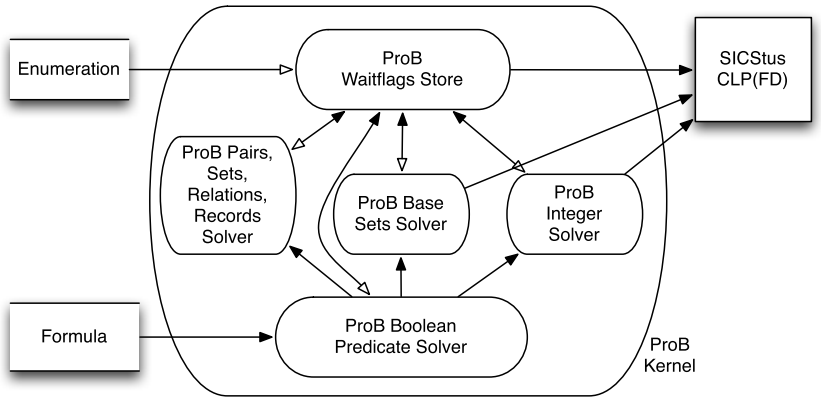


Figure 2.3: Modules of the ProB Kernel

These datatypes and operations are embedded inside B predicates, which can make use of the usual logical connectives ($\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg$) and typed universal ($\forall\, x.P \Rightarrow Q$) and existential ($\exists\, x.\, P \wedge Q$) quantification.

An overview of the various solvers residing within the ProB kernel can be seen in Fig. 2.3. Input formulas are parsed and given to ProB's predicate solver. Inside, the AST is traversed and subpredicates are delegated to more specialized solvers, e. g., solvers for set expressions or integers.

The solvers have two means of communication: truth values between solvers are passed on using reification variables introduced by the predicate solver. At the same time, the "Waitflags Store" is used for scheduling. The different solvers are integrated using "waitflags" to control which constraints should be tackled, i. e., propagation and enumeration are triggered by setting the appropriate flags and passing them to the individual solvers.

Enumeration is controlled by setting up coroutines at various choice points. For more fine-grained enumeration, coroutines might be set up together with a waitflag that allows the solver to trigger delayed constraints. A priority can be attached to any waitflag in order to further enforce particular execution sequences.

The integer part of the solver is mostly based on the CLP(FD) library of SICStus Prolog, which we will describe below. Custom extensions and solvers are implemented for pairs, sets, relations and records. User-defined sets, i. e., enumerated and deferred sets are handled individually as well.

Furthermore, support for quantifiers has been added on top of CLP(FD), again controlled by the top-level predicate solver. All techniques presented in this thesis are fully integrated with the solving kernel and its features.

2.4 Constraint Logic Programming

Constraint programming is a paradigm that is different from the well-known imperative programming. The key difference is that in constraint programming, the relationship between different program variables is expressed as constraints, i. e., one could state that one variable has to be strictly larger than another for the solution to be regarded as correct.

In contrast to imperative programming, the developer does not implement the different steps a program should perform to compute the solution. Instead, one relies on a constraint programming library to find a solution to the given problem specification.

Usually, these libraries introduce constraints into a given host language that is used for other parts of the program. In particular, constraint logic programming refers to embedding constraint programming into a logical programming language.

This thesis will employ constraint programming techniques as offered by the CLP(FD) library of SICStus Prolog. On a high level, the library works as follows:

- It builds up a graph structure called the *constraint graph*. The graph includes a node for every variable. Edges are given by the constraints, e. g., $x < y$ results in an edge connecting x and y.
- It attaches a *domain* to each variable. For instance, this could be an interval $[a, b]$. Ideally, the domain includes exactly those values that could still contribute to a valid solution.
- The domain is gradually refined. In a first step, *node consistency* is ensured. This involves all constraints that refer to a sole variable. A constraint like $x < 5$ updates the upper bound of a variable to be less than five as well. Note that values are removed from domains only after they cannot be part of a solution anymore. Hence, a domain is never expanded. With each constraint involved, it can only get smaller.
- Next, the algorithm tries to achieve *arc consistency*. That is, for each edge we ensure that each element in a domain has a matching element in the opposite domain. If there is no matching element, i. e., the element cannot contribute to a solution anymore, the element is removed from the domain. One algorithm to achieve arc consistency is the AC-3 algorithm by Mackworth [136].
- If during these steps a domain is emptied completely, the constraint satisfaction problem has no solution.

There are of course higher levels of consistency one could think about. For instance, a constraint like "a, b and c are all different" involves more than two variables and is thus out of the scope of arc consistency. Several algorithms to solve such constraints exist [8].

A general overview of the internals of a constraint logic programming system is given in [104]. In particular, those of SICStus Prolog [42] as well as of SWI Prolog [178] follow the general ideas. Additionally, [98] explains consistency techniques and gives two applications that show the expressiveness of constraint programming languages. A more recent look into different techniques to achieve consistency and an introduction to different search algorithms can be found in [123].

Pure constraint propagation using the consistency algorithms is incomplete, i. e., for certain inputs it will report neither satisfiability nor unsatisfiability. Take for example a constraint programming system using intervals as domains. When called on the query

$$x \in [1, 10] \wedge y \in [1, 10] \wedge X = 2 * Y$$

a simple CLP(FD) system as outlined above can only shrink the domains of x and y to

$$x \in [2, 10] \wedge y \in [1, 5] \wedge X = 2 * Y.$$

However, the domain of x could be smaller as only even numbers are solutions to the equation. Due to the interval reasoning, the propagation techniques explained above are too weak to discard the odd numbers.

In other cases the incompleteness may hide the unsatisfiability of a predicate, e. g., in

$$x \in [0, 1] \wedge y \in [0, 1] \wedge z \in [0, 1] \wedge x \neq y \wedge y \neq z \wedge x \neq z \tag{2.1}$$

the domains are not shrunken at all, as each combination of two variables is arc consistent.

In order to query the domains for an actual solution, CLP(FD) systems feature *labeling*: One of the variables is selected and assigned a value taken from its domain. Different options are available to control both the selection of the variable and the selection of the next value to assign to it.

This change propagates to the other variables using the consistency techniques explained above. If one of the domains is now empty, the system backtracks and assigns a different value to the variable. Otherwise, the next variable is selected and processed in the same manner.

Again, let us take a look at the constraint given in Eq. (2.1). Upon labeling, the variable x is set to 0. Achieving arc consistency for $x \neq y$ shrinks the domain of

y to $[1, 1]$. This change triggers $y \neq z$ to update the domain of z to $[0, 0]$. Arc consistency cannot be achieved for $x \neq z$. The system backtracks and assigns x to 1. Again, no solution can be found.

Exhaustive labeling makes CLP(FD)-style constraint solving complete. Obviously, labeling can only be done exhaustively if all domains are finite. In order to overcome this limitation, we had to implement different extensions to CLP(FD) to lift ProB's constraint solver to infinite domains. We will discuss them in Chapter 3.

2.5 SMT-Solving

Satisfiability Modulo Theories [19], or SMT for short, is concerned with checking the satisfiability of first-order logic formulas with respect to some background theory $\mathcal{T}$. A background theory can for example fix the interpretation of predicates. For several applications, one is interested in the satisfiability of a predicate like

$$x < 3 \wedge \ y > x \wedge \ \neg \ (x + y > 7)$$

using the usual interpretations of $<$, $>$ and $+$.

Using a solvers for general first-order logic, this can be achieved by explicitly encoding the semantics of $<$, $>$ and $+$ as axioms inside the formula. Using solvers tailored towards the theories in question can often provide better performance and is less cumbersome.

Currently, there are two major approaches towards SMT solving [19]:

- An *eager* translation of the input predicate into a propositional formula. The semantics of any background theory used are encoded as well. Both are submitted to a SAT solver. This approach makes all theory specific propagation steps available to the solver at once. This might lead to faster propagation and thus expedite solving a formula. However, a translation to propositional logic usually involves a blowup in formula size.
- The *lazy* approach where theory specific solvers are applied to certain parts of the predicate while the overall structure is solved by any SAT solver. This makes communication between different theory solvers and the SAT solver a necessity and causes overhead. The main advantage is that the theory solver can apply specialized algorithms using background knowledge from the given theory.

Chapter 6 will describe the integration of the SMT solvers CVC4 [16] and Z3 [60] in ProB. As both CVC4 and Z3 rely on the lazy approach, we will look at its basics below. For details regarding the eager approach, see [19].

Algorithm 2.2: DPLL($\mathcal{T}$)

```
   Data: T-formula φ, T-assignment μ
   Result: true iff φ is satisfiable
 1 procedure boolean DPLL(T) (φ, μ)
 2     if T-preprocess(φ, μ) returns conflict then
 3         return false
 4     end if
 5     φ^p := TB(φ)
 6     μ^p := TB(μ)
 7     while true do
 8         T-decide_next_branch(φ^p, μ^p)
 9         while true do
10             status := T-deduce(φ^p, μ^p)
11             if status == Sat then
12                 μ := BT(μ^p)
13                 return true
14             else if status == Conflict then
15                 level := T-analyze_conflict(φ^p, μ^p)
16                 if level == 0 then
17                     return false
18                 end if
19                 T-backtrack(level, φ^p, μ^p)
20             else
21                 break
22             end if
23         end while
24     end while
```

One of the standard algorithms for SMT solving is the DPLL($\mathcal{T}$) algorithm, named after the DPLL algorithm used for SAT solving. The algorithm according to [19] is given in Algorithm 2.2. It starts with a formula φ that contains predicates in some theory $\mathcal{T}$. The algorithm allows to submit a (partial) assignment to the variables in φ. In the general case, the initial assignment is empty. Practically all SMT solvers start the search for a solution with a preprocessing step, called $\mathcal{T}$-preprocess in Algorithm 2.2. This involves simplification steps and other rewriting rules. The preprocessing step may modify both the theory and the boolean level.

If no conflict is detected during simplification, the propositional formulas φ^p and μ^p are updated according to the current state of the solver. This involves (at least partially) translating theory knowledge to the boolean level, as indicated by $\mathcal{TB}$.

Afterwards, the algorithm enters a loop:

- It decides on the next literal to branch on. This represents the SAT solver selecting a literal to set to true or false. In DPLL($\mathcal{T}$) however, the proposition variable selection might take theory knowledge into account.
- In the inner loop, the resulting propagation steps are performed:
  - $\mathcal{T}$-deduce performs boolean propagation steps on φ^p and τ^p. If this leads to a conflict, we analyze which theory literals are represented by the boolean conflict. In particular, $\mathcal{T}$-analyze_conflict is called to determine where the conflict occurred and how far the solver has to backtrack. The algorithms backtracks accordingly or returns false if the conflict occurred on the top-level.
  - Otherwise, if $\mathcal{T}$-deduce produces a satisfying assignment on the boolean side, the assignment is transferred to the theory solver. This is indicated by the call to $\mathcal{BT}$. If the assignment is a model in the theory as well, the algorithm returns sat. If the assignment leads to a conflict in the theory solver, $\mathcal{T}$-deduce reports the conflict and the algorithm has to backtrack again.
  - If all deduction and propagation steps have been performed and neither a solution nor a conflict have been inferred, $\mathcal{T}$-deduce is unable to produce a result. The solver has to select a new literal in the next iteration.

Now, let us have a look at an example: let us try to use the DPLL($\mathcal{T}$) algorithm to check if there is a solution to the predicate $\varphi = x > y \wedge y > x$. We do not submit an initial assignment, hence μ is empty. To focus on the algorithm itself, we assume the conflict is not detected by a preprocessing step.

We now have to find a boolean representation of our predicate. Do to so, we represent the truth value of $x > y$ by the boolean variable b_1, and the truth value of $y > x$ by b_2. The resulting boolean predicate is $b_1 \wedge b_2$. The algorithm decides to branch on b_1. Obviously, for the whole predicate to be true, b_1 has to be set to true. Furthermore, we can infer that b_2 has to be set to true as well.

$b_1 = \top \wedge b_2 = \top$ is a satisfying assignment for the propositional formula. However, on the theory level there is a conflict, as both $x > y$ and $y > x$ cannot be true at the same time. $\mathcal{T}$-decide returns the conflict and the algorithm has to backtrack to the top-level. It can thus report unsatisfiability for the given predicate.

The example features only one theory, namely inequality of integers. Quite often useful constraints feature a combination of different theories, such as integers, functions, arrays and so on. One of the most widely employed methods to combine theories is the Nelson-Oppen combination [153]. A survey on other methods for combining decision procedures can be found in [139].

As can be seen, the techniques behind SMT solvers are fundamentally different from the techniques employed by CLP(FD) solvers as introduced in Section 2.4. Both Chapter 4 and Chapter 6 will discuss how ProB employs the different approaches for proof and disproof. Performance will be evaluated using several benchmarks. Chapter 6 describes the integration of Z3 into ProB and later argues towards an integrated solver using both.

There are, however, other SMT solvers that could have been used as well. Notable representatives are Boolector [154], CVC4 [16], MathSAT [50], veriT [33] and Yices [70]. Out of these, veriT and Yices have already been used to discharge Event-B proof obligations [71, 72].

Part II

Improving ProB's Constraint Solver

It's gonna be the future soon
And I won't always be this way
When the things that make me weak and strange
Get engineered away

Jonathan Coulton, "The Future Soon"

3

Constraint Logic Programming over Infinite Domains

In this chapter we will introduce several extensions implemented on top of SICStus Prolog's CLP(FD) system, intended to lift it to infinite domain constraint satisfaction problems as they often occur in symbolic model checking algorithms.

The chapter is based on our paper "Constraint Logic Programming over Infinite Domains with an Application to Proof" [115]. For information regarding authors and their individual contributions see Appendix C.

3.1 Introduction and Motivation

Various techniques can be used to solve constraints expressed in specification languages like B, Z, TLA, Alloy, or VDM. Two popular techniques, namely SMT solving and constraint logic programming have been introduced in Sections 2.4 and 2.5. Another popular technique is SAT solving, made popular for use with specifications by Alloy [103].

So far, these techniques were often limited to first-order constraints and unable to deal with unbounded data values. Let us examine the simple constraint $x = 10 * 10$ over the integer variable x. To solve this constraint using SAT solving the integer x has to be represented by propositions, i. e., as a bit-vector. For this, it is important to know how many bits are needed, both for x itself and

for intermediate values that can occur while computing x. If one chooses too few bits, then the SAT solver may fail to find a solution where one exists, or report a solution where none exists (in case overflows are not detected).[1] Also, the encoding of values as bit vectors reaches its limits for more involved types like relations over large domains or higher-order values.

Take for example a relation $r \subseteq \mathbb{P}(D) \times \mathbb{P}(D)$ which takes a set of elements over a domain D and another set of elements over D. If D is of size 10, we need $2^{20} = 1.048.576$ bits to represent a possible value of r. If D is of size 20, we need $2^{40} = 1,099,511,627,776$ bits, i. e., already 128 Gigabyte to store one variable.

The contributions of this chapter are as follows:

- An extension of classical constraint logic programming techniques to infinite domains will be presented throughout Section 3.3.
- In particular, how ProB checks the exhaustiveness of occurring enumerations is decribed in Section 3.3.1.
- An algorithm for random enumeration of large domains is outlined in Section 3.3.2.
- Additional reasoning rules accompanying CLP(FD) are given in Section 3.3.3.

3.2 Constraint Solving

The key challenges when solving constraints in high-level languages such as B are universal and existential quantifications as well as set comprehensions and lambdas. Each can be arbitrarily nested and is not limited to finite values.

So far, we have tried different approaches to constraint solving as extensions to the CLP(FD)-based kernel of our model checker ProB in order to enable it to handle infinite domain problems as they typically occur in symbolic model checking algorithms: In prior work, a translation to SAT via Kodkod [160] has been developed. In addition, in Chapter 6, we will describe how to integrate SMT solvers such as Z3 [60].

The different techniques each have their own strengths and weaknesses: While there are highly efficient algorithms for SAT solving, encoding of higher-order constraints is often not feasible. Usually, the domain of integers has to be limited in order to allow bitlevel encoding of arithmetics.

[1] Alloy recently has added overflow detection; but in case no model was found we do not know whether an overflow may have prevented finding a solution.

This is even more problematic if higher-order logic or set theory come into play. Due to a combinatorial blowup, resulting SAT constraints contain too many variables and become unsolvable.

SMT solvers on the other hand rely on decision procedures for different underlying logics. Following the DPLL($\mathcal{T}$) algorithm, these solvers enumerate predicate values and try to infer logical consequences. Hence, it is easy to extract a proof from an unsatisfiable query while it is difficult to extract a model out of a satisfiable one.

In contrast, CLP(FD) systems use constraint propagation and are more focused on valuations rather than predicates. In fact, they show the opposite behavior: a satisfiable query always returns a model. At the same time it is difficult to extract proof of unsatisfiability.

3.3 Technique

In the following sections we describe how we extended CLP(FD) to enable handling of infinite domains and quantifiers. In Section 3.3.1, we will explain how we track enumeration of CLP(FD) variables. Afterwards, we introduce a way to randomize the enumeration of large intervals in Section 3.3.2.

For simplicity, we will discuss our techniques on a small interpreter supporting only integer variables with arbitrary and possibly infinite domains together with arithmetic expression on them, negation and nested existential and universal quantification.

The interpreter represents a simplified version of the enumeration detection employed in ProB. Instead of a single single scope, ProB uses nested enumeration scopes for a more fine-grained detection of enumeration and its consequences. Furthermore, ProB relies on different exceptions to communicate non-exhaustive enumerations between scopes and provides fallback behavior.

3.3.1 Detection and Categorization of Enumeration

It is common for constraint satisfaction problems that domain propagation alone is not enough to infer values for participating variables. This might be due to an underspecified problem or limitations in constraint solvers, e. g., global constraints that are too expensive to check. Usually, constraint solvers rely on enumeration of possible values if all other methods fail.

However, there are some key difficulties:

1. Enumerating all values is only possible for reasonably sized domains. Even for finite but large domains, exhaustive enumeration can be impossible due to computational limitations. Obviously, the same holds for infinite domains.
2. Depending on the scope of negations, quantifiers and other (nested) constructs finding a solution for a variable might imply both satisfiability and unsatisfiability of a (sub-)constraint.
3. Hence, if a solution is found it is not immediately clear if enumeration can be stopped.

We intend to overcome these limitations by tracking the scope of enumerations, i. e., tracking in which contexts enumeration occurs. We distinguish the following types of enumerations, based on the effect on the overall solver result:

- Enumeration does not occur. The result is not influenced, e. g., when no valuation is found the formula is unsatisfiable.
- Enumeration is exhaustive. In this case, all possible values for a variable were considered. If no valuation is found, the formula in question is unsatisfiable.
- Enumeration occurs and is not exhaustive. In this case, we cannot directly infer if the formula is satisfiable or not and have to examine the context (aka scope) in which the enumeration occurred.

Figure 3.1 shows the nesting of enumeration scopes for two simple predicates. The outer variable x is quantified existentially in both cases. In the first one, we can enumerate all possible values exhaustively. In the second case non-exhaustive search is the only possibility, as the domain of x is infinite. Since we only need to find one solution anyway, partial enumeration is not a problem. The inner variable y can be enumerated exhaustively. In the first example, we have to do so in order to validate the universal quantification. In the second example, exhaustive enumeration is possible but not necessary.

For further examples, take a look at the following constraints:

- `x*x = 10000` is true, a solution (`x = -100`) can be directly computed. Domain enumeration is only needed to pick one of the two possible values.
- Using backtracking, we can find all solutions. As above, domain enumeration is only needed to pick both values of `x` in `{x|x*x=10000} = {-100,100}`.
- In contrast, enumeration is necessary to find a solution to `x>10000 & x mod 1234 = 1` as propagation of `mod` does not render the domain of `x` finite. However, the solution found is sound, enumeration did not influence the result.

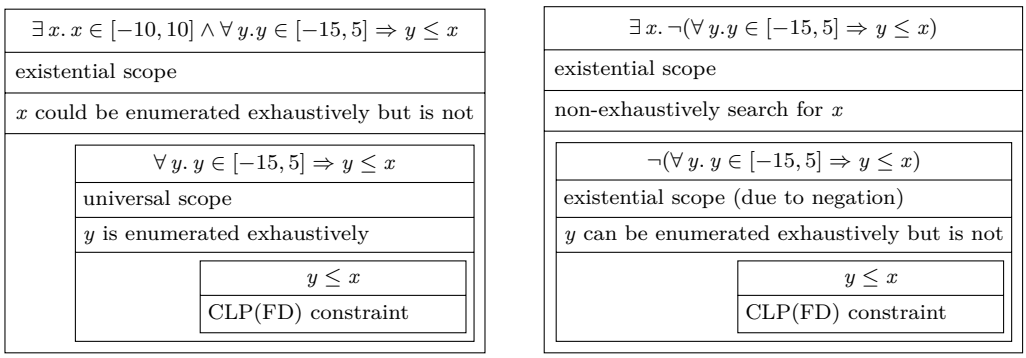


Figure 3.1: Nested Enumeration Scopes

- As a consequence, we cannot compute all solutions. Our approach is unable to solve {`x|x>10000 & x mod 1234 = 1`}.
- For certain unsatisfiable constraints, such as `x*x = 10001`, CLP(FD) detects unsatisfiability by domain propagation. No enumeration occurs, the predicate is guaranteed to be false.
- In contrast, no solution is found for `x>10000 & x mod 1234 = 1 & x*x = 10*x`. Common propagation rules such as the ones used in SWI Prolog's CLP(FD) [179, 9] are too weak to conclude $x = 0 \vee x = 10$ from $x * x = 10 * x$ as long as there is no upper bound attached to x. In consequence, we cannot detect unsatisfiability as we have to (partially) enumerate the infinite domain of `x`. Since no solution has been found, we cannot ignore the fact that enumeration occurred. Indeed, since the domain of `x` was only enumerated partially, we cannot conclude that the predicate is false. Indeed, it might still be true.

Setting up Constraints

In the example interpreter, constraints are set up using two Prolog predicates, `solve` for positive and `solve_not` for negative constraints. CLP(FD) constraints are immediately set up. Part of the interpreter is shown in Listing 3.1, the complete source code can be found in Listing B.1.

To control enumeration, we pass around two "waitflags", i. e., Prolog variables used to trigger the execution of coroutines by grounding them. The first one is used to control setup of constraints and enumeration of variables that are existentially quantified, i. e., they are introduced by $\exists$ or $\neg\forall$. The other one does the same for universal quantification.

Listing 3.2 shows the implementation in our simple solver. Once a waitflag is grounded, the code in Listings 3.3 and 3.4 is called. It sets up the inner constraints of the quantifier using fresh waitflags that will later be grounded as

Listing 3.1: Core of Interpreter

```
solve_constraint(Constraint, TopLevelVars) :-
    retractall(enum_warning),
    solve(Constraint, ExistsWF, AllWF),
    ground_vars(TopLevelVars, ExistsWF, AllWF).

ground_vars(TopLevelVars, ExistsWF, AllWF) :-
    maplist(enumerate_exists_aux, TopLevelVars),
    ground_waitflags(ExistsWF, AllWF).

solve(A & B,EWF,AWF) :-
    solve(A,EWF,AWF), solve(B,EWF,AWF).
solve(A or B,EWF,AWF) :-
    solve(A,EWF,AWF) ; solve(B,EWF,AWF).
...
solve(not(A),EWF,AWF) :- solve_not(A,EWF,AWF).
solve(V in D,_,_) :- V in D.
solve(A = B,_,_) :-
    compute_exprs(A,B,AE,BE),
    AE #= BE.
...
solve(forall(X,LHS => RHS),_EWF,AWF) :-
    when(ground(AWF), enumerate_forall(X,LHS,RHS)).
solve(exists(X,RHS),EWF,_AWF) :-
    when(ground(EWF), enumerate_exists(X,RHS)).

solve_not(A & B,EWF,AWF) :-
    solve_not(A,EWF,AWF) ; solve_not(B,EWF,AWF).
solve_not(A or B,EWF,AWF) :-
    solve_not(A,EWF,AWF), solve_not(B,EWF,AWF).
...
```

well. This accounts for a hierarchy of scopes, where each may need to find a single solution or inspect all possible solutions for a variable. Again, see Fig. 3.1 for an example.

Enumeration

Enumeration occurs in two steps. First, we ground the waitflag triggering enumeration of existentially quantified variables. This leads to the coroutines set up in Listing 3.2 being resumed. The key difference between enumerating an existentially quantified variable (Listing 3.3) and a universally quantified one (Listing 3.4) lies within the solver's reaction to infinite domains.

Listing 3.2: Setup of Quantifiers

```
solve(forall(X,LHS => RHS),_EWF,AWF) :-
    when(ground(AWF),enumerate_forall(X,LHS,RHS)).
solve(exists(X,RHS),EWF,_AWF) :-
    when(ground(EWF),enumerate_exists(X,RHS)).
```

Listing 3.3: Enumerate Existentially Quantified Variable

```
enumerate_exists(Var,RHS) :-
    % setup inner constraints
    solve(RHS,NewEWF,NewAWF), !,
    ground_waitflags(NewEWF,NewAWF),
    enumerate_exists_aux(Var).
enumerate_exists_aux(Var) :-
    fd_size(Var,sup), !,
    % non-exhaustively enumerate infinite domain
    % need to find just one element!
    assert(enum_warning),
    fd_inf(Var,Min), fd_sup(Var,Max),
    enumerate_infinite(Var,0,Min,Max).
enumerate_exists_aux(Var) :-
    indomain(Var).
```

Existentially quantified ones are enumerated as shown in Listing 3.3:

1. The inner constraint of the quantifier is set up as usual,
2. Inner variables are enumerated,
3. We begin enumerating the quantified variable:
    - If the domain is infinite, we store that enumeration cannot be exhaustive. Afterwards, we start enumerating.
    - Otherwise, we use regular CLP(FD) labeling.

In the second step, we cannot enumerate all possible values of the variable in question. We thus have to decide on a value selection strategy. Our simple example interpreter enumerates as follows: Starting from zero we alternate between the positive and negative value of an increasing counter, skipping values not in the corresponding domains. As soon as both upper and lower bounds are passed, the domain has been enumerated exhaustively.

This simple enumeration pattern is not sophisticated enough for complicated constraints. To improve, one could rely on techniques like the level diagonalization suggested in [46].

In ProB, we combine the simple enumerator with other (non-exclusive) strategies like case splits. Another alternative is to use random enumeration as we will show in Section 3.3.2. After all existentially quantified variables have been enumerated, we ground the waitflag that triggers universal quantifiers.

We enumerate as outlined in Listing 3.4:

1. The inner constraint of the quantifier is set up as usual,
2. We begin enumerating the variable:
   - If the domain is infinite, we throw an enumeration exception. We would need to fully enumerate an infinite domain in order to solve the constraint. This futile attempt is discarded. However, the occurrence of infinite domains is usually influenced by the order of variables. By enumerating variables with small finite domains first, we might shrink other domains.
   - Otherwise, we try all values in its domain to check the universal quantification.

With this extension, our CLP(FD)-based solver is able to handle both existential and universal quantification. In several cases, for instance in $y = 2 \wedge \forall\, x.(x \in [0, 10] \Rightarrow x > y)$ the solver can recognize exhaustiveness of labeling. As a result, satisfiability and unsatisfiability can be deduced and reported to the user.

In case of infinite or large domains the solver can tell if a result is still valid, despite the fact that a domain has not been enumerated completely. To increase coverage of large domains in the face of timeouts, the following section will introduce random enumeration.

3.3.2 Randomized Enumeration of Large Intervals

Another limitation of most CLP(FD) systems lies in how the next value of a variable is selected upon labeling. Within SICStus, one can select between the following strategies [41] together with the options *up*, to use ascending order, and *down*, to use descending order:

- *step*, i. e., a binary choice between $X = B$ and $X \neq B$, where B is the lower or upper bound of X.
- *enum*, i. e., multiple choice for X corresponding to the values in its domain.
- *bisect*, i. e., binary choice between $X \leq M$ and $X > M$, where $M = \lfloor \frac{\min(X)+\max(X)}{2} \rfloor$.
- *median*, i. e., binary choice between $X = M$ and $X \neq M$, where M is the median of the domain.

Listing 3.4: Enumerate Universally Quantified Variable

```
enumerate_forall(Var,LHS,RHS) :-
    LHS = (_ in Min .. Max),
    % setup of inner constraints: contains a choicepoint
    % to allow for different solutions to inner variables
    solve(LHS & RHS,NewEWF,NewAWF), !,
    ground_waitflags(NewEWF,NewAWF),
    enumerate_forall_aux(Min,Max,Var).
% exhaustively enumerate infinite domain? -> exception
enumerate_forall_aux(_,sup,_) :- throw(enum_infinite).
enumerate_forall_aux(inf,_,_) :- throw(enum_infinite).
% domain is finite, try all elements
enumerate_forall_aux(Current,Max,Var) :-
    Current =< Max, !,
    try_forall_value(Current,Var), % does not bind Var
    Current2 is Current + 1,
    enumerate_forall_aux(Current2,Max,Var).
enumerate_forall_aux(_,_,_).
```

- *middle*, i. e., binary choice between $X = M$ and $X \neq M$, where $M = \lfloor \frac{\min(X)+\max(X)}{2} \rfloor$.

Summarizing, CLP(FD) variables can be set to domain values in various ways using domain splitting or simple enumeration. One property is common to all strategies. The domains are traversed deterministically. In case the domains are small enough, i. e., they can be enumerated exhaustively, there is no need for a different strategy.

However, for large domains this might not be sufficient. Instead of traversing the domain linearly, it could be beneficial to use a random permutation:

- For large domains, values of different sizes will be tried out before a timeout occurs.
- It is less likely to get stuck in some part of the search space where there is no solution. If we fear search is stuck, we could restart as described in [86]. This is common in SAT and SMT solvers.
- For applications like test case generation it is desirable to compute test inputs that fulfill some coverage criterion, e. g., that certain intervals or sets of parameters or values have been used. With linear enumeration and backtracking, generated test cases might only differ in the variable set last.

Note that we want to compute a random permutation of the domain. To avoid duplicates we do not want to randomly draw elements from the domain.

Algorithm 3.1: Fisher-Yates / Knuth shuffle

```
Data: List a
Result: Random permutation of a
for i ∈ [0, length(a) − 1] do
    chose j randomly such that 0 ≤ j ≤ i
    if j ≠ i then
        perm[i] := perm[j]
    end if
    perm[j] := a[i]
end for
return perm
```

Furthermore, we have to keep track of the exhaustiveness of our enumeration in order to detect unsatisfiability.

A classic algorithm to compute random permutations for given intervals is the Fisher-Yates shuffle [78] or Knuth shuffle [111]. Its pseudo code can be found in Algorithm 3.1. The algorithm has a weakness: the list to be shuffled has to be in memory completely. This is not feasible for intervals too large to be stored. We hence need an algorithm allowing us to compute a random permutation on the fly.

One such algorithm can be constructed using cryptographic techniques as outlined in [134]. The key idea is to construct an encryption function encrypting the elements to be permuted onto themselves. In order to allow for later decryption, an encryption function has to be unambiguous, i. e., given a fixed key there has to be a one-to-one mapping between plaintext and ciphertext. This will ensure that we do in fact compute a permutation, i. e., we do not add or remove elements. Before we can present our implementation, we introduce a few definitions regarding ciphers. The definitions are following [145].

Definition 3.3.1 (Block Cipher)**.** An n-bit *block cipher* is a function $E : [0,1]^n \times K \to [0,1]^n$, such that for each key $K \in \mathcal{K}$, $E(P,K)$ is an invertible mapping from $[0,1]^n$ to $[0,1]^n$, written $E_K(P)$. $E_K(P)$ is called the *encryption function* for K. The inverse mapping is the *decryption function*, denoted $D_K(C)$. $C = E_K(P)$ denotes that ciphertext C results from encrypting plaintext P under K.

Definition 3.3.2 (Random Cipher)**.** A (true) *random cipher* is an n-bit block cipher implementing all $2^n!$ bijections on 2^n elements. Each of the $2^n!$ keys specifies one such permutation. Obviously, the key space for a true random cipher is too large to be used in practice.

Definition 3.3.3 (Iterated Block Cipher)**.** An *iterated block cipher* is a block cipher involving the repeated application of an internal function called a *round function*. In each iteration, the current input is split and encrypted by applying

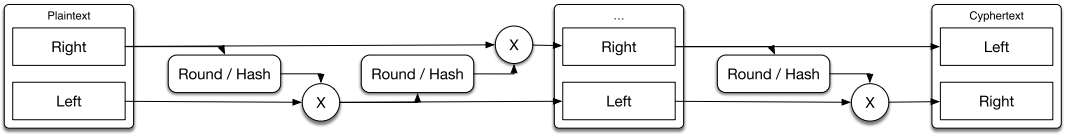


Figure 3.2: Feistel Network

the round function to some part of the input. Afterwards, the different parts are recombined such as to ensure that each part will pass through the round function eventually. Parameters include the number of rounds r, the block bit size n, and the bit size k of the input key K from which r subkeys K_i, one for each round, are derived. For invertibility, for each K_i the round function is a bijection on the round input.

Definition 3.3.4 (Feistel Cipher)**.** A *Feistel cipher* is an iterated block cipher mapping a $2n$-bit plaintext (L_0, R_0), for n-bit blocks L_0 and R_0, to a ciphertext (R_r, L_r), through an r-round process where $r \geq 1$. For $1 \leq i \leq r$, round i maps (L_{i-1}, R_{i-1}) to (L_i, R_i) as follows:

$$L_i = R_{i-1}, R_i = L_{i-1} \oplus f(R_{i-1}, K_i).$$

For each round, the *subkey* K_i is derived from the given cipher key K. f is the *round function* mentioned in Definition 3.3.3. Often other (product) ciphers are used as f. However, f does not need to be invertible for the Feistel cipher to be. Figure 3.2 shows how a Feistel cipher operates on the input blocks.

Now, we can follow one of the approaches suggested by [27] to construct a cipher that does not operate on $[0,1]^n$ but rather on $[0,k]$. The authors describe the construction of a generalized Feistel cipher:

1. First, in order to encrypt an element m taken from $[0,k]$, we have to split it in a left part L and a right part R according to Definition 3.3.4. This is done by arbitrarily choosing $a, b \in \mathbb{N}$ with $ab \leq k$ and decomposing m into $L = m \bmod a$ and $R = \lfloor \frac{m}{a} \rfloor$.
2. Use (L, R) as inputs to a Feistel cipher as given in Definition 3.3.4. This includes performing a given number of iterations using random round functions whose ranges contain $[0,k]$.
3. Depending on the round functions and their ranges, the resulting ciphertext might not be in $[0,k]$. If it is, we return it. Otherwise, the cipher is iterated until an element in $[0,k]$ is returned.

Observe that Definition 3.3.4 assumes that L and R have the same length. Hence, the overall bit-length has to be an even number[2]. With the construction above, we

[2]For technical reasons the actual implementation uses a bit-length divisible by 4.

Algorithm 3.2: Random Permutation: Setup

Data: Interval I to draw from
Result: Bitmasks BL to extract L, BR to extract R, number of bits n

$length = \max(I) - \min(I) + 1$
$n = \ln(length)$
if *n is odd* **then**
    $n = n + 1$
end if
$BR = 2^{\frac{n}{2}-1}$
$BL = 2^n - 1 - BR$

can thus only create ciphers for certain intervals, i. e., we can construct a cipher $[0, 15] \to [0, 15]$ but not $[0, 5] \to [0, 5]$. However, the cipher $[0, 15] \to [0, 15]$ can be used to permute $[0, 5]$ by iterating to the next index if a number ≤ 6 is drawn. Intervals that do not start with 0 are shifted. The shift is later re-added to the drawn number.

The implementation of random permutations is outlined in Algorithm 3.2 and Algorithm 3.3. We can use a simplified version of the construction in [27] because we do not rely on strong cryptographic properties.

On the interval $[1, 6]$ it proceeds as follows:

1. Compute the interval length $l = 6 - 1 + 1$.
2. Find the number of bits needed to store l. In this case $l = 6$ means we need at least 3 bits.
3. Round up to the next even number to allow symmetric split into L and R. We can split an interval of 4 bits. Hence, we need to take into account all inputs from $[0, 15]$.
4. Compute the bit masks $BL = 1100$ and $BR = 0011$ used to extract the first 2 and the last 2 bits.
5. Set the current index to 0 and select a random key to choose a permutation.
6. To draw the next element of the random permutation:
    a) Use a Feistel cipher to encrypt the current index using the key.
    b) Increment the index.
    c) Repeat if the cipher value is larger than 5, else return $(5 + 1)$ to stay within range.

Algorithm 3.3: Random Permutation: Next Element

Data: Interval I, current index c, maximum index $maxIdx$, left/right mask BL, BR, number of bits n
Result: Next random element rnd and next index to draw $next$

do
  $left = c \;\&\; BL >> \lfloor \frac{n}{2} \rfloor$
  $left = c \;\&\; BR$
  $left, right = feistel\_rounds(left, right)$
  $rnd = (left << \lfloor \frac{n}{2} \rfloor) | right$
  $c = c + 1$
while $rnd > \max(I) - \min(I) \wedge c < \max$
if $c > maxIdx$ **then**
  return failure, permutation enumerated exhaustively
else
  return rnd, $next = c + 1$
end if

For the rounding function one can use a hash function such as Prolog's term_hash. We use a given random seed as the encryption key. Given that we do not rely on any cryptographic properties, we reuse the main key for all subkeys to facilitate the implementation.

3.3.3 High-Level Reasoning using CHR

So far ProB was mostly used for animation, model checking and data validation [126]. Hence, it used to be tailored towards finding solutions to satisfiable formulas, which is where constraint programming has its strengths.

Therefore, choosing CLP(FD) as a basis for ProB has been a reasonable decision:

- It can deal with large and only partially known data.
- Even though B is a higher-order language including sets, relations and functions, everything could be expressed by or in conjunction with CLP(FD) variables and constraints.
- Support of reification made it considerably easier to propagate information between the different solvers, i. e., from integer constraints to set constraints and vice versa.
- As predicates might include arithmetic and higher-order functions, satisfiability of B formulas is in general undecidable. Yet, ProB is able to solve a useful class of subproblems.

New applications like symbolic model checking however made shortcomings of CLP(FD) obvious: Even for simple constraints like $x < y \wedge y < x$ contradictions often can only be detected if variables have finite domains. Again, this is due to propagation relying on finite upper and lower bounds [179, 9].

To improve, we implemented a set of rules working on top of the CLP(FD) variables. Our initial idea was to perform some kind of high-level propagation, where new CLP(FD) constraints are discovered from the existing constraints. That is, we would implement propagation rules like the transitivity of $<$ stating that $x < y \wedge y < z \Rightarrow x < z$. These rules are implemented in CHR [81, 82], a committed choice language that can be embedded in Prolog.

CHR supports three different kinds of rules to modify a *constraint store* holding the current state of constraints:

- Simplification rules of the form $h_1, \ldots, h_n \mid g_1, \ldots, g_m \iff b_1, \ldots, b_o$. Here, $h_1, \ldots, h_n$ are the so called "head", i. e., constraints that have to be found in the constraint store for the rule to act upon. $g_1, \ldots, g_m$ are called "guards". These are predicates that have to hold for the rule to be allowed to fire. If the rule can be executed, the heads are rewritten into the "bodies" $b_1, \ldots, b_o$, i. e., $h_1, \ldots, h_n$ are removed from the constraint store while $b_1, \ldots, b_o$ are added.
- Propagation rules of the form $h_1, \ldots, h_n \mid g_1, \ldots, g_m \implies b_1, \ldots, b_o$. Here, the bodies are added to the constraint store without removing the heads.
- Simpagation rules like $h_1, \ldots, h_l \setminus h_{l+1}, \ldots, h_n \mid g_1, \ldots, g_m \iff b_1, \ldots, b_o$ combine the former. The constraints $h_1, \ldots, h_l$ are kept in the constraint store while $h_{l+1}, \ldots, h_n$ are removed.

We augmented the constraint solver with CHR rules handling integer arithmetic, focusing on detection of contradictions involving linear inequalities. The rules are comparable to those introduced in the finite domain solver of [82, Ch. 8]. However, we do not handle domains inside CHR, but rather integrate with CLP(FD). Regarding the implementation of infinite domain solvers in CHR see [82, Ch. 9].

Listing 3.5 includes an extract of the CHR rules encoding properties of $<$ and $\leq$. The complete source code is given in Listing B.3 As can be seen, we introduced rules for (anti-)reflexivity, antisymmetry, idempotence and transitivity. For instance, transitivity of $\leq$ is given by the CHR rule `leq(X,Y), leq(Y,Z) ==> leq(X,Z)`, stating that from $X \leq Y \wedge Y \leq Z$ we can infer $X \leq Z$. Newly inferred constraints are submitted to the underlying CLP(FD) system.

In addition to inferring new constraints, CHR rules can be used to infer unsatisfiability. As an example, the rule encoding the antireflexivity of $<$ states that if we have $X < X$, `fail` has to be executed. Due to CHR being a committed-choice

Listing 3.5: CHR Rules for Integer Inequalities

```
reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).

antireflexivity @ lt(X,X) <=> fail.
idempotence     @ lt(X,Y) \ lt(X,Y) <=> true.
transitivity    @ lt(X,Y), leq(Y,Z) ==> lt(X,Z).
transitivity    @ leq(X,Y), lt(Y,Z) ==> lt(X,Z).
transitivity    @ lt(X,Y), lt(Y,Z)  ==> lt(X,Z).
```

Table 3.1: Small Benchmarks with / without CHR (in s)

predicate	without CHR	with CHR
$x > 3$	1.31	1.27
$x > y \wedge y > x$	timeout	0.92
$x = 3 \wedge x > y \wedge y = 4$	0.96	0.95
$x = 3 \wedge x > y$	1.24	1.04
$x = 3 \wedge x < y$	0.93	1.0
$w > x \wedge x > y \wedge y > z \wedge w = 1 \wedge z = 1$	0.94	1.07
$w > x \wedge x > y \wedge y > z \wedge z > w$	timeout	0.93
$x + 2 > y + 1 \wedge y > x$	timeout	timeout
$x > y \wedge y > x + 1$	timeout	0.98

language, the whole CHR run is discarded. The Prolog rule that added the offending constraint by calling `lt(X,X)` fails. In consequence, further propagation is stopped and the failure has to be handled by the solver.

Of course a fully fledged solver needs to include several other CHR rules dealing with common cases of integer constraints. As can be seen in the examples in Table 3.1, the example set of CHR rules is far from being complete: it is able to handle simple cases like $x > y \wedge y > x$ and transitive cases like $w > x \wedge x > y \wedge y > z \wedge z > w$. However, so far it is unable to do simple arithmetic as in $x + 2 > y + 1 \wedge y > x$. In ProB we have added both arithmetic as well as further high-level rules, e. g., for set membership and (strict) subsets.

3.4 Applications

Besides symbolic model checking, we have several applications for an infinite domain constraint solver within ProB. We already mentioned animation, model

checking and constraint-based validation in the introduction. In this section, we will give a preview of other applications.

As mentioned above, one of our key motivations is to move ProB from being guaranteed sound for finite domains to infinite domains. This is particularly important if ProB is to be used as a prover.

In [132, 113], we embedded ProB into Rodin [3], an IDE for Event-B, in order to generate counterexamples for proof obligations. Given a sequent with goal $G(x_1, \ldots, x_k)$ and hypotheses $H_i(x_1, \ldots, x_k)$ we build the predicate

$$\exists\, x_1, \ldots, x_k : H_1(x_1, \ldots, x_k) \wedge \ldots \wedge H_n(x_1, \ldots, x_k) \wedge \neg G(x_1, \ldots, x_k)$$

and feed it to our constraint solver. If the predicate does hold, ProB returns a valuation for $x_1, \ldots, x_k$, representing a counterexample to the sequent.

In general, checking the satisfiability of propositional formulas is NP complete. Beyond that, typical Event-B proof obligations consist of first-order logic formulas, for which the problem becomes undecidable. Previously [128, 127, 132], we overcame this limitation by limiting domains to be finite. However, this prevents drawing any conclusions from the absence of a counterexample.

The techniques presented in this work have been used to extend the disprover to a fully fledged prover. By observing the state of enumerations as explained in Section 3.3.1, ProB is able to tell if the search for a counterexample was exhaustive. If this is the case, we can report a proof to the user.

We performed several benchmarks comparing our prover to ML and PP [56], two specialized provers for B and Event-B. Additionally, we compared the performance of SMT solvers on the proof obligations. Details will be given in Chapter 4.

The techniques presented can also be used to solve SMT problems, such as the ones collected by the SMT-LIB project. In particular, we used all benchmarks that involve (non-)linear integer arithmetic and quantification but no other constructs like arrays or bit vectors. The translation from SMT-LIB to B will be discussed in Chapter 5. An empirical evaluation will be performed as well. To summarize, ProB augmented with enumeration tracking cannot compete with Z3 [60]. Yet, a considerable number of both satisfiable and unsatisfiable benchmarks can be solved using our technique.

3.5 Related Work

SMT solvers [19] such as Z3 [60] are able to handle infinite domains and quantifiers. As outlined in the introduction, a major difference lies within the handling of

data. While SMT solvers are more focused on predicates, CLP(FD) systems are more oriented towards possible valuations of variables. As a result, SMT solvers can handle infinite domains more efficiently and detect unsatisfiability in more cases. However, model generation (in particular in the presence of quantifiers) is often easier using CLP(FD).

In Chapter 6, we will investigate a different approach to add high-level reasoning to a CLP(FD)-based solver. Instead of implementing rules in CHR, we connect the SMT solver Z3 to SICStus Prolog. We transfer each predicate asserted in CLP(FD) to Z3 and queried both solvers for a solution. Furthermore, intermediate assignments and domains can be communicated back and forth.

While the approach is more powerful when it comes to reasoning, using CHR rules has the advantage of an immediate integration into Prolog. Hence, there is no communication overhead and no translation between different representations is needed.

3.6 Conclusion and Future Work

Summarizing, we have presented an approach to lift a CLP(FD)-based solver to infinite domains. Our approach tracks enumerations occurring during search and interprets how they affect the overall result. Two extensions, high-level reasoning and random enumeration were used to increase applicability.

Techniques have been added to the animator and model checker ProB. In addition, they allowed us to use ProB as a prover and SMT solver. Both applications will be used to assert the performance of our extended solver and to verify if further additions are needed in order for ProB to be suitable as the backend of a symbolic model checker.

You can't prove any hypothesis, you can only improve or disprove it.

Christopher Monckton

4

Application: The ProB Disprover

In this chapter we will use proof obligations taken from B and Event-B models to benchmark the improvements introduced in the last chapter. The models include examples from both academia as well as industry and should allow us to assert ProB's performance as solver and prover.

The chapter is based on our paper "From Failure to Proof: The ProB Disprover for B and Event-B" [113]. For information regarding authors and their individual contributions see Appendix C.

4.1 Introduction and Motivation

As stated in Sections 2.1.1 and 2.1.2 both B [1] and its successor Event-B [2] are used for the formal development of software and systems that are correct by construction. This usually involves formal proofs of different properties of the specification.

Many provers, such as "ml" and "pp" of Atelier B [55], are able to discharge certain proof obligations automatically. In former work [132] Ligot, et al. described a disprover based on using ProB's constraint solver to automatically find counterexamples for given proof obligations and thus saving the user from spending time in a futile interactive proof attempt. Say that we have to prove that the goal G is a logical consequence of the hypotheses $H_1, \ldots, H_n$. The ProB disprover then tries to find a solution for the formula $H_1 \wedge \ldots \wedge H_n \wedge \neg G$. If it can find a solution, the proof cannot succeed and the solution is a counterexample.

In [132] Ligot, et al. already made the observation that in some cases, namely if we encounter neither infinite sets nor deferred sets whose cardinality is unbounded, the absence of a counterexample is actually a proof. The authors thus suggested as future work to implement an analysis that checks if the absence of a counterexample is a valid proof.

This can be achieved using the techniques presented in Chapter 3: ProB can now keep track of infinite enumeration, in particular the scope in which an infinite enumeration has occurred and whether a solution has been found or not. This enables the disprover to detect if the search for a counterexample was exhaustive, i. e., we can now use ProB as a prover. Note that we go beyond the suggested future work of [132]: we allow variables with an infinite domain to occur, as long as they do not have to be enumerated exhaustively. We have also improved the core algorithm of [132] in various ways, by allowing to focus on selected hypotheses and by providing a way to detect inconsistencies in the hypotheses or potential bugs in the disprover.

This chapter will describe how ProB can be used as a prover in more detail in Section 4.2. Inconsistency detection will be presented in Section 4.2.3. A thorough empirical evaluation, comparing our constraint-based proof with existing provers for B and Event-B is presented in Section 4.3 The study shows that the constraint-based proof fares surprisingly well for a variety of case studies.

4.2 Constraint-Based Proof Technique

In the following section we describe how ProB can be used as a prover inside Rodin [3] and Atelier B [55]. First, in Section 4.2.1, we provide a few examples showing how the techniques introduced in Section 3.3 influence ProB's constraint solver when it is applied to typical proof obligations. Further technical details regarding ProB's kernel can be found in [128, 127] or [126]. Following, Section 4.2.2 will outline how ProB was embedded into Rodin's proof architecture. Afterwards, in Section 4.2.3 we will show how ProB can be used to detect inconsistencies in the model.

4.2.1 ProB's Constraint Solving Kernel

Using the techniques introduced in Section 3.3, ProB's constraint solver tracks where and why enumeration occurs. It is able to distinguish between safe and unsafe enumerations, i. e., if all possible values of a variable have to be tried out or if a single solution is sufficient. Exhaustive enumeration can then be detected individually for each variable and later be transferred to the whole constraint if possible.

Let us look at a few examples showing how ProB can be used for proof:

- $i \in \{1, 2, 1024, 2048\} \wedge i > 2 \stackrel{?}{\Rightarrow} i \bmod 2 = 1$
  Here, we have the two hypotheses $i \in \{1, 2, 1024, 2048\}$ and $i > 2$ and we want to prove that $i \bmod 2 = 1$ is a logical consequence. Hence, we would construct the formula $i \in \{1, 2, 1024, 2048\} \wedge i > 2 \wedge \neg(i \bmod 2 = 1)$ and try to find solutions for i. For this formula, ProB finds two solutions ($i = 1024$ and $i = 2048$) and no infinite enumeration has occurred, as ProB has narrowed down the interval of i to 3..2048 before enumeration has started. As such, we can conclude that $G \equiv i \bmod 2 = 1$ is *not* a logical consequence of the hypotheses $H_1 \equiv i \in \{1, 2, 1024, 2048\}$ and $H_2 \equiv i > 2$. The same solutions could be found by a simple CLP(FD) query.

- $i \in \{1, 2, 1024, 2048\} \wedge i > 2 \stackrel{?}{\Rightarrow} i \bmod 2 = 0$
  For the opposite of the goal, i. e, $i \bmod 2 \neq 0$, we construct the formula $i \in \{1, 2, 1024, 2048\} \wedge i > 2 \wedge \neg(i \bmod 2 = 0)$. In this case ProB finds no solution and no infinite enumeration has occurred. As such, we have proven that $i \bmod 2 = 0$ follows logically from $i \in \{1, 2, 1024, 2048\} \wedge i > 2$. A simple CLP(FD) query also confirms that there is no solution.

- $i > 20 \stackrel{?}{\Rightarrow} i \bmod 2 = 1$
  If we want to prove that $(i \bmod 2 = 1)$ is a logical consequence of $i > 20$, we construct the formula $i > 20 \wedge \neg(i \bmod 2 = 1)$. ProB finds a solution ($i = 22$), but infinite enumeration has occurred in the sense that the possible values of i lie in the interval 22..∞. However, in this context this is not an issue, as a solution has been found. As such, we can conclude that $i \bmod 2 = 1$ is not a logical consequence of $i > 20$. This time there is no CLP(FD) query that returns a solution. As there is no finite domain attached to i, labeling cannot be performed. Thanks to enumeration tracking, ProB is able to (partially) enumerate the infinite domain of i in order to find a solution.

- $i > 20 \stackrel{?}{\Rightarrow} (i \bmod 2 = 0 \vee i \bmod 1001 \neq 800)$
  Finally, if we want to prove that $(i \bmod 2 = 0 \vee i \bmod 1001 \neq 800)$ is a logical consequence of $i > 20$, we get the formula $i > 20 \wedge \neg(i \bmod 2 = 0 \vee i \bmod 1001 \neq 800)$. Here ProB finds no solution, but an "enumeration warning" is produced. Indeed, the constraint solver has narrowed down the possible solutions for i to the interval 801..∞, but with the default search settings no solution has been found. Here, we cannot conclude that $i \bmod 2 = 0 \vee i \bmod 1001 \neq 800$ is a logical consequence of $i > 20$. Indeed, $i = 1801$ is a counterexample, which ProB can find if we enlarge the default search space, e. g., by adding $i < 10000$ as additional constraint. Again, CLP(FD) is unable to solve the query due to the infinite domain of i.

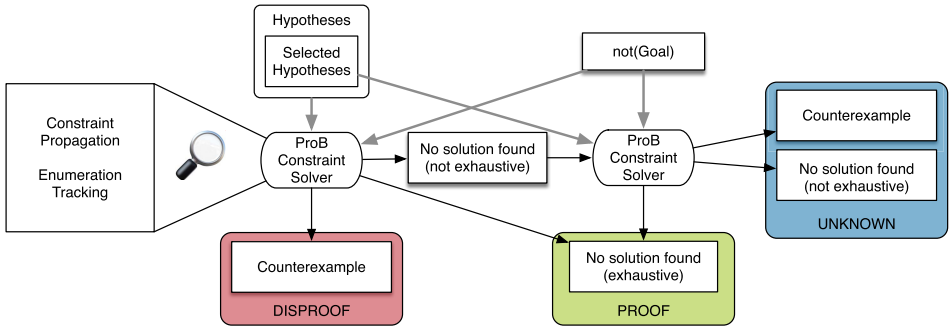


Figure 4.1: Disproving Algorithm

Figure 4.2: Counterexample Inside Rodin Proof Tree

4.2.2 Integration into Rodin for Event-B

When working on a proof obligation, Rodin keeps track of two sets of hypotheses: the set of *all* available hypothesis for the target goal and a *user-selected* subset. The idea is to be able to reduce the search space of the automatic provers by excluding irrelevant hypotheses. In the case of the ProB prover we could, for instance, get rid of hypotheses that are irrelevant for the proof but contain variables over infinite domains, deferred sets or complicated constraints.

This approach cannot lead to false positives, because limiting the number of available hypothesis cannot render a formerly unprovable sequent provable. However, disproving while omitting hypotheses can lead to false negatives if the hypotheses are too weak for a proof. For instance, say the goal G is $i \bmod 2 = 1$ and the hypotheses are $i \in \{1, 2, 3\}$ (H_1) and $i \neq 2$ (H_2). ProB will not find a counterexample for $H_1 \wedge H_2 \wedge \neg G$ but it will find a (false) counterexample for $H_1 \wedge \neg G$.

In [132], the authors used to generate specially crafted B machines holding the hypotheses and the goal into ProB. Newer versions of ProB allow sending constraints directly to the solver without loading a machine. Hence, we can now avoid the overhead of encoding.

Figure 4.1 outlines how the disprover proceeds in more detail:

1. We first try to solve the predicate $H_1 \wedge \ldots \wedge H_m \wedge \neg G$, i. e., the negated goal together with *all* available hypotheses. If we find a solution, we report the proof obligation as *unprovable* by inserting the counterexample into the Rodin proof tree as shown in Fig. 4.2. If no counterexample is found and search was exhaustive, the initial sequent is *proven*, because no counterexample *exists*.
2. If the constraint solver is unable to prove or disprove the predicate in step 1, we reduce the number of hypotheses to the user-selected hypotheses and again look for a counterexample. The three possible outcomes are:
   - No counterexample was found using the reduced set of hypotheses. This is still a valid *proof*, as removing hypotheses can only introduce further counterexamples but not remove them.
   - If we find a solution, we report a *possible* counterexample, but leave the proof obligation status as *unknown*. However, we do not interfere with the ongoing proof effort, as the proof obligation might still be provable using all hypotheses.
   - Otherwise we return without a result (status is *unknown*).

4.2.3 Inconsistency Detection

After the algorithms outlined in Section 4.2.2 return a proof, a second phase can be triggered as outlined in Fig. 4.3: We try to find a proof for the negation of the goal. This time, we send $H_1 \wedge \ldots \wedge H_m \wedge G$ to the constraint solver. The result allows us to decide whether the goal predicate G played a role in the original proof.

If the negated goal can be proven as well, we detected a contradiction in the hypotheses. Contradicting hypotheses might occur due to an error in the model, in particular if they are detected at the root of the proof tree. Deeper within a proof, contradicting hypotheses can occur "naturally",e. g., by case distinctions or proof-by-contradiction. Hence, the user should be notified if they occur in a successful proof.

If contradicting hypotheses or disproven obligations have been found, ProB can afterwards compute the unsat core in order to provide smaller counterexamples and ease understanding of shortcomings in the underlying model.

In our of our benchmark models, this unsat core helped us to identify and fix several bugs caused by various contradictions in the theorems at lower refinement levels. It also highlighted an issue in the first development of the ABZ landing gear from [175]. The ProB disprover was flagging, e. g., the proof

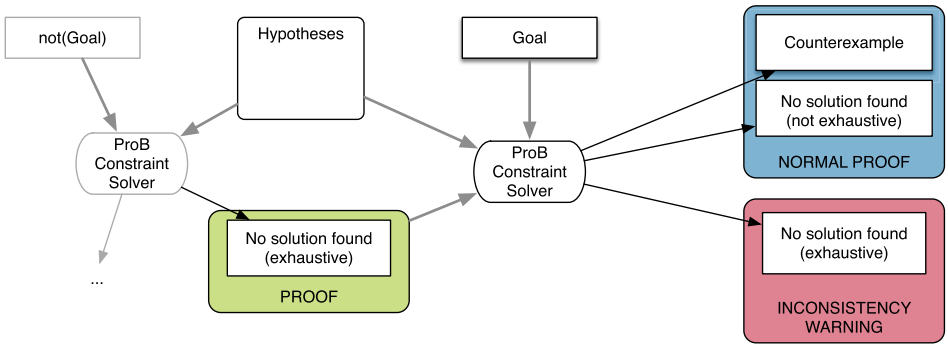


Figure 4.3: Inconsistency Detection

obligation `treat_hndl_up_112/inv1/INV` in the machine LPN4 as containing a contradiction in the hypothesis. The ProB unsat core algorithm found out the following root cause:

```
  close_EV = FALSE & open_EV = FALSE & door = op2cl &
((open_EV = FALSE & close_EV = FALSE) => door = cl) &
  partition(D, \{cl\}, \{cl2op\}, \{op2cl\}, \{op\})
```

The first line comes from the guard of `treat_hndl_up`, the second line is the invariant `inv1` from LPN4. The last line contains an axiom found in the seen context `LPNC0`. As the axiom is used for typing purposes inside ProB, it is never removed from the unsat core by the current algorithm.

Summarizing, the disprover has detected that the event `treat_hndl_up` can never be executed given the invariant. A similar issue was detected for other events, such as `treat_hndl_up_122`, `treat_hndl_up_132`, `treat_hndl_dn_112`, and `treat_hndl_dn_132` in machine `LPN4` of [175].

Furthermore, this two-phase analysis can be used to detect bugs in ProB: if the search for a counterexample fails to explore certain cases, it might be independent of the goal. Hence, we can detect if ProB correctly spots contradictions using crafted sequents. We could even go further and apply other provers to the unsat core generated by ProB in order to validate a proof effort by a second toolchain.

4.3 Empirical Evaluation

In this section, we compare ProB to several other provers available for the Rodin platform [3], i. e., Rodin's automatic tactic and the SMT plug-in [63, 64].

Our evaluation leads us to the following conclusions:

- In many cases ProB can discharge proof obligations that cannot be discharged by other provers. Each additional obligation that is discharged actually saves time and money.
- None of the provers can be replaced by the others.
- The performance of a prover is influenced by the surrounding tactic, including other provers. While the influence of a tactic on ProB is only marginal, it is quite strong for other provers.

4.3.1 Experimental Setup

For our experiments, we have used Rodin 3.2, version 2.1.0 of the Atelier B provers plugin and version 1.3.0 of the SMT plugin. Within the SMT plugin, we used the bundled versions 2.4.1 of CVC3, 1.4 of CVC4, Z3 version 4.4.1 and the bundled development version of veriT. We have used a timeout of 5 seconds for each SMT solver, run in succession. ProB was used in version 1.6.2-beta1, connected through the disprover plugin version 3.0.9.

Again, a timeout of 5 second was used for each constraint solving attempt with a maximum of two attempts per proof obligation (see Fig. 4.1). Both the CLP(FD) and the CHR-based solvers of ProB were activated. We used a global timeout of 25 seconds for a whole tactic.

All benchmarks were run on a MacBook Pro featuring a 2.6 GHz Intel Core i7 CPU and 8 GB 1600 MHz DDR3 memory. We did not run proof attempts in parallel to avoid issues due to hyper-threading or scheduling. We developed an evaluation plugin[1] for the Rodin platform that applies the user- or pre-defined proof tactics to selected proof obligations.

We used the following combined tactics as they represent closely what can be utilized by end-users:

- The automatic tactic that comes with Rodin. It applies a number of rewriting rules and decision procedures to the proof tree. For instance, it checks if the goal is included in the set of hypotheses and thus discharged. The automatic tactic is applied until a fixpoint is reached or the process times out. This is the "Default Auto Tactic" of Rodin where the calls to PP and ML have been removed. Figure 4.4a shows the tactic definition.
- In a second step, we used this tactic in its original state, i. e., with the PP and ML provers from Atelier B enabled.

[1] See `https://github.com/wysiib/ProverEvaluationPlugin` for sources and instructions.

- The SMT plugin [63, 64] applies two different SMT solvers (veriT [33] and CVC3 [20]) to the original goal. We used the default SMT tactic as shown in Fig. 4.4b. Note that it calls PP and ML as well.
- Finally, we added ProB to the tactic as well. It is applied to the goal before the other provers. The corresponding tactic definition is shown in Fig. 4.4c.

In addition, we benchmarked the provers alone, i. e., without tactics. This gives us a better picture of the individual power of each prover.

- PP and ML from Atelier B together,
- SMT plugin on its own, using all bundled solvers, and
- ProB alone.

We used the following models for our benchmarks:

- Answers to the ABZ-2014 landing gear case study [31]. Beside our own version [92], we also used the three models by Su and Abrial [175], a model by André, Attiogbé and Lanoix [7], as well as a model by Mammar and Laleau [138].
- A model of the Stuttgart 21 Railway station interlocking by Wiegard, derived from chapter 17 of [2] with added timing and performance modeling.
- A model of a controller area network (CAN) bus developed by Colley. A CAN Bus is used in vehicles for direct communication between components without a central processor.
- A formal development of a graph coloring algorithm by Andriamiarina and Méry. The graphs to be colored are finite, but unbounded and not fixed in the model.
- A model of a pacemaker by Méry and Singh [146].

The models were selected so as to cover a variety of use cases. The landing gear model [92] contains mainly enumerated sets; hence we suspected ProB to perform well. We included several other versions of the case study to investigate how modeling style influenced prover performance. On the other end of the spectrum, the graph coloring model uses only deferred sets. Hence, we expected ProB not to perform well, as finite enumeration is not possible. The other models were expected to lie in between those extremes as far as ProB's performance is concerned. We do not claim that our selection is representative. Indeed, we could have selected more models using (mostly) deferred sets; but this would have just confirmed that ProB's prover is not useful for proof obligations involving deferred sets.

4.3.2 ProB Solver Settings

Several of ProB's preferences influence its performance as a prover. Below, we discuss which settings we used.

Detection of well-definedness problems: We disabled ProB's built-in detection of well-definedness errors like function application outside of the domain or division by zero. This speeds up the search for a counterexample. In Event-B, proof obligations are assumed to be well-defined. Well-definedness is verified by an independent obligation where necessary. In case a proof obligation is not well-defined, a corresponding well-definedness proof obligation generated by Rodin will thus be unprovable.

Extended ruleset for contradiction detection: Furthermore, we enabled the set of CHR rules discussed in Section 3.3.3. As stated, they should mitigate certain shortcomings of the CLP(FD) system by adding rules to detect unsatisfiability and inferring further constraints. However, they often extend the time the solver takes to complete and may thus increase the amount of occurring timeouts.

Aggressive treatment of choice points: We used ProB's so called SMT mode, in which it enumerates more aggressively. In addition to the variables enumerated by CLP(FD)-style labeling, this makes ProB increase the limits for enumeration of quantifiers or set comprehensions. For instance, ProB starts partially enumerating domain and range of functions as soon as a function application has to be executed by the interpreter. With the SMT mode disabled, ProB would sometimes avoid early partial enumeration and later try to find the whole range or domain in a single search step.

Allow enumeration of infinite types: In order to find counterexamples even if infinite sets occur in the sequent, ProB can try to enumerate unbounded variables. Obviously, this cannot lead to a successful proof anymore. Hence, ProB allows to disable the counterexample search if the user is only interested in proofs. This would speed up the search, but for the benchmarks below, we allowed enumeration in all cases.

4.3.3 Results

The benchmark results for the tactics can be found in Tables 4.1 and 4.2 and Figs. 4.5, 4.7 and 4.8, while the results for the provers alone are in Table 4.3 and

(a) Default Rodin Tactic (b) SMT Tactic (c) ProB Tactic

Figure 4.4: Tactic Definitions

Table 4.1: Benchmark Results: Discharged Proof Obligations

Model	# POs	Tactic alone	+ML/PP	+ML/PP+SMT	+ML/PP+SMT+ProB
Landing Gear System 1, Su, et al.	2328	2025	2198 (+173)	2313 (+115)	2313 (0)
Landing Gear System 2, Su, et al.	1188	817	918 (+101)	1182 (+264)	1182 (0)
Landing Gear System 3, Su, et al.	341	150	184 (+34)	276 (+92)	270 (-6)
CAN Bus, Colley	534	324	402 (+78)	404 (+2)	397 (-7)
Graph Coloring, Andriamiarina, et al.	269	95	133 (+38)	171 (+38)	166 (-5)
Landing Gear System, Hansen, et al.	74	58	60 (+2)	62 (+2)	68 (+6)
Landing Gear System, Mammar, et al.	433	220	308 (+88)	409 (+101)	398 (-11)
Landing Gear System, Andre, et al.	619	298	325 (+27)	447 (+122)	448 (+1)
Pacemaker, Neeraj Kumar Singh	370	258	354 (+96)	364 (+10)	369 (+5)
Stuttgart 21 interlocking, Wiegard	202	30	29 (-1)	82 (+53)	99 (+17)

part (b) of Fig. 4.5. Figure 4.7 summarizes the results of the different landing gear models. Individual results are visualized in Appendix A.1, with Fig. A.1 showing the results of the tactics and Fig. A.2 the results of provers alone.

Table 4.1 shows the total number of proof obligations discharged, as well as the number of proof obligations discharged using ML/PP together with SMT and in the last column the number of obligations discharged by using these two proof tactics together with the ProB disprover. Each Venn diagram shows how many proof obligations are discharged by which prover.

Table 4.2 shows the runtimes of the different provers for all proof obligations and for discharged proof obligations individually. As can be seen, average runtimes are much higher if we take all proof obligations into account. This is due to timeouts occurring if provers are too weak to discharge an obligation in the given time. Some provers, such as the SMT solver based ones are able to detect that they will not be able to proof an obligation and report it to the user. However, this is not always the case.

Table 4.2: Benchmark Results: Average Runtimes (in Seconds / PO)

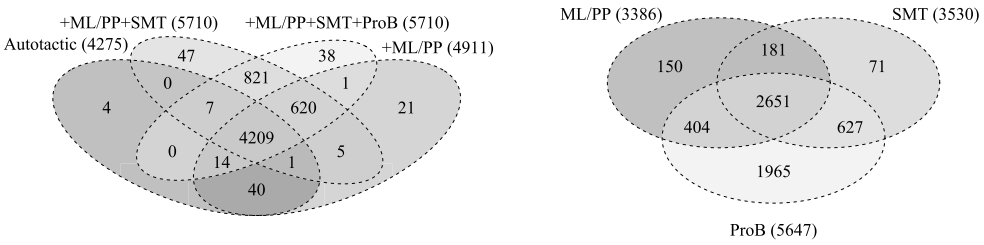
Model	Tactic alone	+ML/PP	+ML/PP+SMT	+ML/PP+SMT+ProB
All Proof Obligations				
Landing Gear System 1, Su, et al.	0.18	0.27	0.27	0.47
Landing Gear System 2, Su, et al.	0.06	0.37	0.44	0.57
Landing Gear System 3, Su, et al.	4.55	4.97	5.78	5.5
CAN Bus, Colley	5.67	5.8	6.12	6.59
Graph Coloring, Andriamiarina, et al.	7.99	8.71	10.6	11.13
Landing Gear System, Hansen, et al.	3.04	3.78	4.07	2.21
Landing Gear System, Mammar, et al.	1.16	1.56	1.81	2.31
Landing Gear System, Andre, et al.	6.28	6.72	8.27	7.03
Pacemaker, Neeraj Kumar Singh	0	0.08	0.06	0.33
Stuttgart 21 interlocking, Wiegard	13.93	14.61	15.47	13.84
Successfully Discharged Proof Obligations				
Landing Gear System 1, Su, et al.	0.02	0.05	0.11	0.31
Landing Gear System 2, Su, et al.	0.02	0.06	0.32	0.45
Landing Gear System 3, Su, et al.	0.01	0.11	1.25	0.37
CAN Bus, Colley	0	0.05	0.1	0.24
Graph Coloring, Andriamiarina, et al.	0	0.14	2.35	2.53
Landing Gear System, Hansen, et al.	0	0.02	0.02	0.2
Landing Gear System, Mammar, et al.	0.01	0.1	0.45	0.31
Landing Gear System, Andre, et al.	0.02	0.05	1.83	0.16
Pacemaker, Neeraj Kumar Singh	0	0.06	0.03	0.26
Stuttgart 21 interlocking, Wiegard	0.08	0.11	1.51	2.23

Except for the graph coloring algorithm, ProB performs surprisingly well. The graph coloring algorithm uses unbounded sets, meaning that some proof obligations cannot be proven using constraint solving and enumeration.

As can be seen in Table 4.1, adding ProB improves the results of automatic proving for some models. The reason for the improvement is that these models only use enumerated sets, booleans and integers as base types. In these cases ProB can produce elaborate case distinctions, combined with constraint solving to narrow down the search space. This type of proof is not supported by the classical provers ML and PP. Generally, the proof obligations that pose problems to ProB are certain well-definedness proof obligations. For instance, function application requires to prove that the parameter is in the domain of the function. Usually this leads to expensive enumeration of the possible parameter values.

For other models, using ProB slows down the prove process. As shown in Table 4.2 ProB's runtime is above average for some proof obligations, while it considerably speeds up other proof attempts. We suspect that this is due to the multiple constraint solver calls ProB performs on different sets of hypotheses as shown in Fig. 4.1. Also, ProB is looking for proofs and counterexamples. This often means that ProB will continue the computation, even after it has realized that no proof is possible (in the hope of finding a counterexample).

It is also interesting to note that, on their own, the ML and PP provers do not fare quite so well as in Table 4.1: they require preprocessing and tactic



(a) Tactics (b) Provers Alone

Figure 4.5: Visualization of the Full Benchmark Results

Table 4.3: Results of Running Provers Alone (without Preprocessing by Rodin)

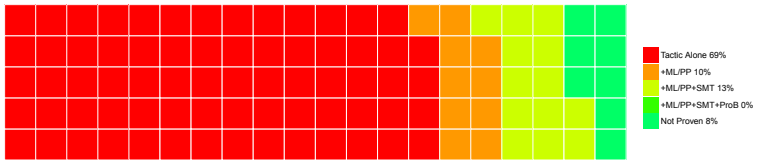
Model	# POs	ML/PP	SMT	ProB	
				prove	disprove
Landing Gear System 1, Su, et al.	2328	1408	1490	2310	0
Landing Gear System 2, Su, et al.	1188	347	570	1176	0
Landing Gear System 3, Su, et al.	341	113	174	289	0
CAN Bus, Colley	534	481	299	282	2
Graph Coloring, Andriamiarina, et al.	269	103	132	1	0
Landing Gear System, Hansen, et al.	74	70	59	74	0
Landing Gear System, Mammar, et al.	433	249	266	412	0
Landing Gear System, Andre, et al.	619	203	269	554	5
Pacemaker, Neeraj Kumar Singh	370	360	224	365	0
Stuttgart 21 interlocking, Wiegard	202	52	47	184	2

support to be fully effective: See Table 4.3 containing the results without any preprocessing.

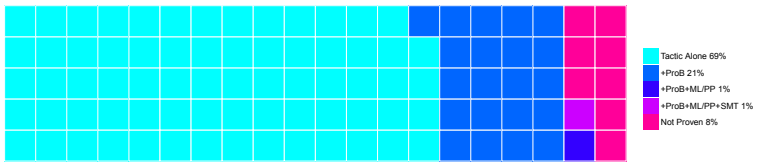
All models except the Landing Gear System by Mammar, et al. show the same behavior: The rate of discharged proof obligations drops significantly if Rodin's default tactics are not applied. Adding SMT solvers or ProB does not replace the tactics either.

In contrast, the model by Mammar, et al. shows the opposite behavior: without preprocessing, more proof obligations can be discharged. This is probably due to the timeouts leaving less time for the actual prover, if we include a preprocessing phase. In future, we want to examine whether better preprocessing can improve the performance of the ProB disprover.

The same effect can be observed in Table 4.4. Here, the performance of the provers on different kinds of proof obligations is given. For most kinds, ProB does perform quite well when compared to ML/PP and the SMT solvers, especially for guard strengthening proofs, theorem proofs and well-definedness proofs. For



(a) ProB Run Last



(b) ProB Run First

Figure 4.6: Influence of Execution Order on Benchmark Results

feasibility and finiteness proof obligations, on the other hand, ProB fares less well. The finiteness proof obligations often involve enumerating different set cardinalities, an operation only weakly supported by the ProB kernel.

Note that for the Stuttgart 21 model and the Andre et al. model, ProB found several unprovable proof obligations, i. e., errors in the model as can be seen in Table 4.3. e. g., for Stuttgart 21 ProB found a counterexample for two proof obligations, while it found five counterexamples in the landing gear model. This is useful feedback to the developer of the model, and the initial purpose of the ProB disprover.

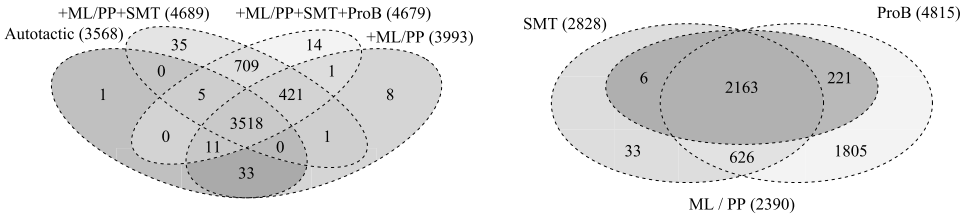
The diagram in Fig. 4.5 shows the gain of using ProB in addition to the other decision procedures. Compared to the SMT Tactic, adding ProB leads to an additional 53 (38 + 1 + 0 + 14) proof obligations being discharged. This is much less than what we reported in our former work [113], where ProB accounted for 304 (238+1+11+54) additionally discharged proof obligations. The reduction is mostly due to the recent improvements of the SMT plugin caused by adding CVC4 and Z3 to the bundled solvers.

However, due to the time consumption by ProB, 53 (47+1+5) proof obligations cannot be discharged anymore. With a higher time-out, these could again be proven.

The second diagram in Fig. 4.5 shows how the individual provers alone contribute: Each of them has a set of proof obligations that cannot be solved by any of the

Table 4.4: Performance of Provers on Different Kinds of Proof Obligations

Kind of PO	# POs	ML/PP	SMT	ProB
Feasibility of non-det. action	59	53 (89.8 %)	40 (67.8 %)	57 (96.6 %)
Guard strengthening	300	30 (10.0 %)	13 (4.3 %)	266 (88.7 %)
Invariant preservation	4950	2925 (59.1 %)	3182 (64.3 %)	4547 (91.9 %)
Action simulation	154	118 (76.6 %)	108 (70.1 %)	134 (87.0 %)
Theorem	98	14 (14.3 %)	30 (30.6 %)	80 (81.6 %)
Well-definedness	780	234 (30.0 %)	146 (18.7 %)	548 (70.3 %)
	6341	3374 (53.2 %)	3519 (55.5 %)	5632 (88.8 %)



(a) Tactics (b) Provers Alone

Figure 4.7: Benchmarks Part 1: Landing Gear Systems

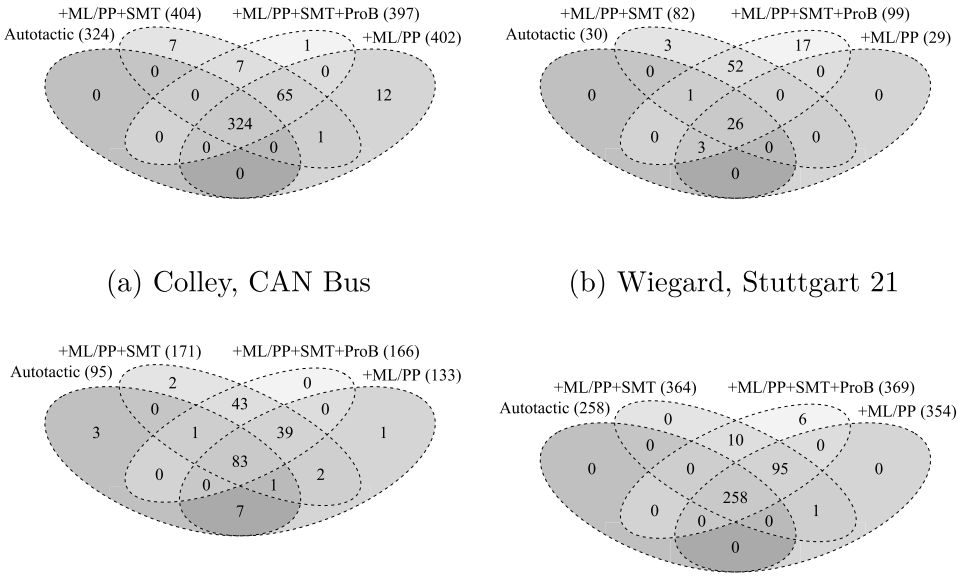
others (150 for ML/PP, 71 for SMT and 1965 for ProB). Again, the performance of the SMT plugin has increased since [113].

Of course the gain achieved by adding another solver to a proof tactic depends on the order in which different provers are called. The effect is visualized in Fig. 4.6. In particular, Fig. 4.6a shows that ProB discharges barely an additional 1% of proof obligations on top of the ones discharged by ML, PP and the SMT plugin of Rodin. If run first, right after the general proof tactic and rewriting rules, ProB discharges an additional 21% as can be seen in Fig. 4.6b.

Influence of Hypothesis Selection on Provers

In [113], we already suspected that the tactic a prover is embedded in can heavily influence its performance. In the following section we will focus on the selection of hypotheses which are given to the provers. The results might influence different implementation details of symbolic model checking algorithms. For instance, we need to consider how and when to clean up inferred hypotheses and whether to prefer a single large predicate over a number of smaller ones.

We performed an additional run of the provers alone, this time after using Rodin's lasso tool to collect all hypotheses sharing a variable with the goal or



(a) Colley, CAN Bus

(b) Wiegard, Stuttgart 21

(c) Andriamiarina and Mèry, Graph Coloring Algorithm

(d) Singh, Pacemaker

Figure 4.8: Benchmarks Part 2: Miscellaneous Models

the already selected hypothesis. The results are given in Table 4.5 and Fig. 4.9. Table 4.5 corresponds to Table 4.3 and lists the performance on the different models. In Figs. 4.9b to 4.9d, we compare running the provers with and without lasso. Figure 4.9 corresponds to Fig. 4.5b, comparing the performance of the three provers when lasso is used.

The provers react differently:

- Performance of ML and PP declines. Without using lasso, they were able to discharge 3386 proof obligations. With lasso, only 2084 proof obligations can be discharged. Among those are 24 obligations which could not be discharged before, as can be seen in Fig. 4.9b.

- ProB drops from 5647 discharged obligations to 5604. The set of discharged obligations is relatively stable, as Fig. 4.9c shows. This is due to the fact that ProB's disproving algorithm is able to use all selected hypotheses anyway. However, the second run highlighted in Fig. 4.1 takes time and thus influences the performance.

- The SMT solvers plugin is able to increase the number of discharged obligations from 3530 to 6036. In consequence, it outperforms both ML/PP and ProB if the full set of hypotheses is given to it. As shown in Fig. 4.9d, only 36 proof obligations cannot be discharged anymore.

Table 4.5: Results of Running Provers with Lasso

Model	# POs	ML/PP	SMT	ProB	
				prove	disprove
Landing Gear System 1, Su, et al.	2328	421	2327	2291	0
Landing Gear System 2, Su, et al.	1188	261	1188	1170	0
Landing Gear System 3, Su, et al.	341	53	331	287	0
CAN Bus, Colley	534	487	504	255	2
Graph Coloring, Andriamiarina, et al.	269	66	194	0	0
Landing Gear System, Hansen, et al.	74	70	63	74	0
Landing Gear System, Mammar, et al.	433	242	432	409	0
Landing Gear System, Andre, et al.	619	119	553	567	5
Pacemaker, Neeraj Kumar Singh	370	356	328	367	0
Stuttgart 21 interlocking, Wiegard	202	9	116	184	2

4.4 Related Work

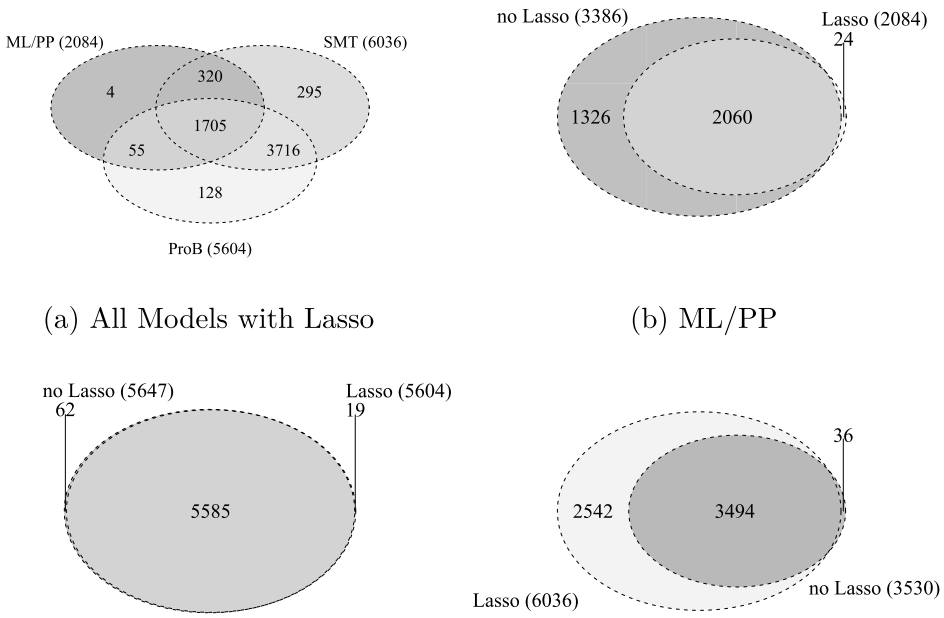
Counterexample generation for proof efforts formulated in Isabelle/HOL [155] has been implement inside Nitpick [29]. Nitpick is based on Kodkod [177], which is also available as a backend for ProB.

Of course, other general purpose solvers can be used to find counterexamples as well. One could rely on SAT solvers such as Glucose [12] or SMT solvers such as Z3 [60]. However, using them to generate counterexamples for B would involve encoding B predicates into a SAT or SMT formula. Furthermore, valuations found by SAT or SMT solvers would have to be translated back into B. For SAT, this can be simplified using Kodkod as done in [160]. For SMT, we will do so in Chapter 6.

4.5 Conclusion and Future Work

One motivation for the experiments conducted in this chapter was the empirical evaluation of our constraint solver, more precisely its capability to serve as the backend of a symbolic model checker. In particular, we were interested in both its ability to detect inconsistencies (a successful proof with the disprover requires finding a contradiction without enumerating unbounded variables; see Fig. 4.1) and its ability to handle infinite domains.

Symbolic model checking aside, finding inconsistencies is important for many other features of ProB, e. g., detecting disabled events during animation. Furthermore, it is useful for constraint-based validation, such as deadlock checking [91], where it avoids the constraint solver exploring infeasible alternatives.



(a) All Models with Lasso

(b) ML/PP

(c) ProB

(d) SMT

Figure 4.9: Comparison with and without Lasso

In the context of model-based testing, it enables ProB to detect uncoverable alternatives, and not spend time trying to find test cases for them.

Another important issue is the soundness of the ProB disprover. In [22], we have presented the various measures we are taking to validate ProB's results in general. In addition, we have developed an SMT-LIB [18] importer for ProB and have applied our disprover to a large number of SMT-LIB benchmarks, checking that no "false theorems" are proven. Techniques and results are presented as a second application of the improved constraint solver in Chapter 5.

To be sure, we have double-checked many of the proof obligations which were only provable by ProB, to ensure that they are indeed provable. As the Venn diagrams in Figs. 4.7 and 4.8 show, numerous proof obligations can be proven by two or even three different provers. As the three provers rely on completely different technologies and have been developed by independent teams, we can have a high confidence that those proof obligations are indeed provable.

We have demonstrated that constraint-based proof in general, and ProB in particular, is capable of discharging proof obligations that currently cannot be proven using Rodin's auto tactic and the SMT solvers. Our prover typically deals well with different kinds of proof obligations than the other provers, and is thus an orthogonal extension rather than a replacement. Rodin's auto tactic

performs well in the realm of set theoretic constructs and relational expressions, some of which cannot be easily represented in the SMT syntax. SMT on the other hand performs well on arithmetic expressions, where the auto tactics often fail. ProB finally covers predicates over enumerated sets, explicit data and explicit computations, and has a good support for integer arithmetic over finite domains.

However, for models making heavy use of deferred sets, such as the graph coloring algorithm model, the ProB disprover can currently mainly play its role as disprover. More precisely, for any proof obligation which involves deferred sets and where no precise value of the cardinality of the deferred set is known, the disprover can only return either a counterexample or the result "unknown".

This is a serious limitation, because several symbolic model checking algorithms rely on the detection of unsatisfiability, as we will show in Chapter 7. In the future, we plan to improve the treatment of deferred sets in ProB, mostly by allowing the constraint solver to determine the cardinalities of sets while solving instead of having the solver working on predefined sets. This should also enable the disprover to act as a prover for more proof obligations involving deferred sets.

In summary, we present the following insights on when to use the ProB disprover (+) and when not to (-):

+ Used solely as a disprover, ProB can prevent futile interactive proof attempts. This is always worthwhile.
+ The inconsistency detection is useful for finding subtle modeling errors.
+ On models such as the ABZ landing gear models (Fig. 4.7), which rely heavily on enumerated sets, booleans and/or bounded integers as base types, ProB performs well.
+ The Stuttgart 21 model shows that explicit data, e. g., track layouts or time tables, can often be used effectively by ProB. Often, this results in a proof by an elaborate case distinction.
+ ProB performs reasonably well on unbounded intervals, when interval reasoning can be applied. This occurs for example in the lower refinement levels of the ABZ case study models or the pacemaker model.
- As soon as the proof goal references deferred sets (e. g., in the graph coloring model), no proof can be done by construction of the disprover (see Fig. 4.1).
- When unbounded datastructures are used, ProB cannot exhaustively enumerate cases and is much less powerful. This happens for example in the CAN bus model that represents a buffer as an unbounded partial function from $\mathbb{N}$ to $\mathbb{Z}$.

In particular, judging by the two drawbacks identified, we can already answer one of the questions posed in the motivation: ProB's constraint solver alone appears to be too weak to be used as the backend of a symbolic model checking algorithm. To overcome the discovered limitations, we suggest an integration of ProB and SMT solvers in Chapter 6.

Symbolic model checking aside, we think that the ProB Disprover is a valuable extension to the existing set of provers, because it can increase the number of proof obligations that are automatically discharged, thus saving time and money. Overall, the outcome of the empirical evaluation was a positive surprise, as ProB's main domain of application is finding concrete counterexamples, not discharging proof obligations.

I have heard there are troubles of more than one kind.
Some come from ahead and some come from behind.
But I've bought a big bat. I'm all ready you see.
Now my troubles are going to have troubles with me!
Dr. Seuss, "I Had Trouble in Getting to Solla Sollew"

5

Application: SMT Solving

In this chapter we will use constraints taken from the SMT-LIB collection of SMT problems to benchmark the improvements to ProB's constraint solving kernel. As in the previous chapter, test cases in the SMT-LIB have been collected from both academia and industry.

5.1 Introduction and Motivation

The SMT-LIB language and its logics [18] and the B language have several similarities. Both are based on predicate logic, they support the same types of arithmetic operators and quantifiers. Furthermore, both are strongly typed languages. However, there are considerable differences as well. For instance, B supports several data types not available in SMT-LIB, like (finite) sets and lists. For SMT-LIB, these only exist as a proposal [183]. However, some solvers already provide partial support, e. g., CVC4 [16] supports finite sets based on the method presented in [40]. SMT-LIB and certain SMT solvers on the other hand are able to cope with real and floating-point arithmetic, while B only supports integer arithmetic.

With our translation we bridge the gap between B and the SMT-LIB language, embedding the translatable parts of SMT-LIB in the B language. This allows reasoning over both SMT-LIB constraints and SMT solving algorithms, using the B method and its tool chain, e. g., one can specify any algorithm working on

SMT-LIB constraints using the B language and prove it correct using Atelier B.

Additionally, as the semantics specified for SMT-LIB are preserved during the translation, one can analyze properties of given SMT-LIB constraints in B. This could be used to perform a meta-level analysis of SMT-LIB.

Furthermore, we want to make the constraint solving and model finding capabilities of ProB available in the form of a standalone constraint solver that takes the SMT-LIB language as input.

Not only does this enable others to use ProB without having to learn B, it also gives us access to the benchmarks and test cases available in the SMT-LIB collection. Since ProB can be used as a prover using the techniques we described in Chapters 3 and 4, asserting its correctness on as many test cases as possible has become vital.

The contributions of this chapter are as follows:

- A translation scheme from the SMT-LIB format to B for logics involving quantification, arrays, free sort and function symbols over integers,
- A discussion about possible translations of bit vectors to B,
- An in-depth empirical evaluation comparing ProB's extended constraint solver to Z3 and CVC4 on several benchmarks taken from the SMT-LIB, notably for the NIA (non-linear integer arithmetic with quantifiers) category, which ProB has won in the 2016 SMT-LIB competition.

5.2 Introductory Examples

First, let us look at two simple examples illustrating the general translation scheme: In Listing 5.1 we encode the assertion $p \wedge \neg p$, which is obviously unsatisfiable. With the first line, we state that the solver should use the logic of quantifier free uninterpreted functions. This is a fact that we do not need to translate to B, because B does not distinguish between different logics. The second line introduces a constant symbol p that is of type boolean. As an SMT constant can have just one value, we translate it to a B constant with the same name.

The type can then be specified using the `PROPERTIES` section of a B machine that holds predicates that have to be true for the constants to be valid. In this case, we assert that $p \in BOOL$, where $BOOL$ is the set consisting of true and false.

Listing 5.1: Boolean Example in SMT-LIB

```
(set-logic QF_UF)
(declare-fun p () Bool)
(assert (and p (not p)))
(check-sat)
```

Listing 5.2: Boolean Example in B

```
MACHINE BooleanExample
CONSTANTS p
PROPERTIES
 p:BOOL & p=TRUE & not(p=TRUE)
END
```

The assertion $p \wedge \neg p$ cannot be written in B as trivially as one might expect, because B does not support the use of booleans as predicates. Hence, the assertion has to be translated as $p \wedge \neg p \Leftrightarrow p = \top \wedge \neg p = \top$.

The complete B machine can be found in Listing 5.2. Once loaded, ProB detects the unsatisfiability and reports that the properties are inconsistent.

The same basic idea can be used for integer arithmetic as shown in Listings 5.3 and 5.4. Here, we solve the indeterminate equation system $6x + 12y + 3z = 30 \wedge 3x + 6y + 3z = 12$. This time, ProB is able to find a valuation for the constants and satisfiability can be reported.

Note that `INTEGER` is the B type for the integers. As with SMT-LIB, the B set of integers represents the mathematical integers. The B method tools in general, and ProB's solver in particular, can handle arbitrarily large integers. The CLP(FD) library employed by ProB may generate overflows, which the ProB solver catches. Once an overflow occurs, ProB tries to provide alternative treatment, i. e., explicit computations instead of constraint based propagation. In case this is not possible, ProB reports "unknown".

In contrast to SMT-LIB, the arithmetic operators of B do not support an arbitrary number of operands. Hence, `(+ x y z)` has to be translated into `x + y + z`.

5.3 Translating SMT-LIB to B

Because the B language provides more involved operators than the SMT-LIB language, translating SMT-LIB into B is mostly straightforward. New SMT-

Listing 5.3: Integer Example in SMT-LIB

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (= (+ (* 6 x) (* 12 y) (* 3 z)) 30))
(assert (= (+ (* 3 x) (* 6 y) (* 3 z)) 12))
(check-sat)
```

Listing 5.4: Integer Example in B

```
MACHINE IntegerExample
CONSTANTS x, y, z
PROPERTIES
 x:INTEGER & y:INTEGER & z:INTEGER &
 6*x + 12*y + 3*z = 30 &
 3*x +  6*y + 3*z = 12
END
```

LIB non-parametric sorts declared by the user are mapped to B *deferred* sets. Deferred sets introduce a new base type and are declared in the `SETS` section of a B machine. They may contain an arbitrary (and possibly infinite) non-zero number of elements.

If a new sort is parametric, further sets are generated on demand for each concrete combination of parameters used in the SMT-LIB input. For instance, the SMT-LIB command `(declare-sort NewSort 0)` results in a B deferred set named `NewSort`.

A parametric sort declared for instance by `(declare-sort Pair 2)` is not directly included in the B machine. As B does not support dynamic typing, we have to wait until the sort is instantiated and the type is fully known. An instantiation of `Pair`, such as `(Pair Int Bool)`, is introduced as a set containing pairs taken from $\mathbb{Z} \times \{true, false\}$. For any higher number of arguments, nested pairs are used.

New function symbols are translated into B constants declared in the B `CONSTANTS` section. The SMT types are thereby mapped to B types, e. g., the `Int` sort is mapped to B's `INTEGER` set. For functions declared using `declare-fun`, we create constants that are defined as total functions, which are typed in B as relations between the parameter types and the result type.

Formally, SMT-LIB sorts are translated into B expressions using the mapping *b_type* defined below. The translation of arrays and bit vectors will be further

discussed in Sections 5.3.3 and 5.3.4

Definition 5.3.1. The translation function *b_type* maps the default SMT-LIB sorts to B types, and is defined recursively as

$$\begin{aligned} b\_type(Int) &= INTEGER \\ b\_type(Bool) &= BOOL \\ b\_type((Array\ F\ T)) &= b\_type(F) \rightarrow b\_type(T) \\ b\_type((\_\ BitVec\ L)) &= [0, L-1] \rightarrow \{0,1\} \end{aligned}$$

where $X \rightarrow Y$ is the set of all total functions from X to Y.

Certain SMT-LIB operators have a direct equivalence in B. The arithmetic operators +, -, * can be translated directly. Furthermore, like SMT-LIB, the B language and tools support universal and existential quantification natively.

There is no absolute value function in B. However it can be translated as $|x| = \max(\{-x, x\})$. Integer division and modulo in SMT-LIB are defined following the Euclidean definition by Boute [32], while B uses a floored division [110].[1] Thus in B $-8/3 = -2$ while in SMT-LIB it is -3.

Furthermore, in B, x mod y is only defined if x is non-negative and y is positive. In contrast, the Euclidean definition mentioned above permits both cases. In Section 5.3.5, we express SMT-LIB's division and modulo by rewriting it to B's floored division.

Another difference is that in SMT-LIB all functions are total, meaning that $2/0$ is an integer value, it is just unknown to us which one. So, in SMT-LIB $x/0 = 2$ is a satisfiable formula, while in B it is ill defined. Currently, our B translation of $x/0 = 2$ is not well-defined in B [4], and ProB will raise an error. In other words, ProB will report unknown.[2]

Aside from the translations above, certain constructs available in the SMT-LIB format are not available in B and require more involved translations. We identified five of them and will explain them in the following sections:

- In B, booleans are values and cannot be used as predicates. We already showed how to overcome this limitation in the introductory examples. Listing 5.2 shows how to turn booleans into predicates simply by comparing them to `TRUE`.
- There is no if-then-else for expressions or predicates in B. The B if-then-else may only be used in substitutions (aka statements). Thus, the if-then-else from SMT-LIB has to be rewritten. How this is done is explained in Section 5.3.1.

[1] More precisely, the definition of division in B [1] is $n/m = \min(\{x \mid x \in \mathbb{Z} \wedge n < m * succ(x)\})$.

[2] We may improve our translation to remedy this, but our assumption is that most SMT-LIB examples are well-defined according to B.

- Analogously, B features a let substitution but no let predicate. We rewrite let as shown in Section 5.3.2.
- B has no dedicated data type adhering to the properties SMT-LIB arrays have to fulfill. However, these can be represented by relations together with a special set of additional assertions. See Section 5.3.3 for details.
- Lastly, B has no integrated support for bit vectors. Again these can be simulated. However, it is less obvious how to encode them. Several possible encodings will be discussed in Section 5.3.4.

Another limitation is that there are no real or floating-point numbers in B [1]. There has been experimental support for floats and reals in Atelier-B since Atelier B 4.1 [55]. However, proof support is far from being useful and there is currently no support for reals or floats in ProB. Hence, we currently do not take them into account in our translation.

5.3.1 If-Then-Else

As mentioned above, the if-then-else construct in B can only be used in substitutions, not in predicates or expressions. However, the core theory of the SMT-LIB language supports the `ite` function that returns its second or third argument depending on the truth value of the first one. To mimic the behavior of `ite` we can reuse a translation that has originally been developed for a translation from TLA^+ to B [93].

In order to translate `(ite` P E_1 E_2`)` we first have to look at the type of E_1 and E_2. If both are predicates, the if-then-else can be expressed in terms of implications, i. e., `(ite` P E_1 E_2`)` is translated into

$$(P \Rightarrow E_1) \land (\neg P \Rightarrow E_2).$$

Implication is available in B, so the predicate above can be used as a replacement for `ite` in case the arguments are all predicates.

When the `ite` construct is used as an expression rather than a predicate, the translation is more complex. We use the translation suggested by [93]: For both branches of the `if` we create a lambda function that maps 1 to E_1 or E_2 respectively. Afterwards, the union of the two lambda relations is computed:

$$(\lambda\, t \cdot (t = 1 \mid E_1)) \cup (\lambda\, t \cdot (t = 1 \mid E_2)).$$

This relation has two elements both mapping 1 to a result. In order to mimic the behavior of if-then-else, we now have to assure that one of the lambda relations is empty depending on the value of P:

$$(\lambda\, t \cdot (t = 1 \land P \mid E_1)) \cup (\lambda\, t \cdot (t = 1 \land \neg P \mid E_2)).$$

Now, either P or $\neg P$ is false, making the respective lambda expression to be the empty set (and avoiding the evaluation of the corresponding expression E_i). The other lambda expression maps 1 to the result of `(ite` P E_1 E_2`)`. Hence, we just have to apply the relation to 1 to extract the result:

$$(ite\, P\, E_1\, E_2) == \\ (\lambda\, t \cdot (t = 1 \wedge P \mid E_1)) \cup (\lambda\, t \cdot (t = 1 \wedge \neg P \mid E_2))(1).$$

Observe that B strictly distinguishes between boolean values and predicates. However, there are operators to convert between the two. We considered other encodings of if-then-else, such as using a constant function. However, they often did not harmonize with the inner workings of B and its tools. The encoding presented above exhibits the best performance so far.

Take for example the SMT-LIB formula, where we suppose x to be a natural number:

$$(ite\, (not\, (=\ x\, 0))\, (div\, 10\, x)\, (div\, 10\, (-\, x\, 1)))$$

Our translation is

$$(\lambda\, t \cdot (t = 1 \wedge x \neq 0 \mid 10/x)) \cup (\lambda\, t \cdot (t = 1 \wedge \neg(x \neq 0) \mid 10/(x-1)))(1)$$

One may think we could translate this into a simpler B expression:

$$\{TRUE \mapsto 10/x, FALSE \mapsto 10/(x-1)\}(bool(x \neq 0))$$

However, it has the problem that in order to determine the value of the expression, one needs to compute the value of the subexpression $\{TRUE \mapsto 10/x, FALSE \mapsto 10/(x-1)\}$. Thus, when x=0, we still need to evaluate $10/x$, generating a well-definedness error in B. Similarly, when x=1 we still need to evaluate $10/(x-1)$, again causing a well-definedness error.

5.3.2 Let

As with if-then-else, B only features a let substitution. Inside of an expression or predicate no let construct is available in B. SMT-LIB on the other hand includes a let construct for both expressions and predicates.

According to the SMT-LIB standard [18] one can rewrite a let of the form

$$(let\, ((x_1\, t_1)\, \ldots\, (x_n\, t_n))\, t)$$

by replacing all free occurrences of x_i in t with t_i for all $i = 1, \ldots, n$. This step may have to include renaming in order to avoid scoping errors due to capturing by quantifiers.

Regarding performance, simple inlining of the t_i may lead to duplicated computation during solving. To some extent this could be countered by ProB's common subexpression detection. However, this would be equal to re-introducing the let internally. In order to avoid duplicating computation we suggest a translation comparable to the one of the if-then-else.

First, let us consider the case of a let where t is a predicate. In this case we can rewrite

$$(let\,((x_1\,t_1)\,\ldots\,(x_n\,t_n))\,t)$$

to

$$\exists\, x_1, \ldots, x_n \cdot t \wedge \bigwedge_{i=1}^{n} x_i = t_i$$

which can be written in B without further translation.

Replacing a let where t is an expression cannot be done as easily. As in Section 5.3.1 we create a function that is called on a fixed value. We translate

$$(let\,((x_1\,t_1)\,\ldots\,(x_n\,t_n))\,t)$$

to

$$\{k, v \mid k = 1 \wedge \exists\, x_1, \ldots, x_n \cdot v = t \wedge \bigwedge_{i=1}^{n} x_i = t_i\}(1).$$

The set comprehension contains only one element: The pair $(1, v)$ where v is equal to t, the expression copied from inside the let binder. We call this function on 1 to extract v.

As an example, consider $(let\,((x_1\,1)\,(x_2\,2))\,(+\,x_1\,x_2))$. We translate this to $\{k, v \mid k = 1 \wedge \exists\, x_1, x_2 \cdot v = x_1 + x_2 \wedge x_1 = 1 \wedge x_2 = 2\}(1)$. The existential quantification can be removed by inlining the definition of the quantified variables, simplifying the comprehension to $\{k, v \mid k = 1 \wedge v = 1 + 2\}(1)$, which represents the partial function $1 \mapsto 1 + 2$. The function is applied to 1 in order to extract the desired result, i. e., $\{k, v \mid k = 1 \wedge v = 1 + 2\}(1) = \{(1 \mapsto 3)\}(1) = 3$.

5.3.3 Arrays

Another SMT-LIB construct that does not have an obvious counterpart in B are array sorts together with the functions to store or select elements. In B, there is no native data type for arrays or maps. The closest equivalent is a partial function that maps keys to values.

An array a in SMT-LIB (using the `ArraysEx` theory from Chapter 3.7 of [18]) with key sort s_1 and value sort s_2, satisfies the following axioms for the two operations *select* and *store*:

1. $\forall(i, e).(i \in s_1 \wedge e \in s_2 \Rightarrow select(store(a, i, e), i) = e)$
2. $\forall(i, j, e).(i \in s_1 \wedge j \in s_1 \wedge e \in s_2 \wedge i \neq j \Rightarrow select(store(a, i, e), j) = select(a, j))$
3. $\forall\, b.(b \in Array(s_1, s_2) \wedge (\forall\, i.(i \in s_1 \wedge select(a, i) = select(b, i)) \Rightarrow a = b))$

First, in order to solve constraints involving arrays in ProB we need a translation for the array sorts themselves, i. e., for the types of constants representing arrays. A typical declaration of a new function symbol named *arr* that represents an array with integer keys and integer values can be done in SMT-LIB by `(declare-fun arr () (Array Int Int))`.

We again base our translation on the notion of relations and functions. We set up a new B variable with the name *arr*, typed as a total function from and to the set of all integers: $arr \in INTEGER \rightarrow INTEGER$. A total function in B fulfills the same properties as the SMT-LIB logics require the arrays to fulfill.

With the translation of array sorts in place, we can implement translations for the array operations *store* and *select*: The `select` operation takes an array A and a key K and returns a value V.

Definition 5.3.2. Given that the array is encoded as a total function as described above, we can mimic the behavior of *select* by the function application of B:

$$(select\, A\, K) == A(K).$$

Storing an element inside of an array is done by the SMT-LIB function *store*, which takes an input array A, a key K and the value to store V. It returns the updated array. In order to translate *store* to B, we need several B operators for relations:

Definition 5.3.3. For a set S and a relation r, the domain subtraction $\lhd\!\!\!\!-$ in B is defined as

$$S \lhd\!\!\!\!- r = \{x \mapsto y \mid x \mapsto y \in r \wedge x \notin S\}.$$

For relations r_1, r_2, the override $\lhd\!\!\!\!-$ in B is defined as

$$r_1 \lhd\!\!\!\!- r_2 = r_2 \cup (dom(r_2) \lhd\!\!\!\!- r_1).$$

Definition 5.3.4. Using the functional operators given above, `store` can be expressed by a functional override that replaces any existing pair $K \mapsto O$ by a newly created pair $K \mapsto V$:

$$(store\, A\, K\, V) == A \lhd\!\!\!\!- \{K \mapsto V\}.$$

In summary, the simple array property stated in Listing 5.5 (taken from the SMT-LIB set of benchmarks) is translated into the B machine shown in Listing 5.6. ProB is able to find a valuation for the variables `k`, `v1`, `v2` and `a`.

Listing 5.5: Array Example in SMT-LIB

```
(set-logic QF_AX)
(declare-sort Keys 0)
(declare-sort Values 0)
(declare-fun k () Keys)
(declare-fun v1 () Values)
(declare-fun v2 () Values)
(declare-fun a () (Array Keys Values))
(assert (= v1 v2))
(assert (= (store a k v1) (store a k v2)))
(check-sat)
```

Listing 5.6: Array Example in B

```
MACHINE ArraysExample
SETS
 Keys; Values
CONSTANTS k, v1, v2, a
PROPERTIES
 k : Keys & v1 : Values & v2 : Values &
 a : Keys --> Values &
 v1 = v2 &
 a <+ {(k,v1)} = a <+ {(k,v2)}
END
```

5.3.4 Bit Vectors

In order to translate SMT-LIB's bit vector logic to B, we first need to decide on a suitable representation of bit vectors in B. This turns out to be much more complex than it was for arrays. This is mostly due to the amount of different operations that the SMT-LIB language permits for bit vectors.

We evaluated several representations which we will discuss in this section:

Approach 1. Represent bit vectors as B sequences, i. e., as total functions mapping an integer interval $[1, n]$ to $\{0, 1\}$.

Approach 2. Work on the word-level and represent a bit vector by the integer value it encodes. That way we might be able to exploit some higher order operations available for integers in B.

Approach 3. Use a combination of the above combined with functions to convert between bit vector sequences and natural numbers to always use the representation in which an assertion is easier to express. Essentially this would mean to implement `nat2bv` and `bv2nat` as defined by the theory

`FixedSizeBitVectors`[3]. A B implementation of both is possible as we will show below.

Approach 1 is equivalent to the well-known bit blasting done by most state-of-the-art SMT solvers: The bit vector operations are broken down into propositional formulas, possibly after adding some rewriting or high-level reasoning [80]. Afterwards, a SAT solver is used to solve the result.

From former experiments we already know that ProB is not competitive on simple SAT instances, as it is tailored towards high-level constraints as occurring in B and Event-B. Thus, no special effort was put into solving low-level boolean formulas. Furthermore, we would probably lose the advantages of using a high-level language like B. Approach 1 (on its own) was thus quickly discarded.

Approach 2 has already been used for example for test case generation [77]. One of the problems of this approach is that various bit-level operations cannot easily be expressed at the word-level. Some of them involve encoding into multiple constraints or the usage of several case splits. Furthermore, a comparison between SAT solvers, SMT solvers and constraint logic programming for hardware verification shows that the approach is by far the slowest [176].

We decided to follow approach 3 and solve bit vectors using both word level arithmetic and propositional logic. Implementing conversion functions between bit vectors as sequences and integers in B would allow us to combine both approaches, as done in the definition of the semantics of the `FixedSizeBitVectors` theory.

$bv2nat$, which converts a bit vector to an integer is defined as follows.

Definition 5.3.5. $bv2nat \in \text{bit vectors} \to \mathbb{Z}$, with

$$bv2nat(b) = b(m-1) * 2^{m-1} + b(m-2) * 2^{m-2} + \cdots + b(0).$$

This can be expressed in B by using the general sum operator:

$$bv2nat(b) = \Sigma_{k,v}\{v * 2^{k-1} \mid (k \mapsto v) \in b\}.$$

The opposite direction, translating natural numbers into bit vectors, is handled by $nat2bv$:

Definition 5.3.6. $nat2bv \in \mathbb{Z} \to \text{bit vectors}$, returns a bit vector b such that

$$b(m-1) * 2^{m-1} + b(m-2) * 2^{m-2} + \cdots + b(0) = n \text{ rem } 2^m,$$

where m is the width of the bit vector. The resulting sequence in B can be computed by the means of a set comprehension:

$$nat2bv(n, m) = \lambda\, k.(k \in [1, m] \mid \frac{n}{2^{k-1}} \bmod 2).$$

[3] See `http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml`

Now that we can convert back and forth between the two bit vector representations, we can pick either one as the internal representation in ProB. The B representation as a (total) function mapping indices to $\{0, 1\}$ allows ProB to reason about single bits and to propagate values for bit vectors only partially known so far. We thus use it as the default representation and convert to natural numbers only if necessary.

Currently, *bv2nat* is used to implement arithmetic operations and unsigned comparison operators as well as *bvneg*, the 2's complement unary minus. For the signed comparison operators, an extended version of *bv2nat* that takes the sign bit into account is used.

Some operators, especially the bit shifts, can be implemented in both ways. Applied on the sequence representation, they shift the indices by an offset and add or remove new elements accordingly. The same result could be achieved by converting to a natural number, multiplying with an appropriate power of 2 and converting back.

We do not know a priori which encoding will provide better propagation as it highly depends on intertwined constraints. Hence, we decided to set up both word- and bit-level encoding of a constraint at the same time wherever possible. Once one of them leads ProB to a solution, the other is solved immediately via the *bv2nat* bridge.

Below, we will show how our translation works, using different examples. First, an SMT-LIB declaration of a constant symbol *bv* as a bit vector of length 32 is given by `(declare-fun bv () (_ BitVec 32))`. In B, we constrain the variable to hold a total function mapping $[0, 31]$ to $\{0, 1\}$: $bv \in 0..31 \rightarrow \{0, 1\}$.

We give two examples for our translation that outline bit-level as well as word-level reasoning. Bit-level operations are shown in Listings 5.7 and 5.8, where we show how to prove De Morgan's law for bit vectors of length 4. The proof is performed by showing the absence of a counterexample. The SMT-LIB file in Listing 5.7 is translated into the B machine given in Listing 5.8.

The three bitwise operators *bvor*, *bvand* and *bvnot* are included in the machine using B definitions. Each computes a new set comprehension representing the result as another total function, using our representation introduced above. Due to the lack of an if-then-else construct in B predicates, we use two implications to define the result.

Listings 5.9 and 5.10 show an example of word-level arithmetic. We introduce two constants `x` and `y` and assign a fixed value to each. Afterwards, we assert that `z` is equal to the sum. In this example, we use B's integer arithmetic by translating back and forth between bit- and word-level using *bv2nat* and *nat2bv* as introduced above.

Listing 5.7: Bitwise Operators Example in SMT-LIB

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 4))
(declare-fun y () (_ BitVec 4))
(assert (not (= (bvand (bvnot x) (bvnot y))
                (bvnot (bvor x y)))))
(check-sat)
```

Listing 5.8: Bitwise Operators Example in B

```
MACHINE BVExampleDeMorgan
DEFINITIONS
 bvor(b1,b2,Len)  == {k,v|k:0..Len & v:0..1
        & (b1(k)=1 or b2(k)=1 => v=1)
        & (b1(k)=0 &  b2(k)=0 => v=0)};
 bvand(b1,b2,Len) == {k,v|k:0..Len & v:0..1
        & (b1(k)=1 &  b2(k)=1 => v=1)
        & (b1(k)=0 or b2(k)=0 => v=0)};
 bvnot(b,Len)     == {k,v|k:0..Len & v:0..1
        & (b(k)=1 => v=0) & (b(k)=0 => v=1)}
CONSTANTS x, y
PROPERTIES
 x:0..3-->{0,1} & y:0..3-->{0,1} &
 bvnot(bvor(x,y,3),3) /= bvand(bvnot(x,3),bvnot(y,3),3)
END
```

As can be seen, our translation directly follows the given semantics of bit vectors in SMT-LIB. This approach will certainly not be competitive when it comes to performance. However, because it stays quite high-level, it allows using the B method tools to reason about SMT-LIB constructs and algorithms using them.

Given these expectations, we did not evaluate the performance of ProB and other B method provers on the full set of bit vector benchmarks found in the SMT-LIB. Running a selection of benchmarks supports our expectations as can be seen in Section 5.4.

5.3.5 Formal Definition of Translation

The translation is implemented as an AST walker carrying around a type environment. We will define how AST nodes are translated by gradually defining a translation function τ, mapping SMT-LIB to B. Just as above, we will use

Listing 5.9: Word-Level Operators Example in SMT-LIB

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 4))
(declare-fun y () (_ BitVec 4))
(declare-fun z () (_ BitVec 4))
(assert (= x #b0011))
(assert (= y #b0001))
(assert (= z (bvadd x y)))
(check-sat)
(get-model)
```

Listing 5.10: Word-Level Operators Example in B

```
MACHINE BVExampleAdd
DEFINITIONS
 bv2nat(b)       ==
   SIGMA(k,v).((k,v) : b | v * 2**k);
 nat2bv(n,width) ==
   {k,v | k:0..width-1 & v=(n / 2**k) mod 2}
CONSTANTS x, y, z
PROPERTIES
 x:0..3-->{0,1} & y:0..3-->{0,1} &
 z:0..3-->{0,1} &
 x = {(0,1),(1,1),(2,0),(3,0)} &
 y = {(0,1),(1,0),(2,0),(3,0)} &
 z = nat2bv(bv2nat(x) + bv2nat(y),4)
END
```

mathematical notation instead of B's ASCII syntax. However, the mathematical representation can be expressed in B without further translation.

Sorts are mapped to B types as shown in Definition 5.3.1. Existential and universal quantifiers are directly available in B. Let and if-then-else are translated as stated above. The theory of arrays only features the two operators we already defined a translation for in Definitions 5.3.2 and 5.3.4 and will thus not be considered further.

In the following, we will only discuss the unary and binary cases of SMT-LIB operators. Higher arities are realized according to the SMT-LIB standard [18]:

- For an operator f defined to be left-associative, we treat $(f\ t_1 \ldots t_n)$ with $n > 2$ as $(f\ (f\ t_1 \ldots t_{n-1})\ t_n)$.
- For an operator f defined to be right-associative, we treat $(f\ t_1 \ldots t_n)$ with $n > 2$ as $(f\ t_1\ (f\ t_2 \ldots t_n))$.

- For an operator f defined to be chainable, we treat $(f\ t_1 \ldots t_n)$ with $n > 2$ as $(and\ (f\ t_1\ t_2)\ (f\ t_2\ t_3) \ldots (f\ t_{n-1}\ t_n))$.
- For an operator f defined to be applied pairwise, we treat $(f\ t_1 \ldots t_n)$ with $n > 2$ as $(and\ (f\ t_1\ t_2) \ldots (f\ t_1\ t_n)\ (f\ t_2 \ldots t_n))$ and process $(f\ t_2 \ldots t_n)$ recursively.

For the SMT-LIB Core theory, we translate as follows:

$$\begin{aligned}
\tau(true) &= \top & \tau(false) &= \bot \\
\tau((=>\ x_1\ x_2)) &= (\tau(x_1) \Rightarrow \tau(x_2)) & \tau((and\ x_1\ x_2)) &= (\tau(x_1) \wedge \tau(x_2)) \\
\tau((or\ x_1\ x_2)) &= (\tau(x_1) \vee \tau(x_2)) & \tau((=\ x_1\ x_2)) &= (\tau(x_1) = \tau(x_2)) \\
\tau((distinct\ x_1\ x_2)) &= (\tau(x_1) \neq \tau(x_2)) & & \\
\tau((xor\ x_1\ x_2)) &= ((\neg\tau(x_1) \wedge \tau(x_2)) \vee & (\tau(x_1) \wedge \neg\tau(x_2))) &
\end{aligned}$$

For the SMT-LIB Ints theory, we translate as follows:

$$\begin{aligned}
\tau(NUMERAL) &= NUMERAL & \tau(-x) &= -\tau(x) \\
\tau((-\ x_1\ x_2)) &= (\tau(x_1) - \tau(x_2)) & \tau((+\ x_1\ x_2)) &= (\tau(x_1) + \tau(x_2)) \\
\tau((*\ x_1\ x_2)) &= (\tau(x_1) * \tau(x_2)) & \tau((abs\ x)) &= (\max(\{-\tau(x), \tau(x)\})) \\
\tau((<=\ x_1\ x_2)) &= (\tau(x_1) \leq \tau(x_2)) & \tau((<\ x_1\ x_2)) &= (\tau(x_1) < \tau(x_2)) \\
\tau((>=\ x_1\ x_2)) &= (\tau(x_1) \geq \tau(x_2)) & \tau((>\ x_1\ x_2)) &= (\tau(x_1) > \tau(x_2))
\end{aligned}$$

Division and modulo are rewritten as discussed above, i. e., we express the Euclidean definitions in terms of B's floored division.

$$\begin{aligned}
\tau((div\ x_1\ x_2)) &= \tau(fdiv\ (-\ x_1\ (ite\ (<\ x_1 0)\ (ite\ (<\ x_2\ 0)\ (-\ 0\ 1\ x_2)\ (-\ x_2\ 1))\ 0))\ x_2) \\
\tau((fdiv\ x_1\ x_2)) &= (\tau(x_1)\ /\ \tau(x_2)) \\
\tau((mod\ x_1\ x_2)) &= \tau((-\ x_1\ (*\ x_2\ (div\ x_1\ x_2))))
\end{aligned}$$

The theory of Fixed Size Bit Vectors is translated as follows: Bit vector literals can be trivially translated into their B correspondent. For the operands we follow the theory definition and have that given s is of sort $(\_\ BitVec\ n)$ and t is of sort $(\_\ BitVec\ m)$:

$$\begin{aligned}
&\tau((concat\ s\ t)) = \\
&\{(x, v) \mid x \in [0, n+m) \wedge (x < m \Rightarrow v = \tau(t)(x)) \wedge (x \geq m \Rightarrow v = \tau(s)(x-m))\}
\end{aligned}$$

Furthermore, for s of sort $(\_\ BitVec\ l)$ with $0 \leq j \leq i \leq l$

$$\tau(((\_\ extract\ i\ j)\ s)) = \{(x, v) \mid x \in [0, \tau(i) - \tau(j) + 1) \wedge v = \tau(s)(\tau(j) + x)\}$$

Table 5.1: Benchmarks: Results

solver	configuration	sat	unsat	unknown	timeout	memory out
ProB	vanilla	8871	2928	22566	9272	22
ProB	random	8871	2928	22565	9273	22
ProB	cse	8871	2895	22443	9428	22
ProB	chr	8872	2899	22273	9603	12
Z3	default	16519	20018	1224	5897	1
CVC4	default	8117	20037	9393	6112	0

For s and t of sort $(\_\ BitVec\ m)$ and $0 < m$ and $bv2nat$ and $nat2bv$ as defined in Section 5.3.4 we translate

$$\begin{aligned}
\tau((bvnot\ s)) &= \{(x,v) \mid x \in [0,m) \land (\tau(s)(x) = 0 \Rightarrow v = 1) \land (\tau(s)(x) = 1 \Rightarrow v = 0)\} \\
\tau((bvand\ s\ t)) &= \{(x,v) \mid x \in [0,m) \land (\tau(s)(x) = 0 \Rightarrow v = 0) \land (\tau(s)(x) = 1 \Rightarrow v = \tau(t)(x))\} \\
\tau((bvor\ s\ t)) &= \{(x,v) \mid x \in [0,m) \land (\tau(s)(x) = 1 \Rightarrow v = 1) \land (\tau(s)(x) = 1 \Rightarrow v = \tau(t)(x))\} \\
\tau((bvneg\ s)) &= (nat2bv(m, 2^m - bv2nat(\tau(s)))) \\
\tau((bvadd\ s\ t)) &= (nat2bv(m, bv2nat(\tau(s)) + bv2nat(\tau(t)))) \\
\tau((bvmul\ s\ t)) &= (nat2bv(m, bv2nat(\tau(s)) * bv2nat(\tau(t)))) \\
\tau((bvshl\ s\ t)) &= (nat2bv(m, bv2nat(\tau(s)) * 2^{bv2nat(\tau(t))})) \\
\tau((bvlshr\ s\ t)) &= (nat2bv(m, bv2nat(\tau(s))/2^{bv2nat(\tau(t))})) \\
\tau((bvult\ s\ t)) &= (bv2nat(\tau(s)) < bv2nat(\tau(t)))
\end{aligned}$$

In the definition of the theory, the two operations bvudiv and bvurem are guarded by $bv2nat(t) \neq 0$, preventing a division by zero. Thus, the same well-definedness aspects we already discussed for the integer division come into play here as well. In contrast to the integer theory, truncated division is used for the bit vectors.

$$\begin{aligned}
\tau((bvudiv\ s\ t)) &= nat2bv(m, bv2nat(\tau(s))/bv2nat(\tau(t))) \\
\tau((bvurem\ s\ t)) &= nat2bv(m, bv2nat(\tau(s)) \bmod bv2nat(\tau(t)))
\end{aligned}$$

5.4 Empirical Evaluation

To assert the usefulness of our translation and the performance of ProB's improved constraint solver, we performed different empirical evaluations. We translated benchmarks taken from the SMT-LIB collection to B and used ProB as an SMT solver.

We compared ProB to Z3 [60] and CVC4 [16] on benchmark files taken from the 2014–06–03 snapshot of the SMT-LIB benchmark repository. We used development snapshot versions of all three solvers. We limited the benchmarks to non-incremental ones taken from the following logics:

- (QF_)ALIA, (QF_)AUFLIA,
- (QF_)BV, QF_AUFBV, (QF_)UFBV
- QF_AX,
- QF_IDL,
- (QF_)LIA, (QF_)NIA,
- (QF_)UF, (QF_)UFIDL, (QF_)UFLIA, (QF_)UFNIA.

Logics starting with QF are quantifier-free. The different and possibly combined abbreviations for logics are LIA for linear integer arithmetic, NIA for non-linear integer arithmetic, A or AX for arrays and UF for uninterpreted functions. IDL represents integer difference logic, i. e., a logic where only expressions of the form $r = o_1 - o_2$ are supported.

In addition to comparing the different provers, we also compared several options of ProB's constraint solver. In particular, we compared the vanilla ProB with every of the following options disabled to:

- A version using random instead of linear enumeration of CLP(FD) domains,
- A version using common sub-expression elimination, and
- A version featuring an extended rule set of CHR [81] rules used to infer certain facts CLP(FD) is unable to infer on its own.

All benchmarks were run on the StarExec cluster [174]. The machines used feature an Intel Xeon E5–2609 Quad-Core CPU running at 2.4 GHz and 256 GB of RAM. Red Hat Enterprise Linux Workstation 6.3 was used as the operation system.

Regarding time and memory limits, we used the same values as the latest iteration of the SMT Competition [17]. The timeout was set to 1500 seconds (25 minutes) walltime and CPU time for all solvers. Solvers were enforced to use 100 GB of memory or less. Otherwise, the process was terminated.

Table 5.1 lists the total number of benchmarks detected satisfiable or unsatisfiable by the different solvers. As was to be expected, ProB is outperformed by Z3 and CVC4. This is especially caused by the high number of benchmarks ProB has to report "unknown" on. In most cases, this is due to infinite sets that would need to be enumerated exhaustively in order to solve the constraint or to prove it unsatisfiable. In certain cases, ProB is able to detect that any further attempt is futile and gives up reporting "unknown".

Table 5.2: Benchmarks: Average Runtimes (in s)

solver	configuration	# successful tests	⌀ runtime sat	⌀ runtime unsat
Successful Tests per Solver				
ProB	vanilla	11799	7.66	14.93
ProB	random	11799	7.69	14.99
ProB	cse	11766	7.68	14.54
ProB	chr	11771	8.97	13.91
Z3	default	36537	8.25	8.03
CVC4	default	28154	25.55	19.92
Common Successful Tests				
ProB	vanilla	3999	19.71	3.89
ProB	random	3999	19.71	3.9
ProB	cse	3999	19.66	3.92
ProB	chr	3999	42.11	11.18
Z3	default	3999	0.19	0.1
CVC4	default	3999	2.49	0.66

Furthermore, the table clearly shows that ProB performs especially well on satisfiable benchmarks. Again, this is due to the CLP(FD) based solving kernel. Using constraint programming, it is by far easier to find a valuation of variables than to detect unsatisfiability. Especially for infinite domains, the latter might even be impossible.

Figure 5.1a shows the number of test cases solved by each of the solvers individually as well as the number of test cases multiple solvers were able to solve. In addition to the comparison on all tests, we present the results on benchmarks involving integer arithmetic, tests involving uninterpreted functions and tests involving bit vectors. Keep in mind that certain benchmarks fit into multiple categories, e. g., a benchmark from the QF_UFBV logic uses both uninterpreted functions and bit vectors. Figure 5.2 compares the results of ProB, CVC4 and Z3 on the different logics mentioned above. In both cases, we only consider the "vanilla" configuration of ProB, in order not to compare CVC4 and Z3 with a portfolio of ProB-based solvers. As can be seen, CVC4 and Z3 each contribute some test cases they alone were able to solve. Furthermore, there is a large class of test cases that can be solved by CVC4 and Z3 at the same time.

Interestingly, the diagram shows that only ProB is able to solve certain benchmarks under the conditions explained above. We suspect that this is due to the different technologies used by ProB and Z3 / CVC4. As can be seen in Fig. 5.1b and Table 5.3, the benchmarks solved by ProB (vanilla) stem from the logics involving linear and non-linear integer arithmetic. Figures 5.1c and 5.1d show that ProB (vanilla) cannot compete with the other solvers on benchmarks

involving uninterpreted functions or bit vectors. Consequently, most benchmarks that could only be solved by CVC4 or Z3 involve uninterpreted functions, arrays or bit vectors. The other configurations of ProB paint a similar picture.

The benchmarks that are only solved by ProB can easily be classified:

1. All of them are satisfiable,
2. They rely on integer arithmetic, with most of them involving non-linear constraints,
3. Variables are highly intertwined,
4. Z3 and CVC4 run out of time. Memory does not seem to be an issue.

In addition to the result, we measured the runtimes of the different solvers. Table 5.2 shows how long the solvers took to produce different results. On average, ProB is much faster for satisfiable benchmarks than for unsatisfiable ones. This is to be expected from a tool that has mainly been used to find models of formulas. In contrast, CVC4 and Z3 runtimes do not differ much between satisfiable and unsatisfiable benchmarks.

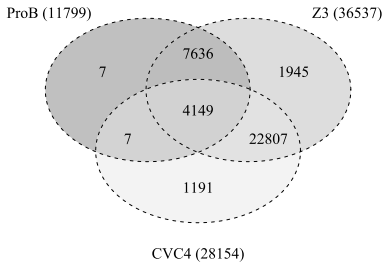
For better comparability, the table also shows the average runtimes on the benchmarks all solvers can solve. ProB is one to two orders of magnitude slower than the dedicated SMT solvers. However, ProB's runtime includes the time spent in the translation phase.

Summarizing, we suspect that, especially for detection of satisfiability, a CLP(FD) based approach can be a useful addition to DPLL($\mathcal{T}$) based algorithms if used in a solver portfolio. Since we introduced a certain overhead by the translation, a direct implementation should add to the gain.

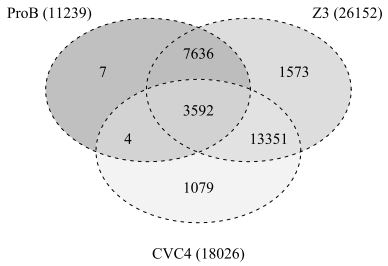
In addition to the benchmarks above, we evaluated how the different options the ProB kernel can be tweaked with influence its performance on SMT-LIB benchmarks. Fig. 5.3 shows the results.

Of the four options mentioned above, only the "CHR" option has significant influence on the number of solved test cases. However, this influence is neither positive nor negative. On the one hand, the CHR rules help to identify unsatisfiable predicates. As can be seen, there are a significant number of test cases that can only be solved with the option enabled. On the other hand, evaluating more propagation rules takes up more time. There is a set of benchmarks that used to be solvable within the timeout, but is not solvable with the extended rule set. Mostly, these are satisfiable.

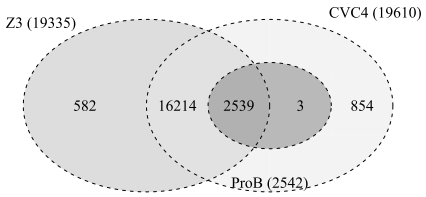
Differing from our expectations, the random enumeration option has close to no influence on the results. In theory, enumerating integer intervals randomly results in a better distribution of checked possible solutions. Hence, a solution might be hit before a timeout occurs or before the linear enumeration was able



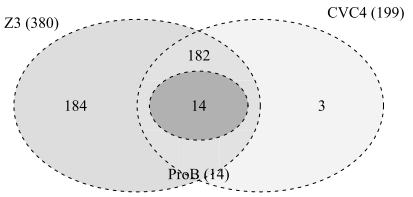
(a) All Tests



(b) Involving Integer Arithmetic



(c) Involving Uninterpreted Functions



(d) Involving Bit Vectors

Figure 5.1: Performance Comparison: Solvers on different logics

to reach it. In contrast, unsatisfiable benchmarks might now run into a timeout due to the added overhead of randomizing, while for satisfiable benchmarks solutions could as well be missed due to the randomization.

Improvements to ProB

Even though ProB is often unable to keep up with CVC4 and Z3, running the benchmarks has been a success for us. Deploying ProB on the numerous new test cases has increased our test coverage. Due to the different nature of constraints occurring in SMT-LIB files and B machines, this was particularly true for code not heavily tested by our regular test suite.

As we used ProB's proving capabilities as in Chapter 4 to try to show unsatisfiability, any inconsistency made obvious by the new test cases could have lead to a bug in our prover.

During the benchmarks, we found bugs both in the SMT-LIB translation and in ProB's solving kernel. Usually, these were made obvious by benchmarks in which the SMT solvers did not agree with ProB, e. g., Z3 reported unsatisfiability while ProB found a model.

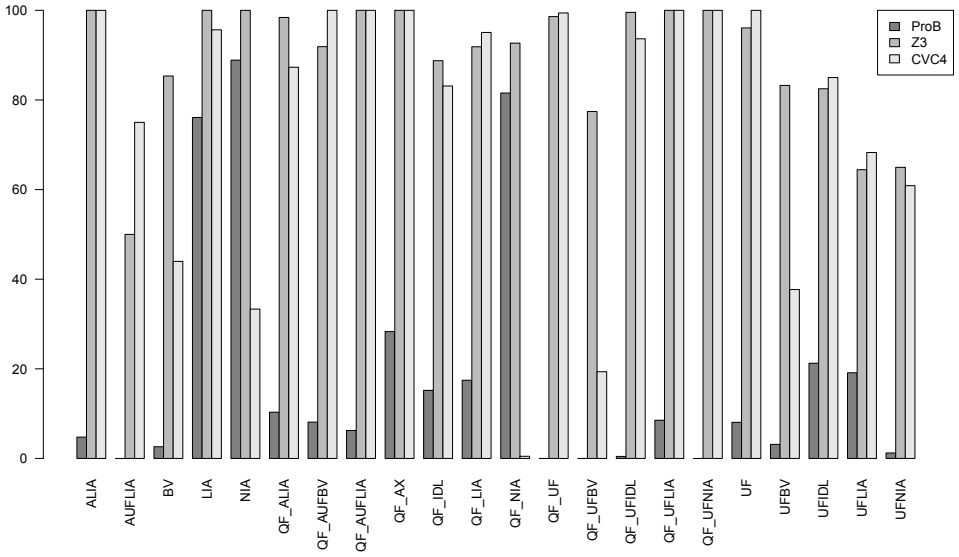


Figure 5.2: Solved Benchmarks per Logic in %

The most interesting bugs identified are:

- It took us several attempts to find a suitable translation of modulo and division, as the rounding behavior of B and SMT-LIB is different. First, we tried to add a correcting $+1$ or -1 to the result if necessary. As B has no if-then-else for expressions, this proved quite error prone. However, existing differences in the semantics were immediately discovered by the QF_LIA and QF_NIA benchmarks. We got rid of the bugs by using the axiomatization given above.

- Some rare edge cases of ProB's detection of exhaustive enumeration were not covered by our tests so far. In consequence, ProB was able to assume exhaustive enumeration and hence unsatisfiability even though not all combinations of variables have been tried out. The huge amount of test cases available in the SMT-LIB collection helped to thoroughly exercise ProB's kernel.

- Cross checking using both ProB and the SMT solvers CVC4 and Z3 to discharge Event-B proof obligations helped us discover a bug in ProB's kernel: In presence of certain combinations of constraints, a logical variable was not bound even though it should have been. The bug could be traced back to a bug in underlying CLP(FD) system of SICStus Prolog [42].[4]

Furthermore, several performance bottlenecks have been brought to our attention and have been resolved. In summary, making ProB available as a general purpose SMT solver has helped us to improve ProB itself.

[4]More precisely to the `element/3` constraint used for function applications.

Table 5.3: Benchmarks: Logics

solver	configuration	sat	unsat	unknown	timeout	memory out
Integer Arithmetic						
ProB	vanilla	8543	2696	14607	6984	0
ProB	random	8543	2696	14607	6984	0
ProB	cse	8543	2670	14633	6984	0
ProB	chr	8479	2647	14320	7384	0
Z3	default	12234	13918	1219	5459	0
CVC4	default	4027	13999	9322	5482	0
Uninterpreted Functions						
ProB	vanilla	85	2457	20637	1815	2
ProB	random	85	2457	20637	1815	2
ProB	cse	85	2431	20515	1963	2
ProB	chr	84	2443	20650	1817	2
Z3	default	3651	15684	811	4850	0
CVC4	default	3616	15994	39	5347	0
Bit Vectors						
ProB	vanilla	11	3	369	45	22
ProB	random	11	3	368	46	22
ProB	cse	11	3	368	46	22
ProB	chr	11	3	369	55	12
Z3	default	143	237	1	68	1
CVC4	default	55	144	67	184	0

5.5 Related Work

Of course there are several other solvers available that support the SMT-LIB language. The most prominent ones being Boolector [36], CVC4 [16], veriT [33], Yices [70] and Z3 [60].

There have been different approaches of solving bit vector constraints by translation into other languages. There is for instance a translation from quantifier-free bit vector formulas to effectively propositional logic (EPR), allowing to apply EPR solvers [112].

As in this chapter, solving bit vector logics by using a mixture of word level arithmetic and propositional logic has already been suggested in [188]. In order to stay closer to pure constraint logic programming, Bardin et al. [15] suggest a specialized CLP domain tailored towards bit-level reasoning. While this seems to be a promising approach, we did not intend to change the internals of ProB's

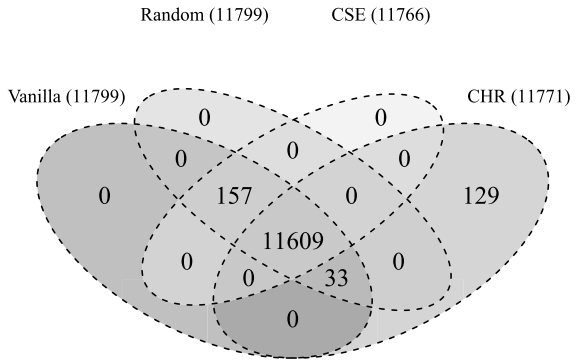


Figure 5.3: Performance Comparison: ProB Options

constraint solver itself. However, ProB can use bit-level reasoning using a Kodkod [177] backend, effectively translating B to SAT [160]. With the Kodkod backend available, we saw no need for another bit-level backend.

The idea to use model checkers to solve bit vector formulas is not new either. A translation to SMV is suggested in [83], using the NuSMV [47] symbolic model checker as solver. In contrast to this approach, we translate into the input language of an explicit state model checker rather than a symbolic model checker. However, as the B models generated by our translation do not include state transitions, this does not cause a considerable difference in behavior or performance.

Logic and Constraint Logic Programming techniques are applied to general SMT-LIB constraints in [101, 102] and [69]. In [101] the authors Howe and King present a SAT solver in Prolog as a programming pearl. The solver is quite small and easy to understand and extend, showing how certain Prolog techniques lead to an efficient implementation. An extension to SMT has been published in [102]. In [69] an SMT solver implemented in CHR [81] is presented. It features an input language comparable to the SMT-LIB language. However, it cannot be applied to SMT-LIB files directly.

In [71, 72] the authors present a translation from Event-B proof obligations to the SMT-LIB format in order to use SMT solvers to discharge them. This is a translation in the opposite direction. Sadly, it proves hard to fully provide B's and Event-B's axioms of set theory to the underlying solvers. To do so, the authors suggest what they call the *ppTrans* approach. Essentially, set theory and arithmetic are broken down into first-order formulas using uninterpreted functions for membership, etc. The resulting constraint is enriched by certain set

theoretic axioms supposed to convey background knowledge about Event-B to the SMT solvers. As a result, encoded formulas only approximate the Event-B semantics.

An approach comparable to that of [177] and Chapter 4 has been followed in the Isabelle community as well. In [29] the authors translate Isabelle/HOL to first-order relational logic. Afterwards Kodkod is used, translating to SAT and solving using a SAT solver. Sledgehammer, a tool used to discharge proof obligations in interactive proofs done using Isabelle, is connected to SMT solvers in [28]. The authors report a considerable increase in automatically discharged proof obligations.

A higher level approach towards embedding B and Event-B into SMT-LIB will be evaluated in Chapter 6. Instead of relying on a first-order encoding, we directly employ Z3's own set theory solver. Thus, we only have to supply axiomatic definitions of operators unavailable in SMT-LIB, like for example the cardinality of a set.

The empirical comparison performed in Chapter 6 will show that both approaches have their merits and none of them is strictly superior. In particular, Z3 was good at detecting inconsistent predicates but not good at finding solutions (aka models). This finding is also confirmed in one of our projects [167], where we experimented with various encodings and solvers for university timetabling.

5.6 Conclusion and Future Work

In the future, we would like to further investigate ProB's performance in comparison to state-of-the-art SMT solvers. We hope to gain a deeper understanding on kinds of SMT constraints and how they relate to the performance of CLP(FD) and DPLL($\mathcal{T}$).

In order to perform a more in-depth comparison, we would like to extend our translation and ProB's constraint solving kernel to support incremental solving. This would enable us to use larger industrial case studies for our evaluation.

We would also like to further evaluate the effectiveness of our translation, considering different representations of SMT-LIB expressions in B. While we have done so for smaller collections of benchmarks, we would like to compare encodings more thoroughly throughout the whole SMT-LIB collection.

In particular, we suspect that the translation of bit vector operators as set comprehensions is not optimal and one of the reasons for ProB's lack of performance. There are different courses of action we can take in this regard. On the one hand, we can try to improve ProB's propagation and solving capabilities on set comprehensions. This would certainly benefit other use cases as well

and fit with the high-level approach of B. On the other hand, a more low-level translation using quantification over the elements of a bit vector might provide better propagation of intermediate values.

Another aspect that could be considered in a later evaluation is the time we allow the solvers to run. It would be interesting to see whether the timeout influences the solvers differently.

Despite the rather low number of test cases successfully solved by ProB, we still believe a translation from SMT-LIB to B is beneficial:

- It allows for cross-checking of results using a different approach to solving.
- It enables using the large library of SMT-LIB examples to validate and improve B tools.
- Using Atelier B or Rodin it provides interactive proof assisted by different solvers. As far as we know, there is currently no other way to interactively tackle SMT-LIB constraints.
- ProB already provides reasoning over sets and strings, features just starting to find their way into SMT-LIB.

Summarizing, we have presented a translation from SMT-LIB to B that allows all tools available for the B method to be employed on SMT-LIB files. Among Atelier B and ProB which we have already discussed, there are other model checkers such as pyB [185] and eboc [141], both using different approaches to constraint solving. A new platform for automatic proof is developed in the BWare [65] project.

We used our translation to extend the constraint solving kernel of ProB to SMT-LIB constraints, making it available in a more general context. However, our evaluation showed that it is competitive only for certain types of benchmarks.

As expected, the CLP(FD) based approach performs well for satisfiable benchmarks. Furthermore, it generates a model for every formula it detects satisfiable. Unsatisfiable formulas are the main weakness of our approach.

In future work, we want to evaluate how a combination of ProB with the Atelier B provers performs as a solver portfolio. However, as classical rule based provers, the Atelier B provers are not designed to report models for satisfiable formulas. An efficient integration with ProB into a combined solver is thus more complex than just running the tools in parallel. For the combined solver to be of use, we have to enable ProB to identify predicates that it should submit to the provers and make it react to the two possible outcomes “proof” and “unknown”.

The empirical evaluation showed that the enumeration strategies employed by ProB are not fully suitable for solving SMT-LIB. In future work, we want to enrich our constraint solver by further enumeration strategies. This will not only

aid the SMT solver but will strengthen the B and Event-B model checkers as well.

Along with tuning the enumeration strategy we would like to deepen our understanding of the effectiveness of the additional propagation rules and how they interact with the CLP(FD) core. The evaluation shows that the added rule sets make certain benchmarks solvable while they decrease the performance on others. As of now, we have no proper way of deciding which rules to enable upfront.

Performance aside, using ProB to solve benchmarks taken from the SMT-LIB collection helped us discover different errors and inconsistencies in ProB itself. In particular, SMT-LIB benchmarks tend to exercise different parts of ProB's kernel than classical B machines due to the diverse usage of constraints.

Furthermore, we laid the groundwork for an analysis or proof of SMT solving algorithms using the B method by embedding SMT-LIB in B or Event-B. We were able to mimic the semantics of SMT-LIB in B, including constructs like if-then-else and bit vectors. Aside from ProB, the translation allows further B method tools like Atelier B to examine SMT-LIB data structures, expressions and algorithms. We have performed a case study showing how to prove SMT propagation laws using the B method. Hence, our translation could help reasoning about solvers and solving procedures in a structured way.

Last, we hope that our work is able to narrow the gap between the SMT solving, the constraint logic programming and the formal methods communities and eases mutual understanding of algorithms and design principles.

Coming together is a beginning;
keeping together is progress;
working together is success.

Henry Ford

6

Integrating SMT Solvers and CLP(FD) in ProB

In this chapter we will show how the SMT solvers Z3 and CVC4 can be integrated with ProB's CLP(FD) solver into a single solving procedure. We will again use proof obligations to assert if the integration is able to overcome the limitations discovered in Chapters 4 and 5.

The chapter is an extended version of our paper "SMT Solvers for Validation of B and Event-B models" [117]. For information regarding authors and their individual contributions see Appendix C. Extensions include using CVC4 in addition to Z3, including benchmarks comparing the two SMT solvers. Additionally, translation rules are discussed in greater detail. In particular, several edge cases omitted in [117] are taken into account.

6.1 Introduction and Motivation

Originally, the ProB kernel has been tailored towards satisfiable formulas, acting primarily as a model finder [128, 127]. This fact has been made obvious in the two empirical evaluations in Chapters 4 and 5 as well. As our benchmarks have shown, ProB still fares better if given constraints are satisfiable. However, when targeting symbolic model checking, the detection of unsatisfiability is of equal importance. Yet, the additions made to CLP(FD) in Chapter 3 have proven to be too weak.

As stated in Section 2.3.1, ProB's approach to constraint solving is based on constraint logic programming. Thus, the solving kernel works fundamentally different from the DPLL($\mathcal{T}$) [84] approach employed by modern SMT solvers like CVC4 [16] or Z3 [60]. In Chapter 4 we already compared both approaches and outlined that neither is able to outperform the other: there is a considerable number of proof obligations that can only be solved by one of them. Hence, our idea is to combine their particular strengths into a single solving procedure.

First, in Section 6.1.1 we will show some examples for strengths and weaknesses and argue towards our integrated approach. How we translate B and Event-B constraints into SMT-LIB problems is explained in Section 6.2. ProB and the SMT solvers CVC4 and Z3 are combined as outlined in Section 6.3. In particular, we focus on communication of inferred predicates. We discuss limitations of our approach in Section 6.4, and substantiate our discussion with empirical evaluation in Section 6.5. We conclude with related work in Section 6.6 and future work and conclusions in Section 6.7.

6.1.1 Small Experiments

To outline some of the weaknesses of the CLP(FD) based solving kernel, have a look at the following predicate: $X > 3 \wedge X < 7 \wedge X < Y \wedge Y < X$. Classic CLP(FD) style domain propagation first sets up the domains $4 \mathinner{..} 6$ for X and $-\infty \mathinner{..} \infty$ for Y. In a second step, all values that cannot be part of a solution are removed from the domains. Both domains end up being empty. Hence, the predicate is detected as unsatisfiable.

As soon as we drop one of the constraints on X, CLP(FD) is unable to do so and has to resort to enumeration. For example, the predicate $X < Y \wedge Y < X$ cannot be proven unsatisfiable by ProB's CLP(FD) kernel alone, as both domains for X and Y are infinite ($-\infty \mathinner{..} \infty$). Similarly, $X < 7 \wedge X < Y \wedge Y < X$ leads to an infinite sequence of narrowed down domains, never reaching inconsistency. SMT solvers on the other hand easily detect the unsatisfiability.

The CLP(FD) based solver in ProB however can handle certain higher-order constructs like set comprehensions better than the SMT solvers: look for example at the predicate $(2 \mapsto 4) \in \{y \mid \exists(x).(y = (x \mapsto x + 2))\}$. It states that the pair $(2 \mapsto 4)$ is a member of the set of all pairs y that are of the form $(x, x + 2)$. ProB reports satifiability and returns an empty valuation as there are now free variables.

Of course the performance shown by the SMT solvers highly depends on the translation. Choosing a low-level translation, the predicate can be broken down to $4 = 2 + 2$ and solved by CVC4 as well as Z3. If we stay on the high-level of set logics, the set comprehension has to be described using universal quantification.

If translated this way, Z3 runs into a timeout; CVC4 does not fully support quantifiers in combination with set theory as of yet [14].

Additionally, the CLP(FD) based solver performs better for model finding tasks that involve non-linear integer constraints. As an example, take the verbal arithmetic puzzle to find (non-equal) digits K, I, S, P, A, O, N such that $KISS * KISS = PASSION$. In B this can be written as $(1000 * K + 100 * I + 10 * S + S) * (1000 * K + 100 * I + 10 * S + S) = 1000000 * P + 100000 * A + 10000 * S + 1000 * S + 100 * I + 10 * O + N$.

As each letter should represent a single digit, constraints like $0 \leq K \leq 9$ are added for all the variables. Finally, we add pairwise disequalities for all variables. The resulting predicate is solved by ProB in milliseconds, while both CVC4 and Z3 answer unknown.

Obviously, both approaches could benefit from each other. In the following sections we suggest a possible integration between the CLP(FD) and SMT approaches, trying to gain the advantages of both.

6.2 High-Level Translation of B to Z3

The following section will explain both our translation from B to SMT-LIB and how we integrated the SMT solvers CVC4 and Z3 into ProB in order to solve constraints given in B or Event-B. First, in Section 6.2.1 we outline a normal form for B that avoids certain constructs that are hard to translate.

Primarily, this is achieved by replacing several expressions by equivalent ones using different operators. Following, in Section 6.2.2 we translate constraints given in normalized B into the (set-)logics of CVC4 and Z3. Lastly, Section 6.3 explains how ProB's kernel and the SMT solver are integrated in order to combine both solvers.

6.2.1 Normalizing B / Event-B

B and Event-B feature many constructs not directly available in the SMT solvers' input language SMT-LIB. In preparation of the translation from B to SMT in Section 6.2.2, we use rewrite rules to transform a B predicate into a normal form that is easier to translate. All these transformation rules are meant to be applied repeatedly until a fixpoint is reached.

In a first step, we replace certain negated operators available in B by the negation of the regular operator. For instance, we replace $x \notin y$ by $\neg(x \in y)$. In addition, we have to rewrite set operations involving strict subsets to subsets and (dis-)equalities. See Table 6.1 for the operators and their translations.

Table 6.1: Normalization of Operators

B	Normalized B
$E \neq S$	$\neg(E = S)$
$E \notin S$	$\neg(E \in S)$
$E \not\subset S$	$\neg(E \subset S)$
$E \not\subseteq S$	$\neg(E \subseteq S)$
$E \subset S$	$\neg(E = S) \wedge (E \subseteq S)$

Currently, the set logics of SMT solvers have no direct support for intervals or the bounded B integer sets `NAT, NAT1, INT`. The same holds true for the unbounded sets `NATURAL` and `INTEGER`. We thus rewrite constraints featuring membership in one of these to a conjunction of inequalities, e. g.,

$$x \in 1..5 \Leftrightarrow 1 \leq x \wedge x \leq 5.$$

In case of `NATURAL` we just set up a lower bound. Membership in `INTEGER` is merely typing and does not pose any restrictions on the variable. The constraint can thus be removed after type checking.

Membership in unions, intersections or set differences of these are handled by decomposing into multiple conjuncts or disjuncts respectively, e. g.,

$$x \in -2..5 \cap \texttt{NAT} \Leftrightarrow (-2 \leq x \wedge x \leq 5) \wedge (0 \leq x \wedge x \leq \texttt{MAXINT}).$$

If one of these sets has to be set up explicitly, e. g., in `nat = NATURAL`, we have to rely on set comprehensions. That is we define `NATURAL` as $\{x \mid x \geq 0\}$ and `INTEGER` as $\{x \mid x \geq 0 \vee x < 0\}$.

Z3's API allows to define the full set of a given type. Thus, with Z3 we can avoid translating `INTEGER` into a set comprehension. For CVC4 we have to rely on quantifiers which are not supported in conjunction with logics featuring sets. Furthermore, CVC4 only supports finite sets as of yet [40].

As an example for these limitations, take the simple constraint $\mathit{naturals} = \mathbb{N} \wedge \mathit{negative} < 0 \wedge \mathit{negative} \in \mathit{naturals}$. It is encoded in SMT-LIB in Listing 6.1. As stated, it cannot be solved by CVC4, but it can be solved by Z3. As the benchmarks in Section 6.5 will show, this limits CVC4's performance when used as a backend for ProB.

ProB represents relations and functions as sets of tuples. Usually, the set is computed exhaustively. For certain relations or functions, e. g., infinite ones, ProB tries to keep the set symbolic. Furthermore, B allows set theoretic operators to be applied to functions as well. For these two reasons, we cannot simply express B functions as uninterpreted functions in SMT-LIB.

Listing 6.1: SMT-LIB Encoding of Constraint using `NATURAL`

```
(declare-fun naturals () (Set Int))
(declare-fun negative () Int)

(assert (forall ((x Int)) (=> (>= x 0)
                              (member x naturals))))
(assert (forall ((x Int)) (=> (< x 0)
                              (not (member x naturals)))))

(assert (< negative 0))
(assert (member negative naturals))
```

Instead, we represent functions in SMT-LIB the same way we do in ProB. This makes it necessary to rewrite some B expressions on functions. For instance, we rewrite the function application using a temporary variable:

$$f = \{(1 \mapsto 4), (2 \mapsto 2)\} \land x = f(1)$$

becomes

$$f = \{(1 \mapsto 4), (2 \mapsto 2)\} \land (1 \mapsto x) \in f.$$

During normalization, we have to keep in mind that well-definedness conditions of a predicate might change. In the given examples, if we request the function value of f at 3, the predicate is not well-defined:

$$f = \{(1 \mapsto 4), (2 \mapsto 2)\} \land x = f(3)$$

We have applied the function f outside of its domain. In contrast,

$$f = \{(1 \mapsto 4), (2 \mapsto 2)\} \land (3 \mapsto x) \in f.$$

is well-defined and evaluates to false. In several cases, we add well-defined conditions later on. We show an example, division, in Section 6.2.2. Note that Rodin creates a separate proof obligation for well-definedness. Hence, one can assume well-definedness to be handled by those proof obligations.

Several other operators like domain (restriction) or range (restriction) can be rewritten to set comprehensions. For example, the following equality holds for the range of a function f:

$$\text{ran}(f) = \{y \mid \exists x.(x \mapsto y) \in f\}.$$

More definitions of B and Event-B operators in terms of set comprehension can be found in the "reference" books on B [1, 2, 168].

B features record datatypes comparable to those supported by CVC4 and Z3. However, when using the SMT solvers, record types have to be introduced and

typed before constraints can be applied to the fields. In normalized B, the declaration of a constrained record is hence split in the declaration of a general record conjoined with a predicate constraining the fields. A record membership expression like

$$r \in \mathit{struct}(f \in 11..20, g \in 12..30)$$

becomes

$$\underbrace{r \in \mathit{struct}(f \in \mathbb{Z}, g \in \mathbb{Z})}_{\text{record type}} \wedge \underbrace{r'f \geq 11 \wedge 20 \geq r'f \wedge r'g \geq 12 \wedge 30 \geq r'g}_{\text{restrictions on fields}},$$

where the $'$ operator is used to fetch the value of a record's field.

Some functions included in B, like the two arithmetic functions min and max, are not directly available in SMT-LIB. Directly referring to the cardinality of a set is only possible in CVC4 at the price of dropping support for infinite sets.

We hence add temporary variables and supply certain axioms as we did to encode function application. For instance, the expression $\min(S)$ is replaced by a variable t and the following additional constraints are added:

- $\forall\, m.m \in S \Rightarrow t \leq m$, i. e., the temporary variable is less or equal to all members of the set.
- $\exists\, m.m \in S \wedge t = m$, i. e., t is equal to one of the members of S.

We encode max using the same pattern. For the cardinality, we add a constraint stating that c is the cardinality of S if there exists a bijection between the interval $1..c$ and S. For the empty set, this holds for any $c \leq 0$. Hence, we add $c \geq 0$, resulting in $\mathrm{card}(\varnothing) = 0$.

The choice of axioms supplied to the SMT solvers in order to define the B functions influences the performance. We could provide more properties of max, e. g.,

$$\max(S1) > \max(S2) \Rightarrow \forall\, c.c \in S2 \wedge \exists\, s.s \in S1 \wedge s > c.$$

Additional axioms might aid in detecting unsatisfiable predicates. However, they might also decrease performance as they have to be considered during reasoning.

The rules above transform a B predicate into an equivalent B predicate. However, we could go even further, depending on how we employ the SMT solvers: For animation and (explicit state) model checking, we have to use an equivalent formula, as we rely on the models.

In contrast, for certain symbolic model checking algorithms or proof attempts, we could use rewriting rules that transform a B predicate into an equisatisfiable predicate. The added freedom could be used to tailor the formula towards the solvers' strengths. We will address this in future work.

While nearly all complicated B constructs can be rewritten to set comprehensions, not all resulting predicates can be solved by CVC4 or Z3. So far, we have not found an efficient translation of the following operators:

- The *general union*, *general intersection*, *general sum* and *general product*. For instance, the general union of $U \in \mathbb{P}(\mathbb{P}(S))$ could be rewritten as $\mathit{union}(U) = \{x \mid x \in S \land (\exists\, s.s \in U \land x \in s)\}$. However, the existential quantification inside the set comprehension leads to highly involved constraints later on. This often results in timeouts, especially when trying to find models for satisfiable formulas.
- The construction of (non-empty) powersets. Again we could translate $\mathbb{P}(X) = \{s \mid s \subseteq X\}$.
- The iteration and closure operators of classical B.

6.2.2 Translation Rules

We feed the normalized constraints generated in the previous section into the C / C++ APIs of CVC4 and Z3. In particular, we use logics including support for sets. Z3 realizes those using the techniques described in [61], for CVC4 see [14]. The set theories differ in the availability of constraints such as cardinality and in the support of infinite sets.

Any logic including integer arithmetic, sets and quantifiers already covers most of the expressions occurring in our normalized constraints. Thus, we can pass most of the constraints unmodified. There are however some exceptions:

- Some common operators have different semantics in B and SMT-LIB.
- SMT-LIB does not support set comprehensions natively. We will translate those by using a universal quantification constraining all members of a set variable.
- User-given sets have to be mapped to SMT-LIB sorts. As use-given sets are disjoint in B and Event-B, we can translate each set to an individual sort.

For an approach that is based on translation to be both sound and complete we have to ensure that semantical differences are taken into account. In particular, B features a distinct concept of well-definedness, i. e., operators may only be applied under certain conditions. This contrasts with SMT-LIB treating operators as total functions that always return a result. Additionally, the results of applying certain operators differ as well.

Integer division is a prominent example: B uses a division that rounds towards zero. In contrast, SMT-LIB semantics define a division rounding towards $-\infty$.

Furthermore, B does not allow division by zero while for SMT solvers division is a total function, e. g., for the predicate $x = 1/0$ the SMT solvers return the solution $x = 0$. In order to overcome these differences, we set up $x = a/b$ using SMT-LIB's if-then-else as

$$x = \text{ite}(a > 0, a/b, \text{ite}(b > 0, (a/b) + 1, (a/b) - 1)) \wedge b \neq 0.$$

Note that we have to insert the additional well-definedness constraint $b \neq 0$ appropriatelly, such as to avoid issues due to negation. The predicate $x \neq a/b$ features the same well-definedness condition as $x = a/b$ does. In consequence, $x \neq a/b$ has to be translated as

$$\neg(x = \text{ite}(a > 0, a/b, \text{ite}(b > 0, (a/b) + 1, (a/b) - 1))) \wedge b \neq 0$$

rather than

$$\neg(x = \text{ite}(a > 0, a/b, \text{ite}(b > 0, (a/b) + 1, (a/b) - 1)) \wedge b \neq 0).$$

Now, let us have a look at the translation of set comprehensions. A B expression like

$$\neg(r \in \{x \mid x \bmod 2 = 1\})$$

is submitted to the SMT solvers using a temporary variable and axiomatizing the set comprehension. The resulting constraint is

$$\neg(\exists\, tmp.(r \in tmp \wedge \forall\, v.v \in tmp \Leftrightarrow v \bmod 2 = 1)).$$

So far we do not provide any additional hints like instantiation triggers [151, 68]. By doing so, we could provide rules for when and how to instantiate quantified variables.

In addition to given types likes `INTEGER`, the B method features user-defined types represented as deferred or enumerated sets. We translate those to custom SMT-LIB sorts. For enumerated sets we additionally introduce the identifiers and enforce their disequality using an additional constraint.

There is one crucial difference between sorts and B's deferred sets however. The newly created sorts representing deferred sets are infinitely large. The deferred set itself can be instantiated to different sets containing an arbitrary, possibly infinite, amount of elements.

To bridge the gap, we introduce an intermediate identifier used to refer to the full deferred set. The intermediate identifier is typed as a set containing elements of the corresponding sort.

For an example of the translation of given sets, have a look at Listings 6.2 and 6.3. In the B machine in Listing 6.2, we introduce a deferred set `D` and an enumerated set `E`. As outlined above, `D` is represented in Listing 6.3 by the newly

Listing 6.2: B Machine with Given Sets

```
MACHINE GivenSets
SETS D;E = {e1,e2}
END
```

Listing 6.3: SMT-LIB Encoding of Given Sets

```
(declare-sort D 0)
(declare-fun D () (Set D))

(declare-sort E 0)
(declare-fun E () (Set E))
(declare-fun e1 () E)
(declare-fun e2 () E)

(assert (= E (insert e1 (singleton e2))))
(assert (distinct e1 e2))
```

declared sort `D` when it comes to typing variables. The set itself can be accessed using the nullary function called `D` as well.

For the enumerated set `E = {e1,e2}`, we introduce three nullary functions, one for each element and one for the set itself. Note that we do not limit the cardinality of the sort `E` that way and thus have to account for it later. Otherwise, a constraint like $other \neq e1 \wedge other \neq e2$, where *other* is of type E, is satisfiable by creating a third element.

Sadly, only Z3 natively supports sorts with given cardinality. Hence, if the cardinality of a user-given type can be computed statically by ProB, we can submit said cardinality to Z3. CVC4 does not support constraining the cardinality of user-defined sorts. We thus use an encoding comparable to that of the cardinality constraint.

6.3 Integration of Solvers

We investigated different modes of using SMT solvers together with the ProB kernel:

- Use it alone without relying on ProB. This approach was quickly abandoned due to the (currently) untranslatable predicates outlined in Sections 6.2.1 and 6.2.2. Additionally, some translations have to resort to quantification that hinders proof efforts and model finding.
- Use CVC4 or Z3 solely for falsification of B predicates. If we only rely on the SMT solvers for the detection of unsatisfiability, we can safely skip untranslatable parts of the predicate without risking unsound results (as those parts will be checked by ProB's solver). However, many predicates cannot be disproven once important parts are missing.
- We could employ a cooperative approach where parts of a predicate are given to one or both of the SMT solvers, while other parts are handled by the ProB kernel. In this case, we would translate partial assignments back and forth between the solvers to communicate intermediate results.
- Lastly, we could use a fully integrated approach where the whole predicate is given to the ProB kernel and as much as is translatable is given to the SMT solvers. In addition to partial assignments we could transport information about inferred or learned clauses or unsatisfiable cores back and forth.

The first approach was quickly discarded, because the SMT solvers alone are often too weak to solve interesting predicates. This is mostly due to cumbersome translations of higher-order B expressions such as set cardinality. The same holds true for the second approach. Even though the SMT solvers are able to falsify several predicates that ProB cannot falsify (see Section 6.1.1), much is left to be desired. Hence, we investigated the integrated approaches more thoroughly.

The third approach is comparable to the one taken in [160], translating B to SAT. The key problem to this approach is to decide which predicates to translate and submit and which ones to keep in ProB. In [160] the authors used a greedy approach: every predicate that can be translate will be translated.

In contrast, we integrated the different solvers further and set up constraints in both simultaneously. We delay the call to the SMT solvers until after the deterministic propagation phase of ProB and also submit the information inferred so far. Additionally, we use the unsat core found by one of the SMT solvers to control backtracking on the Prolog side and to lift ProB from backtracking to backjumping. Details on both techniques are given below.

Algorithm 6.1: Integrated Constraint Solver

```
   Data: Predicate P, (partial) State S
   Result: backjump iff P is unsat, model iff P is sat; might time out
1  procedure boolean solve(P, S)
2      set_up_clpfd_variables(S)
3      set_up_smt_variables(S)
4      while exists conjunct C in P that has not been set up do
5          D = to_clpfd_solver(C)              // domains D from clpfd
            propagation
6          smt_result = to_smt_solver(C,D)          // transfer C and
            domains
7          if smt_result = unsat then
8              backjump using unsat core
9          end if
10     end while
11     while exists unbound variable V in S do
12         clpfd_labeling(V)                          // binds V to value
13         smt_result = to_smt_solver(V)  // V now bound:  transfer
            new value
14         if smt_result = unsat then
15             backjump using unsat core
16         end if
17     end while
18     return S with all variables labeled
```

Transferring CLP(FD) Domains to the SMT Solvers

As can be seen in Algorithm 6.1, communication with the SMT solver starts after the deterministic propagation phase. During this phase, ProB tries to deterministically infer knowledge about the values of the variables in a predicate using the consistency algorithms discussed in Section 2.4. For instance, from $X > 3 \wedge Y > X$ ProB infers $Y > 5$. The underlying propagation rules are not limited to arithmetic but support further B constructs like set theory. Before a predicate is submitted to the SMT solvers, all the statically inferred information is added to it.

Controlled Backjumping Using the Unsat Core

In case unsatisfiability is detected by CVC4 or Z3, we can use the SMT solvers' unsat core computation in order to perform backjumping inside ProB's kernel. The unsat core contains a subset of the conjuncts C taken from P as outlined in

Algorithm 6.1. Note that this subset does not necessarily contain the conjunct submitted last.

Inside ProB's kernel we can now backjump until at least one of the conjuncts inside the unsat core has been removed from both the SMT solver and the CLP(FD) solver. After the backjump, ProB can choose a different path inside case distinctions or decide on different heuristics. Thus, the backjump has cut off parts of the search space ProB would have explored otherwise.

In addition to what is shown in Algorithm 6.1, we could analyze the SMT solvers' knowledge about the causes of a conflict. In particular, we could transfer parts of the conflict clauses or the unsat core back to ProB to avoid problematic instantiations of variables. Used this way, CVC4 and Z3 would provide clause learning capabilities to ProB, a feature typical CLP(FD) implementations do not provide.

However, the approach would need an improved way of translating the SMT representation of a constraint back to the B representation. While we have laid the foundations of such a translation in Chapter 5, our approach is not yet potent enough, as translating a predicate from B to SMT-LIB and back again often leads to a blowup of the predicate's complexity. This is even more limiting if we consider that predicates might pass the bridge between ProB and the SMT solvers multiple times before a final solution is found.

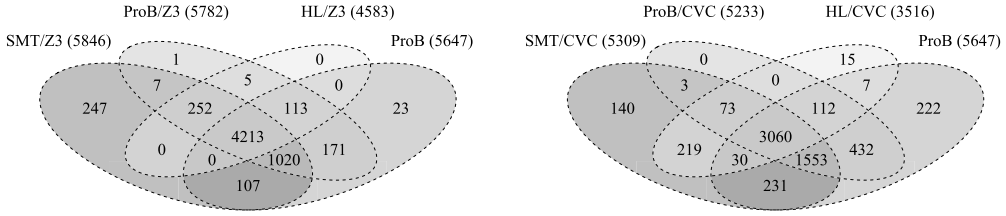
6.4 Limitations

One key limitation of our approach is related to the type system of B. There is no strict differentiation between functions, sets and sequences. For instance, one can apply the set union operator to two functions leading to a result that might not be a function.

For the same to be allowed in the SMT-LIB translation, we had to use a common representation: we express relations and functions as sets of pairs connecting input and output values; sequences are encoded as sets of pairs consisting of the sequence index and the value.

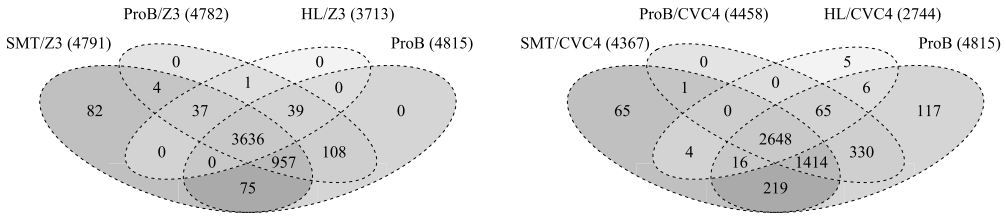
Using this common base representation, all B and Event-B operators can be encoded. However, we cannot use more sophisticated SMT-LIB representations anymore. In particular, sequences could have been mapped to SMT-LIB arrays, resulting in improved performance due to the usage of specialized decision procedures.

Another limitation is the missing support for set cardinality in Z3's set logic. Although it was part of the initial proposal for the SMT-LIB finite set theory [183]



(a) Z3 (b) CVC4

Figure 6.1: Performance on Proof Obligations



(a) Z3 (b) CVC4

Figure 6.2: Performance on Proof Obligations: Landing Gear Systems

it has not yet been implemented in Z3. We thus encode $c = \text{card}(S)$ as

$$\exists\, t.t \in S \rightarrowtail [1, c]$$

i. e., we search for a total bijection from S to the interval $[1, c]$. This encoding is quite cumbersome and often leads to Z3 answering "unknown". Cardinality constraints however have to be used in the translation of some B operators, e. g., to compute the next index of a sequence upon concatenation. Hence, those can often not be solved as well.

6.5 Empirical Evaluation

In this section we will evaluate two different aspects. First, we want to know how our high-level translation of set theory compares to the more low-level approach of the SMT translation outlined in [71, 72].

Second, we want to evaluate if it is worthwhile to integrate SMT solvers into ProB and to communicate back and forth. In order to find out, we compare the integrated solution to both CVC4, Z3 and ProB on their own. Additionally, we compare CVC4 and Z3 to assert which one to use in ProB's new backend in Section 6.5.3.

Table 6.2: Benchmark Results of Z3-based Provers

Model	# POs	SMT/Z3	HL/Z3		ProB		ProB/Z3	
			prove	disprove	prove	disprove	prove	disprove
Landing Gear System 1, Su, et al.	2328	2318	2206	0	2310	0	2319	0
Landing Gear System 2, Su, et al.	1188	1169	987	0	1176	0	1184	0
Landing Gear System 3, Su, et al.	341	317	137	0	289	0	287	0
CAN Bus, Colley	534	501	403	0	282	2	433	2
Graph Coloring, Andriamiarina, et al.	269	152	66	0	1	0	68	0
Landing Gear System, Hansen, et al.	74	63	57	0	74	0	74	0
Landing Gear System, Mammar, et al.	433	430	242	0	412	0	364	0
Landing Gear System, André, et al.	619	494	84	0	554	5	554	5
Pacemaker, Neeraj Kumar Singh	370	296	369	0	365	0	370	0
Stuttgart 21 interlocking, Wiegard	202	106	32	0	184	2	129	2

Table 6.3: Benchmark Results of CVC4-based Provers

Model	# POs	SMT/CVC4	HL/CVC4		ProB		ProB/CVC4	
			prove	disprove	prove	disprove	prove	disprove
Landing Gear System 1, Su, et al.	2328	2318	2206	0	2310	0	2319	0
Landing Gear System 2, Su, et al.	1188	1169	987	0	1176	0	1184	0
Landing Gear System 3, Su, et al.	341	317	137	0	289	0	287	0
CAN Bus, Colley	534	501	403	0	282	2	433	2
Graph Coloring, Andriamiarina, et al.	269	152	66	0	1	0	68	0
Landing Gear System, Hansen, et al.	74	63	57	0	74	0	74	0
Landing Gear System, Mammar, et al.	433	430	242	0	412	0	364	0
Landing Gear System, André, et al.	619	494	84	0	554	5	554	5
Pacemaker, Neeraj Kumar Singh	370	296	369	0	365	0	370	0
Stuttgart 21 interlocking, Wiegard	202	106	32	0	184	2	129	2

6.5.1 Experimental Setup

For the benchmarks, we have used Rodin 3.2 and version 1.3.0 of the SMT plugin. For better comparability, we did not use the bundled SMT solvers this time. Instead, we relied on Z3 version 4.4.1 as used in the ProB integration as well and CVC4 1.4, again as used inside ProB. ProB was used in version 1.6.1-beta1, connected through the disprover plugin version 3.0.9.

We used a global timeout of 5 seconds for a single solving attempt. As discussed in Chapter 4, all solvers based on ProB's disprover might use multiple solving attempts as shown in Fig. 4.1.

All benchmarks were run on a MacBook Pro featuring a 2.6 GHz i7 CPU and 8 GB of RAM. We did not parallelize the benchmarks in order to avoid issues due to hyper-threading or scheduling. Benchmarks were run using the evaluation plugin for Rodin which we already employed in Section 4.3.

Benchmarked models, hardware, versions and timeout settings are identical to those used in Section 4.3. In consequence, the empirical results of Chapter 4 can be compared to the ones below.

We benchmarked the following configurations:

- **SMT/Z3**, the SMT solvers plugin for Rodin as presented in [71, 72], using Z3 as the backend prover,
- **SMT/CVC4**, the SMT solvers plugin using CVC4 as the backend prover,
- **HL/Z3**, our high-level translation from Event-B to SMT featuring Z3's set theory, alone without ProB's solver,
- **HL/CVC4**, the same configuration but using CVC4 instead of Z3,
- **ProB**, a plain version of ProB's constraint solving kernel,
- **ProB/Z3**, ProB's constraint solving kernel integrated with Z3, and
- **ProB/CVC4**, ProB integrated with CVC4.

For better comparability, we used the same set of benchmarks already employed in Chapter 4:

- Answers to the ABZ-2014 landing gear case study [31]. Beside our own version [92], we also used the three models by Su and Abrial [175], a model by André, Attiogbé and Lanoix [7], as well as a model by Mammar and Laleau [138].
- A model of the Stuttgart 21 Railway station interlocking by Wiegard, derived from chapter 17 of [2] with added timing and performance modeling.
- A model of a controller area network (CAN) bus developed by Colley.
- A formal development of a graph coloring algorithm by Andriamiarina and Méry. The graphs to be colored are finite, but unbounded and not fixed in the model.
- A model of a pacemaker by Méry and Singh [146].

6.5.2 Results

The data is presented as follows:

- Figure 6.1 shows two Venn diagrams comparing the number of discharged proof obligations by each of the configurations mentioned above. The diagram in Fig. 6.1a compares Z3 to ProB and the SMT solvers plugin, while Fig. 6.1b does so for CVC4.
- For comparability with Chapter 4, Figs. 6.2a and 6.2b show two Venn diagrams based only on the landing gear models. As in Chapter 4, the results of individual landing gears are given in Appendix A.2.

- Table 6.2 shows how the individual Z3-based configurations perform on the different models. In particular it distinguishes between proof and disprove.
- Table 6.3 does the same for the CVC4-backed provers.
- Table 6.4 shows how the individual configurations based on Z3 perform on different kinds of proof obligations.
- Table 6.5 does the same for the CVC4-backed provers.

Regarding the different performance of the high-level vs. the low-level SMT translation we have mixed results. Judging by the total numbers, in contrast to the benchmarks in Chapter 4, the low-level approach based on Z3 is superior: as can be seen in Fig. 6.1a, it is able to discharge 5846 proof obligations, while the high-level approach only discharges 4583.

Figure 6.1b shows that the difference between the high-level and the low-level translation is considerably larger when it comes to CVC4. The low-level approach based on CVC4 also outperforms the high-level approach by a margin. It discharges 5309 proof obligations, while the high-level one can only handle 3516.

This underlines the suspicion we set up in Section 4.3 that the introduction of CVC4 and Z3 directly into the SMT solver plugin leads to a performance increase. However, there is a considerable amount of proof obligations that can be discharged with the low-level approaches but not with the high-level ones and vice-versa.

Since the original SMT plugin does not support disproving POs, we cannot say anything about the performance. The high-level approaches based on CVC4 and Z3 are unable to disprove a single of the defective obligations. ProB's kernel remains superior when it comes to finding valuations for B predicates: The integrated solver and ProB alone identify all 9 known defective obligations.

Comparing ProB solo and together with Z3 paints a similar picture. The integrated solution is superior but the margin is small. Again, 130 proof obligations cannot be discharged anymore once the SMT integration is enabled. Virtually all of these result in a timeout afterwards. Since a global timeout is used and Z3 takes up too much time, ProB misses the solution. We could indeed use a local timeout for the integrated SMT solver. So far, we have not yet found a sensible heuristic to decide when to give time to Z3 vs. giving it to the ProB kernel.

Regarding disproving, integrating Z3 into ProB led to the discovery of a new counterexample not yet discovered in our former work [117]. Despite our usage of the CAN Bus model in Chapter 4 the error went unnoticed till now. The counterexample is now found by ProB alone as well as can be seen in Table 6.2. Yet again, the counterexamples not identified by the integrated approach in [117] are now discovered by the integrated solver as well.

Table 6.4: Performance on Different Kinds of Proof Obligations (Z3)

Kind of PO	# POs	SMT/Z3	HL/Z3	ProB	ProB/Z3
Feasibility of non-det. action	59	57 (96.6 %)	54 (91.5 %)	57 (96.6 %)	57 (96.6 %)
Guard strengthening	300	294 (98.0 %)	196 (65.3 %)	266 (88.7 %)	274 (91.3 %)
Invariant preservation	4950	4638 (93.7 %)	4108 (83.0 %)	4547 (91.9 %)	4710 (95.2 %)
Natural number for a numeric variant	6	6 (100.0 %)	6 (100.0 %)	4 (66.7 %)	6 (100.0 %)
Action simulation	154	138 (89.6 %)	108 (70.1 %)	134 (87.0 %)	148 (96.1 %)
Theorem	98	59 (60.2 %)	39 (39.8 %)	80 (81.6 %)	64 (65.3 %)
Decreasing of variant	6	6 (100.0 %)	6 (100.0 %)	6 (100.0 %)	6 (100.0 %)
Well-definedness	780	644 (82.6 %)	66 (8.5 %)	548 (70.3 %)	514 (65.9 %)
Feasibility of a witness	1	1 (100.0 %)	0 (0.0 %)	1 (100.0 %)	1 (100.0 %)
Well-definedness of a witness	4	3 (75.0 %)	0 (0.0 %)	4 (100.0 %)	2 (50.0 %)
	6358	5846 (91.9 %)	4583 (72.1 %)	5647 (88.8 %)	5782 (90.9 %)

Table 6.5: Performance on Different Kinds of Proof Obligations (CVC4)

Kind of PO	# POs	SMT/CVC4	HL/CVC4	ProB	ProB/CVC4
Feasibility of non-det. action	59	18 (30.5 %)	39 (66.1 %)	57 (96.6 %)	57 (96.6 %)
Guard strengthening	300	275 (91.7 %)	127 (42.3 %)	266 (88.7 %)	244 (81.3 %)
Invariant preservation	4950	4451 (89.9 %)	3167 (64.0 %)	4547 (91.9 %)	4439 (89.7 %)
Natural number for a numeric variant	6	6 (100.0 %)	5 (83.3 %)	4 (66.7 %)	0 (0.0 %)
Action simulation	154	99 (64.3 %)	65 (42.2 %)	134 (87.0 %)	122 (79.2 %)
Theorem	98	40 (40.8 %)	32 (32.7 %)	80 (81.6 %)	44 (44.9 %)
Decreasing of variant	6	6 (100.0 %)	6 (100.0 %)	6 (100.0 %)	6 (100.0 %)
Well-definedness	780	414 (53.1 %)	75 (9.6 %)	548 (70.3 %)	321 (41.2 %)
Feasibility of a witness	1	0 (0.0 %)	0 (0.0 %)	1 (100.0 %)	0 (0.0 %)
Well-definedness of a witness	4	0 (0.0 %)	0 (0.0 %)	4 (100.0 %)	0 (0.0 %)
	6358	5309 (83.5 %)	3516 (55.3 %)	5647 (88.8 %)	5233 (82.3 %)

Table 6.2 outlines for which models we see better or worse performance for the Z3-based high-level SMT translation. In particular the landing gear systems and the Stuttgart 21 interlocking models show a decline in successfully discharged POs when compared with the low-level SMT translation. These models feature a considerable amount of concrete data that can easily be translated using the low-level approach. We assume that some of these POs can be discharged on the boolean level, without any higher-order reasoning. Table 6.2 also shows that these are the models where ProB alone works well.

As can be seen in Table 6.3 the situation remains largely the same if CVC4 is used instead of Z3. However, the overall performance of CVC4 is not comparable to that of Z3.

The high-level SMT approach, both with and without ProB integration, performs better for more abstract models like the CAN Bus, the graph coloring algorithm and the pacemaker model. This stresses our assumption that integrating the high-level SMT translation into ProB is worthwhile as they represent orthogonal technologies that could benefit from one another. However, scheduling the different solvers under a global timeout remains complicated.

6.5.3 Comparison of CVC4 and Z3

Figure 6.3 compares ProB alone to the integrated solvers based on CVC4 and Z3. As can be seen, integrating Z3 into ProB increases the performance: the integrated solver is able to discharge 5782 proof obligations. Given that ProB already discharged 88.8% of proof obligations, the additional gain of 2.1% is significant.

In particular, the improved detection of unsatisfiability will be helpful if ProB is used as the backend of any symbolic model checking algorithm. The Z3 integration can serve as a replacement as well as an addition to the additional CHR rules given in Section 3.3.3. Using both at the same time is possible and supported in ProB.

Integrating CVC4 into ProB is not nearly as successful: The number of discharged obligations drops from 5647 to 5233, or 82.3%. As can be seen in Fig. 6.1b, ProB/CVC is able to discharge 76 proof obligations not discharged by ProB alone. In contrast to the Z3 integration this does not outweigh those lost due to the increased number of timeouts: ProB alone discharges 490 proof obligations not discharged by the CVC4-based integrated solver.

There is a key difference in behavior between the Z3 integration and the CVC4 integration, revealed when comparing Fig. 6.1a to Fig. 6.1b. ProB/Z3 is able to discharge all the proof obligations solved by HL/Z3. Apparently, integrating ProB and Z3 brings the additional proving power of the high-level translation into the ProB kernel. At the same time, it does not impact the performance of ProB's kernel too much, dragging only 130 proof obligations into a timeout.

The situation is different for CVC4: The integrated solver ProB/CVC4 is unable to successfully handle all obligations discharged by the high-level translation to CVC4 alone. A considerable amount of proof obligations is not discharged anymore, once ProB is added. At the same time, ProB's kernel alone is impacted by adding CVC4, leaving the aforementioned 490 proof obligations behind. At the moment, we suspect that poor scheduling is to blame for the difference between the two SMT solvers. Another possible explanation could be a difference in the unsat core computations of CVC4 and Z3: if CVC4's cores are bigger than those of Z3, there is less backjumping in Algorithm 6.1.

6.6 Related Work

As mentioned above, in [71, 72] the authors present an integration of SMT solvers into Rodin [3], an IDE for Event-B development. In this scenario, the SMT solvers are used as provers in order to discharge Event-B proof obligations. The authors investigate two different ways of translating Event-B to SMT-LIB.

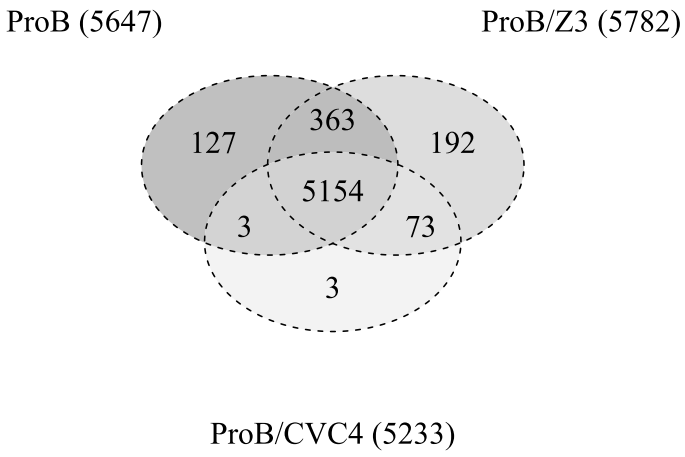


Figure 6.3: Comparison of CVC4 and Z3

For SMT solvers in general they suggest the *ppTrans* approach. Here, set theory and arithmetic are broken down into first-order formulas using uninterpreted functions for membership, etc. On the one hand, this approach is more flexible than the one presented here: it does not rely on the API of a specific SMT solver. On the other hand, the resulting formulas only approximate the Event-B semantics, as operators are replaced by uninterpreted functions. The authors thus add certain set theoretic axioms to the SMT problem in order to recover from this.

A second approach, called *λ-based* relies on an extension to SMT-LIB provided by the veriT solver [33]. Set theoretic constructs are then translated into λ-expressions. The major shortcoming of this approach is that sets of sets cannot be handled.

Many of the rewrite rules presented here are similar to those in [71, 72]. The key difference is that we rely on the given set theory of the SMT solvers instead of translating further into first-order logic. While this is responsible for a considerable performance decline, it allows for easier translation of unsat cores computes by the SMT solvers back to B.

In addition to other SMT-based approaches, there are different ways of solving B and Event-B predicates. ProB itself mainly relies on constraint logic programming. There is also the formerly mentioned backend [160] translating B to Kodkod [177]. Kodkod then uses a SAT solver to find solutions to the given formulas.

Of course SMT solvers have been used for verification tasks relying on specification languages other than B and Event-B. For instance, TLA$^+$ has been translated into many-sorted first-order logic in [149] and [148].

6.7 Conclusion and Future Work

One motivation for the integration of SMT solvers into ProB was to overcome the weaknesses we spotted previously in Chapters 3 and 4: ProB should be enabled to handle infinite domains and detection of unsatisfiability should be improved.

With the suggested high-level translation of B to SMT-LIB both goals could be achieved. The integrated solution is able to discharge several proof obligations not discharged by ProB alone.

In certain cases, using our combined approach seems advantageous over a low-level translation to predicate logic. Indeed, our high-level translation relying on both Z3 and ProB discharges 4782 proof obligations in total, out of which 148 cannot be discharged by the previous SMT translation [71, 72].

Our evaluation also showed that there is not only a gain in the number of proof obligations: the low-level translation discharges 157 proof obligations that are not discharged by the integrated solver. Yet, it is not easy to decide a priori when to employ a high-level and when to employ a low-level approach. A practical solution is to use both in a solver portfolio.

For the future, we have different directions in mind. First, we would like to investigate whether using an equisatisfiable translation instead of an equivalent one is of use. In particular for approaches like proving or disproving as discussed in Chapter 4 we expect improved performance.

We also want to tighten the integration of the SMT solvers and ProB. Currently we are transporting partial assignments and we use the unsat core to control backjumping on the Prolog side. In future, we want to investigate whether we can access and use clauses learned on the SMT side in order to set up further constraints on the Prolog side. For instance, we want to investigate whether we can use interpolants or conflict clauses in case of unsatisfiable predicates.

Regarding our translation to SMT-LIB, the benchmarks show that in particular the usage of quantifiers can be improved. One option to do so is to further investigate how to set instantiation triggers for comprehensions typically occurring in our scenarios. In [125] the authors already outlined a general approach that can serve as a starting point.

Another option is to try to reduce the amount of quantifiers we use. This could be achieved by providing a custom theory to the SMT solvers, e. g., including inference rules for min and max, avoiding some of the quantifiers introduced in Section 6.2.2. Changing the set of axioms we supply to the SMT solvers in order to define min and max is certainly another direction that should be evaluated.

Another technique we want to implement should help us to overcome some limitations discussed in Section 6.4. As mentioned, the B type system allows to use set operators on sequences. Hence, we had to represent sequences as sets of pairs, instead of relying on a native sequence type. A static check could investigate how operators are applied in a B machine. It could determine if sequences are only used with sequence operators. In this case, we could employ a more efficient translation to SMT-LIB, e. g., encode them as arrays or try the preliminary support for sequences recently introduced to CVC4 [131] and Z3 [189].

Regarding benchmarks and applications, we would like to move from solving predicates to explicit state model checking and constrained based validation techniques. A first step towards more thorough benchmarks is the usage of our integrated solver as the backend of several symbolic model checking algorithms.

It remains yet to be seen, how SMT solvers will evolve regarding high-level theories. The current version of the SMT-LIB standard only features a "possible declaration for a theory of sets and relations" [18]. How and if different possibilities are realized will certainly influence the impact SMT solvers have in the formal methods community.

Summarizing, we provided new ways to tackle the complexity of constraints in B and Event-B. We provided a new high-level translation of B to CVC4's and Z3's input language, which can be used on its own or integrated into ProB's solver.

This high-level SMT based solver appears to be an orthogonal addition to the other solvers, solving many constraints that could not be solved by the previous low-level translation and is better suited at finding models. Our evaluation also confirms that the integration of the ProB solver with SMT solvers is worthwhile, discharging 5782 proof obligations in case of Z3. It is only outperformed by the low-level translation, which cannot fully be used to generate models. The combined solver should be able to handle constraints brought up by symbolic model checking algorithms, both regarding detection of unsatisfiablity and model finding.

Part III

Symbolic Model Checking

Quick decisions are unsafe decisions.

Sophocles

7

Selecting Symbolic Model Checking Algorithms

The following chapter will discuss several symbolic model checking algorithms. After the introduction in Section 7.1, we start with symbolic model checking using BDDs in Section 7.2. Following, the chapter will mainly focus on SAT and SMT based model checking techniques. In Sections 7.3.2 to 7.3.4 bounded model checking, interpolation-based model checking and k-Induction will be introduced. Section 7.3.5 will then introduce the IC3 algorithm. We will conclude with an evaluation of the algorithms in question, justifying the decision to implement BMC, k-Induction and IC3 in ProB.

This chapter is based on our paper "Proof Assisted Symbolic Model Checking for B and Event-B" [116]. For information regarding authors and their individual contributions see Appendix C.

7.1 Introduction and Motivation

Two variants of model checking are currently in use: explicit state model checking and symbolic model checking. In explicit state model checking, every reachable state is traversed, the invariant is verified and discovered successor states are queued to be analyzed themselves.

Symbolic model checking on the other hand tries to represent the state space and possible paths through it by predicates representing multiple states or paths

at once. Instead of stepwise exploration of the state space graph, the model checking problem is encoded as a formula and given to a constraint solver.

So far, existing model checkers for B and Event-B like ProB [128, 127], Eboc [141], pyB [185] or TLC [187] (via [94]) rely on explicit state model checking. ProB features some symbolic techniques for error detection [91] and test-case generation [165], but not full-blown symbolic model checking.

This is mostly due to the high-level nature of B and Event-B. Both the use of higher-order constructs and the underlying non-determinism accounts for complicated constraints during symbolic model checking. Some complexity can be coped with by relying on SMT solvers [71, 72] or SAT solvers [160]. However, this is not always the case, as translation to SMT or SAT can be complicated and might add overhead. Additionally, some predicates cannot efficiently be solved by SAT or SMT solvers as well. Often, the remaining constraints are still too complicated for symbolic model checking to be feasible.

7.2 State Space Representation using BDDs

In the following section, we will present binary decision diagrams [124] and explain how they can be used to reduce the impact of the state space explosion problem. Binary decision diagrams, or BDDs for short, have been introduced in 1959 as a concise representation of binary switching circuits and are defined as follows.

Definition 7.2.1 (Binary Decision Diagram)**.** A *binary decision diagram* is an acyclic and directed graph $G = (V, E)$, such that

- Each leaf node is either labeled $\top$ or $\bot$.
- Each inner node N belongs to a boolean variable X_N.
- Each inner node N has two child nodes called low_N and $high_N$. The edge to low_N represents assigning $X_N = \bot$ while the edge to $high_N$ represents $X_N = \top$.

As an example, take the formula $f = (\neg x \wedge \neg y) \vee (y \wedge z) \vee (\neg x \wedge z)$. Figure 7.1 shows the decision tree and the corresponding BDD for f. The edges to low nodes are dashed while the edges to high nodes are solid.

BDDs have later been extended to shared reduced ordered binary decision diagrams by Bryant [37, 38]. The changes include several optimization steps. Together they ensure that BDDs are a canonical representation of boolean formulas (modulo variable ordering).

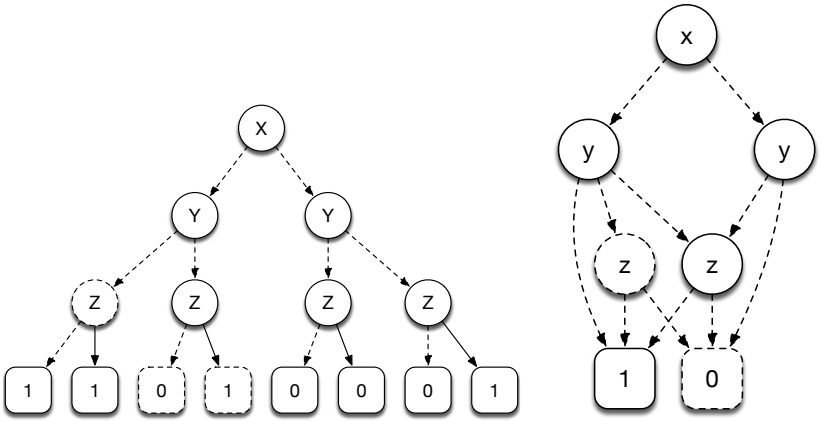


Figure 7.1: Decision Tree and BDD for f

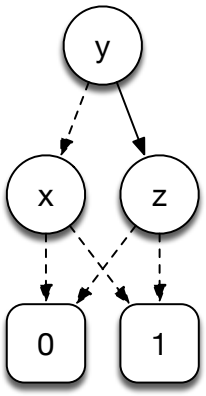


Figure 7.2: Reduced Ordered BDD for f

Definition 7.2.2 (Ordered Binary Decision Diagram)**.** A binary decision diagram is called *ordered* if the variables X_N appear in the same order on all paths from the root to the leafs.

Definition 7.2.3 (Reduced Binary Decision Diagram)**.** A binary decision diagram is called *reduced* if all isomorphic subgraphs have been merged and all nodes whose children are isomorphic have been eliminated.

The variable ordering can indeed considerably influence the size of a resulting BDD. However, finding the optimal variable ordering is an NP-hard problem [30].

In Fig. 7.2 the reduced ordered BDD for the example above is shown. As can be seen, removing isomorphic nodes often leads to a substantial reduction of BDDs.

The value of a boolean function represented by a BDD can be computed recursively using Boole's expansion theorem:

Definition 7.2.4. For a BDD $G = (V, E)$ and a node $v \in V$, the value of the boolean function f_v represented by v is

- The label of v iff v is a leaf node.
- If v is an inner node belonging to boolean variable X_v, the value of f_v is given by $f_v(x) = X_v \wedge f_{high_v}(x) \vee \neg X_v \wedge f_{low_v}(x)$.

Operations like conjunction or disjunction that can be performed on boolean formulas can often be performed on the reduced and ordered form of a BDD without having to reorder or rereduce the result. This allows them to be used as the representation of states during model checking [39], avoiding the state space explosion to some extent.

Unfortunately, using BDDs is often not enough to handle the state spaces of large models, especially when considering software rather than hardware model checking. While the influence of the number of states is reduced, the asymptotic complexity of explicit state model checking remains [39]. In consequence, memory and performance issues ultimately limit the applicability of BDD-style symbolic model checking.

BDD-style symbolic model checking as outlined in [39] has been implemented for B and Event-B by integrating ProB with LTSmin [23]. While originally targeted at deadlock checking, the integration has been extended towards invariant checking and verification of LTL formulas in [118].

Using BDD-style symbolic model checking is a different line of work and not considered further in this thesis.

7.3 Fully Symbolic Algorithms

7.3.1 Notation and Running Example

We will use the running example in Listing 7.1 to illustrate various concepts. The techniques used in this chapter have been implemented both for classical B and Event-B. Both languages will be used in our empirical evaluation in Section 8.3. For the sake of brevity we will only talk about Event-B events in the following sections instead of distinguishing events and operations.

First, let us introduce the notation we will be using. By x we will denote a vector of state variables. x' denotes the state variables in the successor state. A predicate p over the state variables x is denoted by $p(x)$. The same predicate

Listing 7.1: A simple, erroneous B machine

```
MACHINE Counter
CONSTANTS m
PROPERTIES m : {127,255}
VARIABLES c
INVARIANT c>=0 & c<=m
INITIALISATION c:=0
OPERATIONS
  incby(i) = PRE i:1..64 THEN c := c+i END
END
```

over the successor state is written as $p(x')$. By *Events* we denote the set of events of an Event-B machine. By *Inv* we denote its invariant.

Definition 7.3.1. For an event $evt \in Events$ let $BA_{evt}(x, x')$ denote the before-after-predicate connecting state variables in x to their successors in x'. In order to fully represents a system's behavior using the before-after-predicate, we do not only consider actions but also guards and parameters where applicable.

For the example in Listing 7.1, we have $BA_{incby}(c, c') = \exists\, i.\, (i \in 1..64 \wedge c' = c + i)$.

Definition 7.3.2. By T we refer to a *monolithic transition predicate*, i. e., the disjunction of all before-after-predicates: $T(x, x') = \bigvee_{e \in Events} BA_e(x, x')$. By I we denote the *after predicate of the initialization*, including the properties about the constants.

For Listing 7.1 we have $I(c) = m \in \{127, 255\} \wedge c = 0$.

7.3.2 Bounded Model Checking

Bounded Model Checking, or BMC for short, has been suggested by Armin Biere, et al. in 1999 [25]. One of the main goals is to avoid the blowup and resulting slowdown of BDD-based model checking algorithms. This is achieved by replacing the BDDs by a SAT solver.

The basic idea is as follows: For an initial state relation I, a transition relation T, a property p and a bound k starting with $k = 0$, a sequence of propositional formulas is generated. Each of the formulas is satisfiable if and only if there exists a counterexample to the property with length $\leq k$.

This can be expressed as:

$$BMC(p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i). \tag{7.1}$$

In [25], the authors identify the following key advantages of bounded model checking:

- Due to the depth first search performed by the underlying SAT solver, counterexamples are usually detected quite fast.
- The algorithm always finds the shortest path to a counterexample by incrementally increasing the bound k.
- The approach avoids the shortcomings of BDD based model checking, such as the memory blowup.

In its pure form, bounded model checking is obviously incomplete: The unsatisfiability of $BMC(p,k)$ for any k does not imply anything regarding states reachable in more then k steps. For bounded model checking to be complete, one has to find an upper bound for k.

Definition 7.3.3 (Completeness Threshold)**.** For M a finite model and P a safety property there exists a minimal k such that the absence of a counterexample within k or fewer steps implies that $M \vDash P$. This k is called the *completeness threshold*.

If the completeness threshold is known, bounded model checking becomes complete: Starting with 0, a bounded model checking step is performed for each value of k smaller than the threshold. If no counterexample is detected, the model is proven correct.

Definition 7.3.4 (Diameter)**.** The *diameter* of a model M is the length of the longest of all shortest execution sequences connecting any two states.

Lemma 7.3.1. *For safety properties the completeness threshold is less than or equal to the diameter.*

Even though we can compute the completeness threshold for certain properties, doing so usually involves computing the full state space of a given model. This would involve using an explicit state model checker as introduced in Section 2.2.1. Afterwards, using BMC does not offer any benefit. However, there are techniques to compute an over-approximation of the completeness threshold [54].

While initially designed for hardware verification, bounded model checking can be extended to software in a straight forward fashion. Two possible techniques are used:

1. Datatypes can be translated into propositional formulas using for example a bit-width encoding for integers. Using this technique, richer programs tend to consume a lot of boolean variables. Hence, possible scalability issues might arise.

2. The approach can easily be lifted from propositional satisfiability to richer logics by replacing the SAT solver with an SMT solver. No changes to the algorithm are necessary.

Using only the encoding in Eq. (7.1) lifting bounded model checking to infinite state spaces is impossible, since no bound k exists in this case. Of course, even for infinite models, bounded model checking is still useful to search for counterexamples with a finite length.

Furthermore, there are extensions to BMC that enable infinite-state model checking. For instance in [62] the authors show how theorem proving can be used to apply bounded model checking to models featuring infinite domains. We will follow a similar approach in Chapter 8, where we use static proof information to improve the performance of symbolic model checking algorithms for B and Event-B.

An alternative approach is proposed in [169], where the authors suggest using more expressive logics in order to lift BMC to infinite state spaces. Bounded model checking can be lifted to software using SMT solvers as suggested in [11]. Another model checking technique comparable to BMC is based on interpolants and will be introduced below.

7.3.3 Interpolation

Interpolation can be used to lift BMC to an unbounded model checking algorithm as outlined in [142].

Definition 7.3.5. $\mathcal{I}$ is called an *interpolant* for an unsatisfiable formula ϕ, if

1. $\phi = A \wedge B$,

2. $A \Rightarrow \mathcal{I}$,

3. $\mathcal{I} \wedge B$ is unsatisfiable,

4. $\mathcal{I}$ only refers to the common variables of A and B.

The last requirement rules out using A itself as an interpolant. In general, there can be multiple interpolants for ϕ. If used in model checking algorithms, this can influence performance of approximations, depending on the choice of interpolants used.

Interpolants can be extracted from the refutations produced by SAT solvers using a linear procedure [162]. Producing interpolants for the theories used in SMT solvers is more involved but still possible for several theories [144]. These methods are based on the construction of interpolants from proofs. A more direct approach has been suggested in [163], relying on constraint logic programming for interpolant construction.

Following [142], interpolants can be used in the constraints of bounded model checking as follows. The conjuncts of the BMC constraints given in Eq. (7.1) are split into two partitions. Initially, the first partition (A) includes the initial state predicate and the first instance of the transition predicate. The remaining transitions together with the negated property form the second partition (B):

$$BMC(p,k) = \underbrace{I(s_0) \wedge \neg p(s_0) \wedge T(s_0, s_1)}_{A} \wedge \underbrace{\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg p(s_i)}_{B}$$

If there is no counterexample, the formula is unsatisfiable. From the underlying solver, we can then extract $\mathcal{I}$, an interpolant for $BMC(p,k)$. The common variables in this case are the variables in state s_1. Since $\mathcal{I}$ is implied by A, I is an over-approximation of the states reachable in a single step. Furthermore, since $\mathcal{I} \wedge B$ are unsatisfiable no state satisfying $\mathcal{I}$ can reach a counterexample in k steps. $\mathcal{I}$ can be seen as an under-approximation of the states that are backwards reachable from a counterexample state in k steps.

The process can now be repeated each time the bound k is increased. Due to the approximation, spurious counterexamples can occur. However, increasing k will ultimately either lead to a real counterexample once k is large enough or it will prove the counterexample to be spurious. The absence of a counterexample is detected using a fix-point loop. By gradually refining both the over- and under-approximations, one finds an inductive invariant as soon as a fix-point is reached.

Interpolation based model checking can be combined with an abstraction technique like predicate abstraction to make it more suitable for software model checking [105].

7.3.4 k-Induction

k-Induction [171] is a model checking technique for finite state systems. It follows the idea of a mathematical proof by induction, i. e., it tries to establish a base case and proof an inductive step. The algorithm thus tries to prove

that if a property p holds at the current step it holds after performing another transition.

For the method to be complete, one has to avoid getting stuck in loops. Hence, the constraints are strengthened to avoid a state occurring twice on given a path.[1]

The base condition is encoded as shown in Eq. (7.2); it is basically a BMC step and tries to find a counterexample of length k starting from the initialization. Like in Section 7.3.2 we assume that we gradually increase the value of k starting from 0, as shown in Algorithm 7.1.

The inductive step, including the uniqueness of states, can be expressed as done in Eq. (7.3), where Axm are the axioms on the constants of the model (e. g., $m \in \{127, 255\}$ in our running example).

$$Base(p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k) \quad (7.2)$$

$$Step(p,k) = Axm \wedge \bigwedge_{0 \leq i < j \leq k} s_i \neq s_j \wedge \bigwedge_{i=0}^{k} T(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} p(s_i) \wedge \neg p(s_{k+1}) \quad (7.3)$$

The model checking algorithm now starts with $k = 0$. As long as the base constraint is not satisfiable, no counterexample of length k exists. If the step constraint is not satisfiable as well, no further steps can be performed and the state space has been traversed exhaustively. If it is not, k is increased. During the search, Eq. (7.2) should never be satisfiable. Otherwise, a counterexample to the property has been found. Summarizing, a k-Induction based model checking algorithm performs as outlined in Algorithm 7.1.

For $k = 0$, $Step(Inv, k)$ corresponds to trying to find counterexamples to the B invariant preservation proof obligations. In a similar fashion, $Base(Inv, 0)$ corresponds to finding initial states which violate the invariant. Hence, if $Base(p, 0)$ and $Step(p, 0)$ are unsatisfiable, we have found an inductive proof of the property p.

However, the difference with B's approach to proving invariants does appear when $Step(p, 0)$ is satisfiable, i. e., when there exists a state which satisfies p and a successor state violates p. The k-Induction method tries to construct a real counterexample, starting from a valid initial state, not from *any* state satisfying p. Hence, the value of k is now increased and we try to find a real counterexample of length $k + 1$ using the BMC constraint given in Eq. (7.2).

[1] We could have added these constraints $s_i \neq s_j$ also in Section 7.3.2.

Algorithm 7.1: k-Induction

```
Data: Property p
Result: true iff p holds
1  procedure boolean k-Induction(p)
2      k := 0
3      while true do
4          if Base(p, k) satisfiable then
5              return false
6          end if
7          if Step(p, k) unsatisfiable then
8              return true
9          end if
10         k := k + 1
11     end while
```

For infinite state systems, there might always be another step that can be performed. In consequence, proof by induction can fail. Hence, Eq. (7.3) is not guaranteed to be unsatisfiable eventually. As a result, for certain models, k-Induction can only be used to disprove but not to prove a property.

Compared to BMC as introduced in Section 7.3.2, k-Induction has the advantage of including an explicit termination condition. No prior knowledge or computation is needed to determine whether model checking was exhaustive.

Suppose for example we take for p the predicate $c \geq -2 \wedge c \neq -1$ for Listing 7.1. In this case BMC will never terminate, as for every value of k no counterexample can be found. k-Induction, however, can already stop with $k = 1$, as $\mathit{Step}(\mathit{Inv}, 1)$ is unsatisfiable. This is an interesting result, given that the state space of the model is infinite. The constraints are shown in Fig. 7.3 and are unsatisfiable except for $\mathit{Step}(\mathit{Inv}, 0)$, which has the solution $m = 127, c_0 = -2, c_1 = -1$.

In more detail, the constraints from Fig. 7.3 are:

$$\begin{aligned}
\mathit{Base}(\mathit{Inv}, 0) = \quad & m \in \{127, 255\} \wedge c_0 = 0 \wedge \neg(c_0 \geq -2 \wedge c_0 \neq -1) \\
\mathit{Step}(\mathit{Inv}, 0) = \quad & m \in \{127, 255\} \wedge c_0 \geq -2 \wedge c_0 \neq -1 \\
& \wedge \exists\, i.\, (i \in 1..64 \wedge c_1 = c_0 + i) \wedge c_1 \neq c_0 \\
& \wedge \neg(c_1 \geq -2 \wedge c_1 \neq -1)
\end{aligned}$$

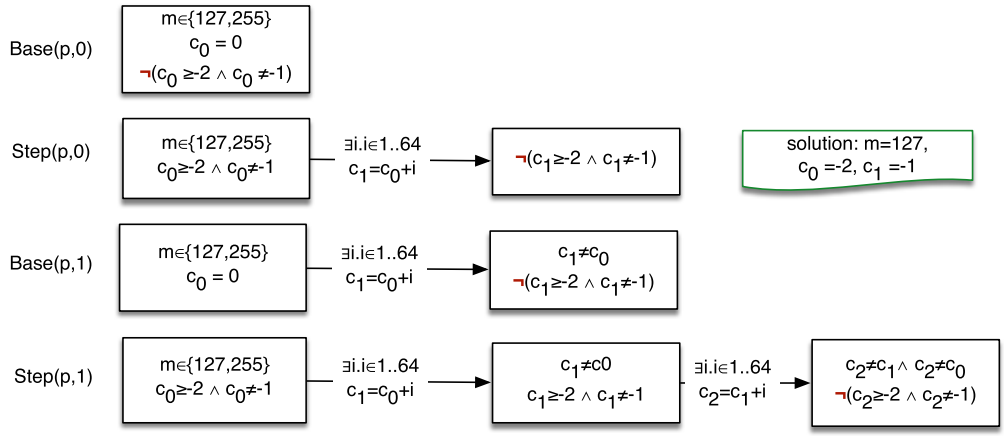


Figure 7.3: k-Induction on Example in Figure 7.1 and Property $c \geq -2 \wedge c \neq -1$

$$
\begin{aligned}
Base(Inv,1) = \quad & m \in \{127,255\} \wedge c_0 = 0 \wedge (c_0 \geq -2 \wedge c_0 \neq -1) \\
& \wedge \exists\, i.\, (i \in 1..64 \wedge c_1 = c_0 + i) \wedge \neg(c_1 \geq -2 \wedge c_1 \neq -1) \\
Step(Inv,1) = \quad & m \in \{127,255\} \wedge c_0 \geq -2 \wedge c_0 \neq -1 \\
& \wedge \exists\, i.\, (i \in 1..64 \wedge c_1 = c_0 + i) \wedge c_1 \neq c_0 \wedge (c_1 \geq -2 \wedge c_1 \neq -1) \\
& \wedge \exists\, i.\, (i \in 1..64 \wedge c_2 = c_1 + i) \wedge c_2 \neq c_1 \wedge c_2 \neq c_0 \\
& \wedge \neg(c_2 \geq -2 \wedge c_2 \neq -1)
\end{aligned}
$$

$Step(Inv, 0)$ corresponds to the B proof obligation for the event, checking that p is inductive relative to a single predecessor state. In contrast, k-Induction successively increases the number of base cases until it can show that p is indeed inductive relative to a number of predecessor states. Hence, the B proof method is not able to prove that $c \geq -2 \wedge c \neq -1$ always holds. A user would need to find an inductive invariant such as $c \geq 0$ implying the property. The IC3 algorithm in the next section will do just that automatically.

7.3.5 IC3

The IC3 [34, 73] model checking algorithm has initially been developed by Aaron Bradley for hardware model checking. In contrast to BMC presented in Section 7.3.2 and k-Induction presented in Section 7.3.4 the IC3 algorithm does not use an unwinding ($\bigwedge_{i=0}^{k} T(s_i, s_{i+1})$) of the transition system. Instead, only single step queries are performed.

In order to verify a system, IC3 tries to *automatically* find an inductive invariant implying the property in question. To do so, it keeps a list of *frames* F_i over-approximating the set of states reachable in $\leq i$ steps. For I a predicate

describing the initial states and T a predicate describing the transition relation and P the property in question, these frames satisfy the following invariants:

$$\begin{aligned} I &\Leftrightarrow F_0 \\ \forall\, 0 \leq i < k.\, F_i &\Rightarrow F_{i+1} \\ \forall\, 0 \leq i < k.\, F_i &\Rightarrow P \\ \forall\, 0 \leq i < k.\, F_i \wedge T &\Rightarrow F'_{i+1} \end{aligned}$$

Counterexamples reachable in one or two steps are handled as a special case, as shown in line 2 of Algorithm 7.2. Afterwards, for each level k IC3 tries to find a property violation reachable in a single step, i. e., a solution to $F_k(s) \wedge T(s, s') \wedge \neg p(s')$.

If no solution exists, k is incremented and a new frame holding p is added. Additionally, the algorithm checks whether it can push forward clauses from lower to higher frames. This is done by checking whether they are implied to hold after the execution of a transition, i. e., for a clause c the predicate $c(s) \wedge T(s, s') \Rightarrow c(s')$ holds.

Otherwise, IC3 tries to show that the faulty state is in fact not reachable from the initialization. This is done by incrementally strengthening frames until F_k becomes strong enough to prevent the property violation from occurring. To do so, IC3 keeps a queue of remaining counterexamples, starting with $(cs, k-1)$, where cs is a predicate describing the counterexample state found for frame number k.

For each tuple (cs, k), IC3 tries to prove cs unreachable by verifying the relative inductiveness of cs to F_k. This is achieved by the query $F_k(s) \wedge \neg cs(s) \wedge T(s, s') \Rightarrow \neg cs(s')$. If cs is indeed inductive, it can be added to the frames $F_i, 1 \leq i \leq k+1$.

If it is not, the query is satisfiable and returns a valuation representing a successor state to the one defined by cs. In case the successor found is an initial state and $k = 0$, IC3 has found a counterexample trace and the system under consideration is indeed faulty. Otherwise, IC3 tries to prove the predecessor unreachable, by creating a new tuple and adding it to the queue.

The added clause will have to be propagated through the frames in order to reestablish the properties stated above. Of course, it is desirable to generalize added clauses before propagation in order to exclude a number of counterexamples from occurring. This is done by computing a subclause of cs with the desired properties. A number of generalization procedures has been suggested:

- In Bradley's original implementation [34], a stepwise cone of influence [25] is used to compute the desired clause.

- An improvement to the generalization procedure is described in [97], where the authors consider counterexamples to generalization based on multiple states in contrast to the counterexamples to induction described above.
- Using ternary simulation has been suggested in [73].
- Using SAT conflict analysis, counterexample states can be reduced to partial assignments as well [45]. Here, the cause of a conflict in a specially crafted query consists of only the variable assignments needed.

We use the last approach, as it integrates nicely with the existing solving procedures: given a counterexample state predicate $cs(s)$, its successor state predicate $ct(s')$, and T_z the exact transition that connects s to s', we know that $cs(s) \wedge T_z(s, s') \wedge \neg ct(s')$ is unsatisfiable. Note that T_z removes existing non-determinism from T in order to reproduce the same exact transition. From the unsatisfiable query, one can extract an unsatisfiable core, typically including only a subclause of cs.

After generalization and strengthening the frames, a new counterexample might be found and IC3 will start to iterate between finding counterexamples and strengthening frames. If strengthening the frames eventually fails, a counterexample to the property is found.

IC3 can also prove a model correct. If no counterexample can be found anymore, the frames have been strengthened enough to form an inductive invariant. Furthermore, due to the properties established, the invariant found implies the property in question. It is thus proven to hold on every reachable state of the model. Indeed, an inductive invariant has been found.

A partial outline of IC3 is shown in procedure *strengthen* in Algorithm 7.3. The *Counterexample* exception is thrown by *inductivelyGeneralize* if generalization fails and the counterexample cannot be proven spurious.

In the following chapters, we will only go into details of IC3 wherever it has to be adapted in order to work with B and Event-B. For a complete overview, see Bradley's original paper [34] or the one by Een, et al. in [73]. Algorithms 7.2 and 7.3 follow the implementation of [34].

Let us see how our running example is solved by IC3: First, the algorithm checks if the initial state of the model can already violate the property. This is done using the same query as in BMC, $BMC(Inv, 0)$, as given in Eq. (7.1). Afterwards, one-step reachability is checked using the query $BMC(Inv, 1)$. Both queries are unsatisfiable.

IC3 now sets up the first two frames, F_0 and F_1, where F_0 holds the initial state of the machine and F_1 holds the property in question. The domain in which the machine's constants have to reside is added to the frames as well. A third

Algorithm 7.2: IC3: Main Loop

```
   Data: Property p
   Result: true iff p holds
 1 procedure boolean ic3(p)
 2     if sat(I(s) ∧ ¬p(s)) ∨ sat(I(s) ∧ T(s, s′) ∧ ¬p(s′)) then
 3         return false
 4     end if
 5     F_0 := I, clauses(F_0) := ∅
 6     F_1 := p, clauses(F_1) := ∅
 7     k := 1
 8     while true do
 9         if not strengthen(k, p, F) then
10             return false
11         end if
12         propagate_clauses(k)
13         if ∃ i. (i ∈ [1, k] ∧ clauses(F_i) = clauses(F_{i+1})) then
14             return true
15         end if
16         k := k + 1
17     end while
```

frame F_2 to use it as a target for propagation. Again, it holds both the possible constants and the property in question. In summary, we have

$$\begin{aligned} F_0 &= m \in \{127, 255\} \wedge c = 0 \\ F_1 &= m \in \{127, 255\} \wedge c \geq 0 \wedge c \leq m \\ F_2 &= m \in \{127, 255\} \wedge c \geq 0 \wedge c \leq m. \end{aligned}$$

The algorithm now enters the `strengthen` step with $k = 1$. Here, it tries to reestablish the frame invariants given above. This is done by finding counterexamples to the inductiveness of $c_0 \geq 0 \wedge c_0 \leq m$ and reacting to them. The while loop in Algorithm 7.3 effectively iterates over all the counterexamples of length $k + 1 = 2$. For $k = 1$ the query in the while loop is

$$\begin{aligned} & F_1(s) \wedge T(s, s') \wedge \neg p(s') \\ \Leftrightarrow & m \in \{127, 255\} \wedge c \geq 0 \wedge c \leq m \\ & \wedge \exists\, i. \, (i \in 1..64 \wedge c' = c + i) \wedge \neg(c' \geq 0 \wedge c' \leq m) \end{aligned}$$

There are a number of possible solutions, each representing a counterexample to the inductivity of $c \geq 0 \wedge c \leq m$. Let's say $m = 127 \wedge c = 127 \wedge c' = 128$ (with $i = 1$) is the first to be found by ProB. Following Algorithm 7.3 we extract the predecessor $s(c)$ which is $c = 127$.

Algorithm 7.3: IC3: Strengthen

```
1  procedure boolean strengthen(k, p, F)
2      try
3          while sat(F_k(s) ∧ T(s, s') ∧ ¬p(s')) do
4              pre := the predecessor extracted from the witness
5              n := inductivelyGeneralize(pre, k − 2, k)
6              pushGeneralization((n + 1, pre), k)
7          end while
8          return true
9      catch Counterexample
10         return false
```

Afterwards, the call to `inductivelyGeneralize` is supposed to compute a subclause of $\neg c = 127$ that is inductive relative to some F_i with $i < k$. Inductivity is checked using a query of the form $F_i \wedge \neg pre(c) \wedge T(c, c') \Rightarrow \neg pre(c')$.

In fact, $c = 127$ is inductive relative to F_0, as

$$\underbrace{m \in \{127, 255\} \wedge c = 0}_{F_0} \wedge \neg \underbrace{c = 127}_{pre(c)} \wedge \underbrace{\exists\, i.\, (i \in 1..64 \wedge c' = c + i)}_{T(c,c')} \wedge \underbrace{c' = 127}_{pre(c')}$$

is unsatisfiable. In consequence, the algorithm is allowed to add $\neg c = 127$ to F_1, effectively detecting that the counterexample is spurious.

In the next iteration, another counterexample is found and the process again strengthens the frames. Eventually, no further counterexample can be found and the model has been proven correct.

As you can see, IC3 has to consider a possibly infinite amount of counterexamples. Thus, it is often necessary to incorporate abstraction techniques into IC3 to enable it to handle infinite state spaces efficiently. Several abstraction techniques have been suggested and implemented on top of IC3. We will discuss some of them in Chapter 9.

Aside from safety properties, the IC3 algorithm has already been extended to liveness properties. With FAIR [35] there exists a variant of IC3 for ω-regular properties like those expressed in LTL. Furthermore, CTL properties can be model checked by a variant called IICTL [96]. In addition, IC3 has been extended to timed systems in [107] as well as to bit vector problems in [184].

Table 7.1: Comparison of Symbolic Model Checking Algorithms

	Explicit State	BMC	k-Induction	interpolation	IC3
falsification	✓	✓	✓	✓	✓
verification	✓	(✓)	✓	✓	✓
infinite-state	✗	(✓)	✓	✓	✓
suitable for B	✓	✓	✓	(✗)	✓

7.4 Evaluation and Decision

The key properties of the algorithms introduced above are summarized in Table 7.1. As you can see, two of the algorithms have drawbacks:

- Bounded model checking is limited when it comes to verification instead of falsification. Computing an upper bound to achieve a termination condition is complicated and often infeasible.
- Interpolation could in theory be used to model check B specifications. However, we are limited by ProB's kernel. So far, no efficient interpolation procedures have been implemented for the CLP(FD) package of SICStus Prolog.

As bounded model checking is still useful for the falsification of models, we will include it in the algorithms to be implemented. In addition, k-Induction as well as IC3 and its extension seem likely candidates. Especially IC3's focus on single step queries should be helpful. We hope that it will prevent constraints from accumulating until they cannot be solved by ProB anymore.

Chapter 8, reports on our experience using these algorithms to model checking B and Event-B specifications. At the same time, it will explain how the results of proving efforts can be used to increase performance.

The complicated constraints used in the symbolic model checking algorithms and the lack of interpolations procedures in ProB were the major motivations for the integration of Z3 into ProB. As we have shown in Chapter 6, integrating Z3 improves the constraint solving capabilities of ProB. However, Z3 is often too weak to solve B constraints on its own, without support by ProB. This again limits the availability of efficiently computed interpolants. We thus decided not to implement interpolation based model checking.

For a more in-depth comparison of different symbolic model checking algorithms see [6].

Man cannot produce a single work without the assistance of the slow, assiduous, corrosive worm of thought.

Eugenio Montale

8

Proof Assisted Symbolic Model Checking

This chapter is based on our paper "Proof Assisted Symbolic Model Checking for B and Event-B" [116]. For information regarding authors and their individual contributions see Appendix C.

8.1 Introduction and Motivation

Initial prototypical implementations of the symbolic model checking algorithms selected upon in the previous chapter did not perform as expected. To improve, we studied various ways to use proof information to optimize them. The proof information is used to strengthen constraints and reduce the counterexample search space. For Event-B, the information stems from discharged proof obligations exported from Rodin [3]. For classical B, no automatic proof information is available at the moment within ProB.[1]

However, we can recompute the proof information upon loading a B model. Essentially, for a B operation with before-after-predicate BA we search for a solution to

$$invariant(x) \wedge BA(x, x') \wedge \neg conjunct\_of\_invariant(x').$$

[1] In theory, one could export proof information from Atelier B.

If ProB reports a contradiction, we know that the operation cannot lead to a violation of the particular conjunct. In addition to the techniques used in Chapter 4, we used a bridge to the Atelier B provers ml and pp to discharge these proof obligations.

We already introduced the model checking algorithms BMC, k-Induction and IC3 in Chapter 7. In Sections 8.2.1 to 8.2.3, we will show how to include proof information into the occurring constraints.

Including proof information will result in simpler constraints to be tackled by ProB. Furthermore, in certain cases, proof information helps to reduce the search space in question, enabling ProB to exhaustively model check infinite-state systems.

Following, in Section 8.3, we will empirically compare symbolic model checking to explicit state model checking and model checking with and without proof assistance. Related work is given in Section 8.4, discussion and conclusions will be presented in Section 8.5.

8.2 Integrating Proof Assistance

When using the B method to develop a software or system, one often alternates between different phases. Among those are writing and adapting the specification, manual and automated proof efforts as well as model checking.

These steps usually influence one another: an error detected by model checking is resolved by changing the specification. This again leads to new proof obligations being generated and discharged if possible. Yet, the different steps are only loosely coupled when it comes to tool support. Model checkers often do not incorporate proof information; the provers have no knowledge about model checking results.

In [24] the authors have shown how to augment explicit state model checking with proof information. In the following, we similarly incorporate proof information into the symbolic model checking algorithms introduced in Chapter 7. To do so, we need a few definitions regarding proof information and how we use it.

Definition 8.2.1. For a predicate $p = \bigwedge_{i \in I} p_i$ and event evt let $proven_{evt,p}$ denote a set of conjuncts p_i that are proven to hold after the execution of evt on a p-state, i. e., we have $proven_{evt,p} = \bigwedge_{i \in J} p_i$ for some $J \subseteq I$ such that

$$\forall\, x, x'.\, p(x) \wedge BA_{evt}(x, x') \Rightarrow proven_{evt,p}(x').$$

Let $unproven_{evt,p} = \bigwedge_{i \in I \setminus J} p_i$ denote the complement of $proven_{evt,p}$, i. e., all the conjuncts of p that are not in $proven_{evt,p}$. We also define $proven_{evt} = proven_{evt,Inv}$ and $unproven_{evt} = unproven_{evt,Inv}$.

For the example in Listing 7.1, $proven_{incby}(c) = c \geq 0$; the invariant $c \geq 0$ is preserved by *incby*. This implies $unproven_{incby}(c) = c \leq m$.

In our current implementation, we have that $proven_{evt,p} = \bigwedge_{j \in J} p_j$ with $J \subseteq I$. This however is not a strict limitation. One could add other predicates discovered to be implied to $proven_{evt,p}$ to further strengthen the predicates given below.

Lemma 8.2.1. *A valid solution for Definition 8.2.1 is always* $proven_{evt,p} = true$*, meaning that nothing is proven for the event evt. At the other extreme, if all conjuncts of p are proven to hold after the execution of evt then* $proven_{evt,p} = p$ *and* $unproven_{evt,p} = true$*.*

In the following sections, we show how proof information can be embedded in the queries of bounded model checking (Section 8.2.1), k-Induction based model checking (Section 8.2.2) and IC3 (Section 8.2.3). An empirical evaluation of the algorithms and the influence of using proof information will be performed in Section 8.3.

8.2.1 BMC — Bounded Model Checking

BMC has been introduced in Section 7.3.2. In order to include proof information we have to rewrite the predicate

$$BMC(p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i).$$

First, if we increase k step-by-step as done in ProB's implementation of BMC, it is sufficient to check only the last state for a violation of p:

$$BMC(p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k)$$

For the example machine in Listing 7.1, we have:

$$BMC(Inv,0) = m \in \{127,255\} \wedge c_0 = 0 \wedge \neg(c_0 \geq 0 \wedge c_0 \leq m) \quad (8.1)$$

$$BMC(Inv,1) = m \in \{127,255\} \wedge c_0 = 0 \wedge \exists\, i.\, (i \in 1..64 \wedge c_1 = c_0 + i) \wedge \neg(c_1 \geq 0 \wedge c_1 \leq m) \quad (8.2)$$

$$BMC(Inv,2) = m \in \{127,255\} \wedge c_0 = 0 \wedge \exists\, i.\, (i \in 1..64 \wedge c_1 = c_0 + i) \wedge \exists\, i.\, (i \in 1..64 \wedge c_2 = c_1 + i) \wedge \neg(c_2 \geq 0 \wedge c_2 \leq m) \quad (8.3)$$

ProB's constraint solver finds no solution for Eqs. (8.1) and (8.2), but does so for Eq. (8.3): $m = 127, c_0 = 0, c_1 = 64, c_2 = 128$. One can see that the constraint solver has instantiated the parameter of the event in such a way as to violate the invariant. ProB's classical model checker on the other hand "blindly" enumerates all 64 possible successor states. Using breadth-first search, the counterexample is found after having generated 325 states and 12420 transitions; taking $\sim 1.5s$ whereas BMC finds the counterexample with $k = 2$, i. e., after three calls to the constraint solver and $\sim 1s$.

A depth-first search may generate a long counterexample of up to 127 steps, depending in which order the successors are processed. ProB in this case actually processes the successors with the larger i values first; leading to a counterexample of length 4 after generating 324 states and 323 transitions. The state space is shown in Fig. 8.1, the corresponding counterexample is shown in Fig. 8.2.

The larger the branching-factor, the better BMC becomes as compared to explicit state model checking. When the number of possible parameter values becomes unbounded, e. g., supposing the `incby` event had no upper bound on i, BMC is often the only practical solution.

ProB gives the user the opportunity to set an upper-bound on the number of successor states per event for the explicit model checker. In consequence, exhaustive model checking is then not possible but counterexamples can still be found.

Next, we extend the transition relation to either assert a property after every step or assert its negation:

Definition 8.2.2. For a predicate p we define T_p and $T_p^\neg$ by

$$T_p(x, x') = \bigvee_{evt \in Events} (BA_{evt}(x, x') \wedge p(x'))$$

$$T_p^\neg(x, x') = \bigvee_{evt \in Events} (BA_{evt}(x, x') \wedge proven_{evt,p}(x') \wedge \neg unproven_{evt,p}(x'))$$

For $k \geq 1$, the proven conjuncts of p can be used to strengthen the constraint:

$$BMC(p, k) = I(s_0) \wedge p(s_0) \wedge \bigwedge_{i=0}^{k-2} T_p(s_i, s_{i+1}) \wedge T_p^\neg(s_{k-1}, s_k) \tag{8.4}$$

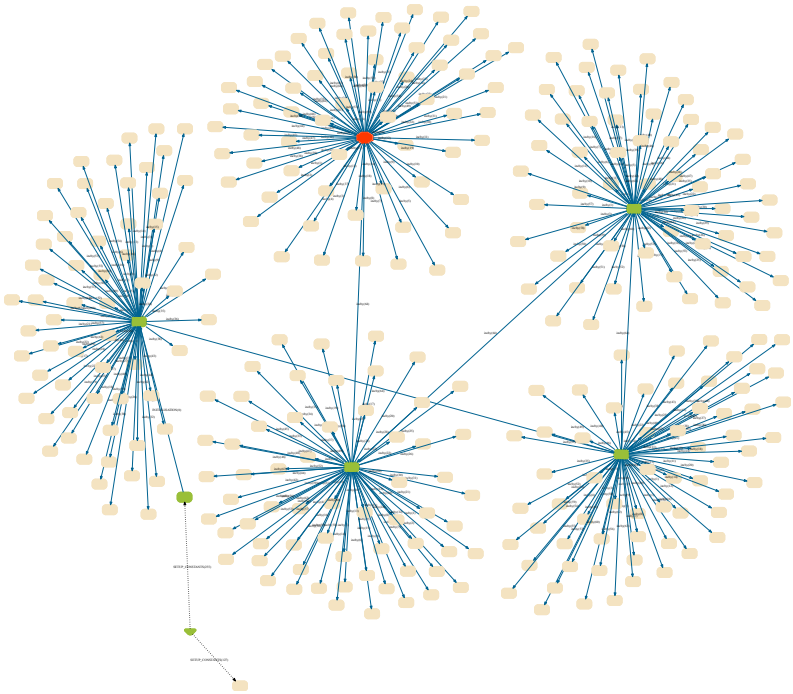


Figure 8.1: State Space of Explicit State Model Checking

Figure 8.2: Counterexample Found by BMC

For the example machine in Listing 7.1, we have that for $k = 0$ the constraint remains unchanged, but for $k = 1$ and $k = 2$ we obtain:

$$
\begin{aligned}
BMC(Inv, 0) = \quad & m \in \{127, 255\} \wedge c_0 = 0 \wedge \neg(c_0 \leq m) \\
BMC(Inv, 1) = \quad & m \in \{127, 255\} \wedge c_0 = 0 \\
& \wedge \exists\, i.\, (i \in 1..64 \wedge c_1 = c_0 + i) \wedge c_1 \geq 0 \wedge \neg(c_1 \leq m) \\
BMC(Inv, 2) = \quad & m \in \{127, 255\} \wedge c_0 = 0 \\
& \wedge \exists\, i.\, (i \in 1..64 \wedge c_1 = c_0 + i) \wedge c_1 \geq 0 \wedge c_1 \leq m \\
& \wedge \exists\, i.\, (i \in 1..64 \wedge c_2 = c_1 + i) \wedge c_2 \geq 0 \wedge \neg(c_2 \leq m)
\end{aligned}
$$

Remember that $unproven_{evt,p}(s_k)$ evaluates to true if all conjuncts of p have been proven to hold after the execution of evt. Hence, for completely proven events $\neg unproven_{evt,p}(s_k)$ is false and the corresponding disjunct in $T_p^{\neg}$ is obviously unsatisfiable. However, we cannot remove such completely proven events from the first $k-1$ steps as they might contribute to the path to a violation of p, using another final event. The usage of proof information thus only simplifies occurring constraints but does not lead to a reduced state space.

Another BMC approach is to use the test-case generation algorithm from [165], using $\neg p$ as target predicate. In contrast to the BMC technique above, the transition predicate is not monolithic, and the algorithm builds up a tree of feasible paths. We have extended the algorithm from [165] to also use $\neg unproven_{evt,p}$ instead of $\neg p$, where *evt* is the last event of any given path.

The algorithm optionally uses a static enabling analysis to filter out infeasible paths before calling the solver. Another difference to classical BMC is that the test-case generation algorithm is able to detect the exhaustiveness of test paths under certain conditions. In consequence, it can be used to verify the correctness of models as well. In the remainder of this chapter we refer to this algorithm as BMC*.

8.2.2 k-Induction

k-Induction as introduced in Section 7.3.4 is a mixture of BMC and proof by induction. The *Base* constraint is equal to the one in BMC: $Base(p,k) = BMC(p,k)$. Hence, we can include proof information in the same fashion and simply reuse the optimized constraint Eq. (8.4) from Section 8.2.1. For the inductive step, we can again use $T_p^\neg$ for the last step, to only look for violations of *unproven* parts of p. Following Algorithm 7.1, we also know that all intermediate states must satisfy the property p; this we can encode using T_p, leading to the definition

$$Step(p,k) = Axm \wedge p(s_0) \wedge \bigwedge_{0 \le i < j \le k} s_i \neq s_j \wedge \bigwedge_{i=0}^{k-1} T_p(s_i, s_{i+1}) \wedge T_p^\neg(s_k, s_{k+1}).$$

As in BMC, we cannot remove before-after-predicates from the first steps. Constraints are simplified but the search space is not reduced.

So far, we have always immediately added the constraint $\bigwedge_{0 \le i < j \le k} s_i \neq s_j$, ensuring that no loops occur for the induction step. It can also be added on demand, once a loop occurs in the counterexample to the inductiveness, as has been suggested in [74] by Eén and Sörensson. With the reduced constraint

$$tStep(p,k) = Axm \wedge p(s_0) \wedge \bigwedge_{i=0}^{k-1} T_p(s_i, s_{i+1}) \wedge T_p^\neg(s_k, s_{k+1}),$$

Algorithm 8.1 shows how one could only prevent loops if needed, i. e., at positions in the program's trace where they did indeed occur. This should prevent the inequalities from piling up in the underlying solver, enabling it to handle more involved constraints. Following [74], we call this approach *temporal induction*, or *t-Induction* for short.

Algorithm 8.1: t-Induction

```
   Data: Property p
   Result: true iff P holds
 1 procedure boolean t-Induction(p)
 2     k := 0
 3     loops := true
 4     while true do
 5         if Base(p, k) satisfiable then
 6             return false
 7         end if
 8         while true do
 9             if tStep(p, k) ∧ loops unsatisfiable then
10                 return true
11             end if
12             if tStep(p, k) ∧ loops satisfiable with s_i = s_j in model then
13                 loops := loops ∧ s_i ≠ s_j
14             else
15                 break
16             end if
17         end while
18         k := k + 1
19     end while
```

8.2.3 IC3

The first change to incorporate proof support takes place in the main loop of IC3. When implemented as suggested by Bradley in [34], IC3 features a special case for 0-step and 1-step reachability of a property violation as explained above. This is shown in line 2 of Algorithm 7.2. The query on line 2 can be changed in the same way we did for BMC and k-Induction. After splitting the transition relation and adding proof information we obtain:

$$sat(I(s) \wedge \neg p(s)) \vee sat(I(s) \wedge T_p^{\neg}(s, s'))$$

The key point where adding proof assistance improves the performance however is inside the *strengthen* procedure of IC3. The original version is given in Algorithm 7.3. Inside, the algorithm tries to find a state included in F_k that has a successor violating the property. With the usual transformation, $F_k(s) \wedge T(s, s') \wedge \neg p(s')$ is turned into $F_k(s) \wedge T_p^{\neg}(s, s')$.

At this point, we can get an additional gain out of using proof information by not relying on the full monolithic transition predicate. The while loop in

Algorithm 7.3 iterates over all counterexamples to inductivity of a given length.

Using an additional loop over all the events of a model, we can reduce $F_k(s) \wedge T_p^{\neg}(s, s')$, which is defined as

$$F_k(s) \wedge \bigvee_{evt \in Events} (BA_{evt}(s, s') \wedge proven_{evt,p}(s') \wedge \neg unproven_{evt,p}(s'))$$

by skipping any event *evt*, if it is proven not to lead to an invariant violation, i. e., if $unproven_{evt,p} = true$. The reduced constraint is

$$F_k(s) \wedge \bigvee_{\substack{evt \in Events \\ unproven_{evt,p} \neq true}} (BA_{evt}(s, s') \wedge proven_{evt,p}(s') \wedge \neg unproven_{evt,p}(s')).$$

The reduced set of events can be computed once upon loading a model. Omitting events leads to a reduced counterexample search space and thus improves the performance of the model checker. In particular cases, the state space might even be rendered finite owing to the reduction.

IC3's two sub routines *inductivelyGeneralize* and *pushGeneralization* afterwards try to prove detected counterexamples spurious by strengthening the frames, adding inductive clauses as needed.

In addition to simplifying counterexample search, we can provide these sub routines with the event that lead to the violating state. This enables simplifying the respective predicates considerably, as we can again only consider the relevant part of the monolithic transition relation.

In IC3, adding proof information has more benefits than just simplifying the occurring constraints. Due to the one-step nature of queries, constraint solving can be skipped altogether if $unproven_{evt,p} = true$.

As no paths are built up explicitly, fully proven events have to be considered only during strengthening. They can safely be omitted during the counterexample search. Thus, including proof information leads to a reduction of the search space.

8.3 Empirical Results

In the following we will empirically assert the performance of the model checking algorithms introduced above. To do so, we will compare to ProB's explicit state model checker. Furthermore, we want to compare different solvers for B predicates as backends for the algorithms.

First, in Section 8.3.1 we will introduce our selection of benchmarks and describe how they were executed. Following, we will compare the symbolic algorithms to ProB in Section 8.3.2 and the different solvers in Section 8.3.3.

Table 8.1: Runtimes in Seconds (Models with Invariant Violations)

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
LargeBranching	-	-	1.79	-	1.61	1.59	-	-	-	-	1.62	1.91
Search	-	-	-	-	-	-	-	-	-	-	-	-
SearchEvents	-	-	2.32	2.34	1.86	2.15	-	-	-	-	1.71	1.74
TravelAgency	1.89	1.85	-	-	14.32	21.39	-	-	-	-	-	-
*ABZ*16_*m*910_*i*1	1.67	1.67	-	-	-	-	-	-	-	-	-	-
*ABZ*16_*m*910_*i*2	3.6	2.97	-	-	-	-	-	-	-	-	-	-
CountersWrong	2.16	2.15	2.29	2.31	2.14	2.2	2.32	2.31	2.3	2.33	2.33	2.34

Table 8.2: Speedup in Percent (Models with Invariant Violations)

Model	MC	BMC	BMC*	k-Induction	t-Induction	IC3
LargeBranching		∞	1.24%			-17.9%
SearchEvents		-0.86%	-15.59%			-1.75%
TravelAgency	2.12%		-49.37%			
*ABZ*16_*m*910_*i*1	0.0%					
*ABZ*16_*m*910_*i*2	17.5%					
CountersWrong	0.46%	-0.87%	-2.8%	0.43%	-1.3%	-0.43%

8.3.1 Experimental Setup

The augmented algorithms have been implemented and are available in ProB. For the empirical evaluation we want to focus on two questions:

- Does the use of proof information considerably improve the performance of symbolic model checking algorithms for B and Event-B?
- Can symbolic model checking algorithms compete with explicit state model checking (MC) as done by ProB?

We apply both the algorithms introduced in Chapter 7 and Section 8.2 as well as ProB's explicit state model checker (MC) to a selection of models, including artificial and real benchmarks using ProB version 1.6.1-beta3.

We use the explicit state model checker with and without proof support as outlined in [24]. For the symbolic model checking algorithms we relied on ProB's internal constraint solver working in a portfolio together with the Z3 backend presented in Chapter 6 and the Kodkod backend presented in [160]. For BMC, we used an iteration limit of 25.

Again, we rely upon the models already introduced in Chapter 4. We dropped some of them as they could not be checked exhaustively by any of the mentioned algorithms. However, we added several crafted models highlighting different aspects of the algorithm's characteristics.

Table 8.3: Runtimes in Seconds (Models without Invariant Violations)

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
ABZ16_m0	1.5	1.4	-	-	1.56	1.5	2.05	2.06	1.63	1.62	1.53	1.44
ABZ16_m1	1.5	1.64	-	-	2.0	1.76	2.01	1.77	1.64	1.63	1.76	1.59
ABZ16_m2	1.67	1.62	-	-	1.94	1.91	1.71	1.47	1.58	1.62	1.59	1.59
ABZ16_m3	1.5	1.6	-	-	1.63	1.6	1.46	1.43	1.46	1.42	1.57	1.53
ABZ16_m4	1.46	1.39	-	-	1.5	1.64	1.53	1.46	1.55	1.45	1.62	1.58
ABZ16_m5	1.88	1.8	-	-	1.59	2.34	-	-	-	-	-	-
ABZ16_m6	1.61	1.56	-	-	1.82	2.45	-	-	-	-	-	-
ABZ16_m7	1.71	1.64	-	-	1.6	2.12	-	-	-	-	-	-
ABZ16_m8	-	-	-	-	46.27	17.46	-	-	-	-	-	-
ABZ16_m9	-	-	-	-	40.47	15.34	-	-	-	-	-	-
ABZ16_m910	-	-	-	-	-	45.8	-	-	-	-	-	-
Coloring	1.61	1.73	2.33	2.18	1.73	1.85	2.19	2.2	2.21	2.2	2.23	2.19
Coloring_40	-	-	-	-	-	-	2.12	2.0	2.13	1.94	2.62	2.66
Counters	-	-	-	-	-	-	1.3	1.23	1.25	1.23	1.28	1.31
f_m0	1.26	1.25	-	-	1.24	1.26	31.66	-	31.75	-	1.31	1.28
f_m1	1.24	1.24	-	-	1.21	1.22	-	-	-	-	1.39	1.41
PM_M0_AAI	1.21	1.23	-	-	1.24	1.18	-	-	-	-	1.47	16.56
PM_M0_AAT	1.2	1.25	-	-	1.16	1.2	-	-	-	-	16.51	1.33
PM_M0_AOO	1.24	1.2	-	-	1.26	1.25	-	-	-	-	16.37	16.44
PM_M0_VOO	1.29	1.2	-	-	1.23	1.24	-	-	-	-	16.84	16.69
PM_M0_VVI	1.35	1.43	-	-	1.28	1.26	-	-	-	-	1.43	16.48
PM_M0_VVT	1.26	1.3	-	-	1.33	1.32	-	-	-	-	1.5	16.67
PM_M1_AOOR	1.31	1.37	-	-	1.31	1.34	-	-	-	-	2.7	16.59
PM_M1_VOOR	1.24	1.27	-	-	1.23	1.52	-	-	-	-	2.89	17.58
PM_M2_AAI	1.3	1.3	-	-	1.25	1.31	-	-	-	-	16.81	32.46
PM_M2_AAT	1.8	1.45	-	-	1.45	1.76	-	-	-	-	32.17	32.03
PM_M2_VVI	1.45	1.42	-	-	1.46	1.6	-	-	-	-	16.77	31.96
PM_M2_VVT	1.31	1.33	-	-	1.42	1.25	-	-	-	-	16.63	31.68
PM_M3_AAI	1.29	1.32	-	-	1.24	1.37	-	-	-	-	18.18	31.96
PM_M3_AAT	1.34	1.35	-	-	1.34	1.33	-	-	-	-	33.47	32.8
PM_M3_VVI	1.35	1.36	-	-	1.32	1.63	-	-	-	-	18.39	32.02
PM_M3_VVT	1.4	1.35	-	-	1.31	1.33	-	-	-	-	18.04	31.71
PM_M4_AAIR	1.32	1.35	-	-	1.31	1.31	-	-	-	-	15.37	31.7
PM_M4_AATR	1.34	1.35	-	-	1.19	1.19	-	-	-	-	14.2	31.57
PM_M4_VVIR	1.32	1.18	-	-	1.48	1.29	-	-	-	-	15.19	31.85
PM_M4_VVTR	1.3	1.33	-	-	1.29	1.3	-	-	-	-	15.73	31.72
R0_GearDoor	1.3	1.28	-	-	-	-	1.44	1.3	1.43	1.31	1.37	1.37
R1_Valve	1.4	1.39	-	-	-	-	7.11	1.34	7.15	1.33	1.46	1.4
R2_Outputs	2.3	2.23	-	-	1.4	1.95	1.46	1.49	1.5	1.47	1.55	1.52
R3_Sensors	3.72	3.4	-	-	1.43	2.47	18.01	1.39	17.96	1.39	1.7	1.46
R4_Handle	30.95	26.71	-	-	-	-	-	-	-	-	-	-
R5_Switch	59.85	55.9	-	-	-	-	17.52	16.24	17.41	15.92	11.27	1.71
R6_Lights	-	-	-	-	-	-	17.59	17.55	17.42	17.57	4.9	1.81

Table 8.4: Speedup in Percent (Models without Invariant Violations)

Model	MC	BMC	BMC*	k-Induction	t-Induction	IC3
ABZ16_m0	6.67%		3.85%	-0.49%	0.61%	5.88%
ABZ16_m1	-9.33%		12.0%	11.94%	0.61%	9.66%
ABZ16_m2	2.99%		1.55%	14.04%	-2.53%	0.0%
ABZ16_m3	-6.67%		1.84%	2.05%	2.74%	2.55%
ABZ16_m4	4.79%		-9.33%	4.58%	6.45%	2.47%
ABZ16_m5	4.26%		-47.17%			
ABZ16_m6	3.11%		-34.62%			
ABZ16_m7	4.09%		-32.5%			
ABZ16_m8			62.26%			
ABZ16_m9			62.1%			
ABZ16_m910			∞			
Coloring	-7.45%	6.44%	-6.94%	-0.46%	0.45%	1.79%
Coloring_40				5.66%	8.92%	-1.53%
Counters				5.38%	1.6%	-2.34%
f_m0	0.79%		-1.61%	∞	∞	2.29%
f_m1	0.0%		-0.83%			-1.44%
PM_M0_AAI	-1.65%		4.84%			-1026.53%
PM_M0_AAT	-4.17%		-3.45%			91.94%
PM_M0_AOO	3.23%		0.79%			-0.43%
PM_M0_VOO	6.98%		-0.81%			0.89%
PM_M0_VVI	-5.93%		1.56%			-1052.45%
PM_M0_VVT	-3.17%		0.75%			-1011.33%
PM_M1_AOOR	-4.58%		-2.29%			-514.44%
PM_M1_VOOR	-2.42%		-23.58%			-508.3%
PM_M2_AAI	0.0%		-4.8%			-93.1%
PM_M2_AAT	19.44%		-21.38%			0.44%
PM_M2_VVI	2.07%		-9.59%			-90.58%
PM_M2_VVT	-1.53%		11.97%			-90.5%
PM_M3_AAI	-2.33%		-10.48%			-75.8%
PM_M3_AAT	-0.75%		0.75%			2.0%
PM_M3_VVI	-0.74%		-23.48%			-74.12%
PM_M3_VVT	3.57%		-1.53%			-75.78%
PM_M4_AAIR	-2.27%		0.0%			-106.25%
PM_M4_AATR	-0.75%		0.0%			-122.32%
PM_M4_VVIR	10.61%		12.84%			-109.68%
PM_M4_VVTR	-2.31%		-0.78%			-101.65%
R0_GearDoor	1.54%			9.72%	8.39%	0.0%
R1_Valve	0.71%			81.15%	81.4%	4.11%
R2_Outputs	3.04%		-39.29%	-2.05%	2.0%	1.94%
R3_Sensors	8.6%		-72.73%	92.28%	92.26%	14.12%
R4_Handle	13.7%					
R5_Switch	6.6%			7.31%	8.56%	84.83%
R6_Lights				0.23%	-0.86%	63.06%

The following small and mostly crafted models were used:

- *LargeBranching*, a crafted benchmark featuring a counterexample reachable in two steps. However, the initialization has numerous outgoing edges. Discovering the counterexample thus heavily relies on picking the right transitions to follow. The model is included to show that the symbolic algorithms are not influenced by this fact.
- *Search*, a classical B model of a binary search algorithm. *SearchEvents* models the same algorithm, but is written in Event-B style with simpler events, i. e., it features operations without involved substitutions. While this leads to simpler constraints, it increases the number of conjuncts due to the increased number of events.
- *TravelAgency*, a classical B model of a travel agency system storing and managing car and room rentals. The model includes an invariant violation.
- *Coloring*, a model of a graph coloring algorithm by Andriamiarina and Méry. The algorithm is specified for all graphs, no concrete graph is given in the model. In contrast, *Coloring40* models the algorithm working on a concrete graph of 40 nodes.
- *f_m0* and *f_m1*, two hybrid models taken from [5].
- *Counters* and *CountersWrong*, two artificial benchmarks featuring two independent counters, one of them bounded and one counting up infinitely. Both models feature an infinite state space. *CountersWrong* has a finite counterexample.

Additionally, we used the following larger models and case studies:

- *R0_Gear_Door*, *R1_Valve*, *R2_Outputs*, *R3_Sensors*, *R4_Handle*, *R5_Switch* and *R6_Lights* are our model [92] for the ABZ 2014 landing gear case study [31].
- The models starting with `ABZ2016` are part of [99], the submission to the ABZ 2016 case study [140] by Hoang, et al.
- The models starting with `PM_` are taken from the model of a pacemaker by Méry and Singh [146].

All benchmarks were run on a MacBook Pro featuring a 2.6 GHz i7 CPU and 8 GB of RAM. We did not run anything in parallel in order to avoid issues due to hyper-threading or scheduling. For each benchmark, a number of results can occur:

- *Verified*, i. e., the model could be model checked exhaustively without an invariant violation being detected.

- *Counterexample found*, i. e., a state violating the invariant was found in the model and a trace to it has been computed.
- *Incomplete*, i. e., no invariant violation has been found but model checking was not exhaustive. This could be due to timeouts or due to ProB being unable to solve occurring constraints. Currently, we do not try to recover. In case of BMC or k-Induction one could for instance try to increase k anyway.

8.3.2 Results

The results are given in Tables 8.1 and 8.3 showing the runtimes on successful benchmarks, i. e., benchmarks where the result is either *Verified* or *Counterexample found*. Table 8.1 features models without invariant violations, while Table 8.3 features models with violations. Tables 8.2 and 8.4 show the speedup achieved by using proof information.

The state space of the *Search* model is too large to be traversed by ProB's explicit state model checker. Unfortunately, the involved substitutions result in complex constraints that cannot be checked by the symbolic algorithms. The effect is increased by the unwinding of the transition system, as complicated constraints start to occur multiple times.

The *SearchEvents* model features simpler substitutions and is thus more suited for symbolic analysis. With and without proof information, the two variants of bounded model checking and IC3 are able to find the counterexample. Both k-Induction and t-Induction are unable to do so. *LargeBranching* paints a similar picture.

The *TravelAgency* model on the other hand has a relatively small state space and can easily be verified using MC. However, it features involved constructs like sequences resulting in complicated constraints. BMC* is the only symbolic technique to find the counterexample, albeit taking much longer than MC.

The *Coloring* model can be checked by all algorithms. Note that even though the model is parametric in the size of the deferred sets *NODES* and *COLORS*, only one configuration is checked. Depending on one of ProB's preferences, the sets are fixed to a pre-defined size. For our benchmarks, we used ProB's default configuration, defining the cardinality of otherwise unspecified deferred sets to be 2.

The more low-level version *Coloring40* is operating on concrete data. A static graph to be colored is given in the model. We assumed the coloring algorithm on given data would be easier to prove correct than the general case. However, only IC3 and the induction based algorithms are able to do so.

There are two reasons for our observation. First, it is caused by the considerably larger graph, consisting of 40 rather than 2 nodes. Once we increase the default set size used by ProB, the model gradually becomes harder to model check. Second, even though the graph is given in the model, the two included axioms `axm8` and `axm9` contain generic properties over all graphs of the given size. For 40 nodes, both are infeasible to check. In consequence, the fact that a concrete graph is given does not impact performance as much as expected.

Abrial's hybrid models can be verified by MC, BMC* and IC3. Here, constraints become considerably more involved with each unwinding of the transition relation done in BMC and k-Induction. IC3 is again able to verify the model due to its local search for counterexamples.

The infinite counters show one of the key limitations of explicit state model checking. Once a state space is infinite, exhaustive analysis is obviously impossible. For the correct model, BMC reaches its iteration limit without detecting an error. Both k-Induction and IC3 are able to analyze the models.

The landing gear model shows that the benefit of using proof information increases with the complexity of the model. As can be seen in Tables 8.1 and 8.4 computation times often go down once proof information is used. For the first refinement steps, IC3 is slightly quicker than explicit state model checking with ProB. However, looking at the later refinement levels shows that the performance of the symbolic model checking algorithms declines. In fact, only IC3 is able to handle the last refinement level *R6_Lights*. The explicit state model checker runs into the one-minute timeout on the same model, while IC3 is able to check it in a matter of seconds. ProB's explicit state model checker terminates successfully after $\sim$ 32 minutes and 1511227 states.

The first five refinement levels of the dialysis machine model are quite simple and can be handled by all algorithms. As they do not include invariant violations, BMC does not find errors. The later refinement levels feature more involved constructs leading to constraints which are more complicated. While this does not hinder the explicit state model checker of ProB, it puts too much stress on the constraint solver. Refinement levels five to eight cannot be validated by any of the symbolic algorithms selected in Chapter 7.

The pacemaker model cannot be checked by the explicit state model checker at all if only ProB's constraint solver is used. However, they can be model checked relying on the Kodkod backend. IC3 and BMC* are able to do so as well, again based on one of the additional backends.

These models show another interesting behavior: For several of them, the inclusion of proof information slows down IC3. We assume that the proof obligations include complicated constraints that are not needed for model checking to succeed, yet they increase the workload on the constraint solvers significantly.

Another example is the *Counters* model which takes a little longer to be model checked by IC3. We suppose that this might be due to the simplicity of the model: No involved constraints occur and any benefit is consumed by the costs of evaluation of the proof information.

In addition, the *Search* model shows a slowdown. We suspect that this is mostly due to the involved substitutions limiting the prover's capabilities. We did not measure a performance decline for BMC and k-Induction.

Again, a constraint that is hard to solve for one solver might be easy for another. We will discuss the influence different solving backends have on the results in the following section.

Summarizing, we can answer the two questions stated at the beginning:

- The inclusion of proof information into the symbolic model checking algorithms can improve the performance. Regarding speedup, we can report from $\sim 0.5\%$ (*CountersWrong* using k-Induction) up to $\sim 92\%$ (*R3_Sensors* with k-Induction). Furthermore, some models can only be checked if proof information is used.

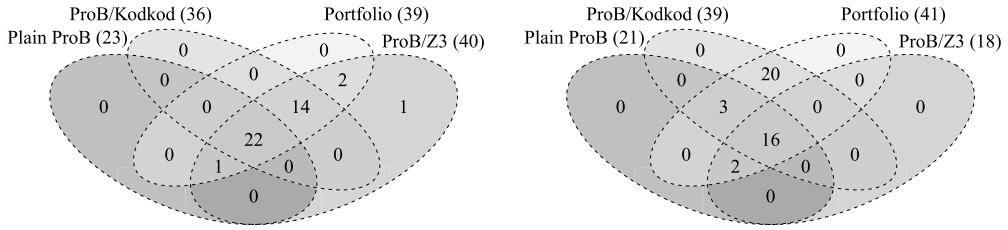
- For some models, albeit small, symbolic techniques can compete with explicit state model checking. Symbolic model checkers allow to verify infinite state spaces which are beyond the scope of ProB's classical model checker.

- Among the symbolic techniques, BMC* was the best for erroneous models, while IC3 was best for correct models.

- However, existing solvers for B and Event-B are still too weak to handle the constraints occurring in larger or more involved models. This currently hinders symbolic model checking efforts. Our current way to overcome this limitation is to rely on different solving backends, each with its own strengths and weaknesses. We tried the same benchmarks relying only on Kodkod as described in [160] and only on ProB combined with Z3 as described in Chapter 6. The results are given in the next section.

8.3.3 Influence of Solvers on Results

In the former section we evaluated the performance of symbolic model checking algorithms in general and the effect of including proof information into them. However, we also wanted to assert the effect the different backends and solvers have on the efficiency of the symbolic algorithms.



(a) IC3 (b) Explicit State MC

Figure 8.3: Influence of Solvers on Results

To do so, we executed all benchmarks again using different configurations:

- ProB alone, without additional solvers or special options.
- ProB together with Kodkod, translating parts of predicates to SAT as shown in [160].
- ProB together with Z3 as outlined in Chapter 6.
- A portfolio approach combining all of the above into a single procedure.

Overall results are shown in Fig. 8.3. As IC3 was the most successful algorithm, we summed up all successful benchmarks of IC3, both with and without including proof information in Fig. 8.3a. We compare it to ProB's explicit state model checker shown in Fig. 8.3b.

As you can see, both solvers influence the number of models that can be checked successfully. In particular, both backends lead to a performance increase when compared to plain ProB. ProB together with Z3 is superior to both ProB alone and ProB with Kodkod.

Surprisingly, the Z3 backend even outperforms the portfolio approach used in the previous section. We assume that this is mostly due to scheduling of the different solvers. The Z3 backend performs well if combined with the symbolic model checking algorithm IC3.

There are a number of models that can additionally be solved using one of the two backends. However, three other models can only be solved by the Z3 backend. This again stresses the point made in Chapter 6: Kodkod and Z3 appear to be orthogonal additions to ProB.

In Table 8.6 we show some of the models that can be solved using Kodkod. Most notably, all the higher refinement levels of the pacemaker model can be checked using BMC* and IC3. As can be seen in Table 8.5, this is not the case with plain ProB.

Table 8.5: Models Checkable Using Plain ProB

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
ABZ16_m0	1.57	1.51	-	-	1.65	1.59	1.63	1.52	1.84	1.46	1.6	1.6
ABZ16_m1	1.89	1.71	-	-	2.32	2.31	1.5	1.49	1.61	1.87	1.83	1.74
ABZ16_m2	2.0	2.1	-	-	2.25	2.27	1.73	1.72	1.72	1.8	1.67	2.02
ABZ16_m3	1.69	1.64	-	-	-	7.07	1.68	1.7	1.49	1.48	1.9	1.66
ABZ16_m4	1.56	1.48	-	-	2.1	2.29	1.5	1.44	1.5	1.54	1.76	1.75
ABZ16_m5	2.28	1.92	-	-	-	-	-	-	-	-	-	-
ABZ16_m6	1.6	1.59	-	-	-	-	-	-	-	-	-	-
ABZ16_m7	1.7	1.64	-	-	-	-	-	-	-	-	-	-
Coloring	1.55	1.5	-	-	-	-	-	-	-	-	-	-
Counters	-	-	-	-	-	-	1.49	1.43	1.45	1.34	1.37	1.37
f_m0	1.32	1.36	-	-	1.35	1.3	-	-	-	-	1.73	1.64
f_m1	1.6	1.54	-	-	1.59	1.76	-	-	-	-	2.01	2.37
PM_M0_AAI	-	-	-	-	-	-	-	-	-	-	1.7	-
PM_M0_AAT	-	-	-	-	-	-	-	-	-	-	-	4.63
PM_M0_AOO	-	-	-	-	-	-	-	-	-	-	-	47.84
PM_M0_VOO	-	-	-	-	-	-	-	-	-	-	-	33.68
PM_M0_VVI	-	-	-	-	-	-	-	-	-	-	1.36	-
PM_M0_VVT	-	-	-	-	-	-	-	-	-	-	1.51	-
R0_GearDoor	1.4	1.4	-	-	-	-	1.47	1.37	1.5	1.55	1.78	1.39
R1_Valve	1.46	1.43	-	-	-	-	7.12	1.37	7.02	1.37	1.47	1.48
R2_Outputs	2.31	2.3	-	-	-	-	1.38	1.37	1.35	1.38	1.45	1.45
R3_Sensors	3.46	3.3	-	-	-	-	59.94	1.41	-	1.44	1.78	1.5
R4_Handle	30.99	26.09	-	-	-	-	-	-	-	-	-	-
R5_Switch	57.91	58.07	-	-	-	-	-	15.93	-	16.17	11.21	1.69
R6_Lights	-	-	-	-	-	-	-	-	-	-	4.65	1.83

Of course a different backend also influences the performance of the explicit state model checker. Kodkod enables it to exhaustively check the pacemaker model as well. The regular BMC, and the two induction based algorithms however do not benefit from using Kodkod at all and remain unable to verify the models. For a table containing the results of all models obtained when applying the Kodkod based solver see Appendix A.3.2. A complete list of results obtained using ProB's internal solver is given in Appendix A.3.1.

Using Z3, Table 8.7 shows a performance increase for the *Coloring40* model. The model can now be checked using the induction based symbolic algorithms and IC3, while still being beyond the scope of MC and the two BMC variants. For the later refinement levels of the ABZ'16 case study we see that they can be checked by both MC and BMC* when the Z3 backend is used. Neither the induction based algorithms nor IC3 can benefit. As for Kodkod, we give the exhaustive list of results in Appendix A.3.3.

However, Z3 does not perform as well if used as a backend to ProB's explicit state model checker. As you can see in Fig. 8.3b, adding Z3 to ProB for explicit state model checking reduces the overall performance. Here, the Z3 backend is

Table 8.6: Models Additionally Checkable Using ProB & Kodkod

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
Coloring	2.83	2.74	-	-	2.68	2.68	-	-	-	-	-	-
PM_M0_AAI	4.08	4.1	-	-	4.21	4.43	-	-	-	-	1.45	2.62
PM_M0_AAT	1.25	1.3	-	-	1.64	1.46	-	-	-	-	1.58	2.78
PM_M0_AOO	1.31	1.36	-	-	1.26	1.29	-	-	-	-	1.43	1.37
PM_M0_VOO	1.22	1.21	-	-	1.22	1.19	-	-	-	-	1.53	1.57
PM_M0_VVI	1.36	1.36	-	-	1.34	1.33	-	-	-	-	1.47	2.63
PM_M0_VVT	1.28	1.25	-	-	1.25	1.25	-	-	-	-	1.49	2.72
PM_M1_AOOR	1.26	1.29	-	-	1.25	1.27	-	-	-	-	1.53	1.51
PM_M1_VOOR	1.31	1.29	-	-	1.26	1.26	-	-	-	-	1.45	1.55
PM_M2_AAI	1.3	1.29	-	-	1.2	1.29	-	-	-	-	1.45	1.44
PM_M2_AAT	1.24	1.21	-	-	1.21	1.2	-	-	-	-	1.54	1.55
PM_M2_VVI	1.27	1.24	-	-	1.21	1.29	-	-	-	-	1.48	1.53
PM_M2_VVT	1.25	1.25	-	-	1.23	1.27	-	-	-	-	1.56	1.66
PM_M3_AAI	1.58	1.32	-	-	1.34	1.55	-	-	-	-	2.21	2.19
PM_M3_AAT	1.28	1.26	-	-	1.22	1.23	-	-	-	-	2.19	2.16
PM_M3_VVI	1.31	1.32	-	-	1.27	1.28	-	-	-	-	2.15	2.1
PM_M3_VVT	1.28	1.45	-	-	1.4	1.24	-	-	-	-	2.24	2.16
PM_M4_AAIR	1.37	1.34	-	-	1.33	1.33	-	-	-	-	2.28	2.18
PM_M4_AATR	1.29	1.32	-	-	1.35	1.26	-	-	-	-	2.3	2.24
PM_M4_VVIR	1.36	1.31	-	-	1.32	1.34	-	-	-	-	2.34	2.25
PM_M4_VVTR	1.36	1.37	-	-	1.31	1.34	-	-	-	-	2.57	2.4
R5_Switch	56.14	52.35	-	-	-	-	2.25	2.26	2.25	2.23	3.59	3.4

outperformed by both plain ProB and by the Kodkod-based backend. Adding Z3 to ProB does not enable ProB to model check any additional model.

As we have argued in Chapter 6, Z3 mostly outperforms the CLP(FD)-based solver when it comes to detection of unsatisfiability. This is often the case with the symbolic algorithms, where the absence of a counterexample has to be proven. In contrast, the explicit state model checker tries to enumerate the successor states: it has to find all valuations in case the transition predicate is satisfiable. Thus, selecting an appropriate backend for ProB highly depends on the framing model checking algorithm. This also needs to be kept in mind when using ProB for data validation or timetabling purposes.

8.4 Related Work

In [24] the authors presented a similar integration of proof information into explicit state model checking algorithms. As is the case with our implementation, the authors report a speedup by not checking invariants known to be true. In contrast to our approach, the use of proof information never slowed down the model checking process.

Table 8.7: Models Additionally Checkable Using ProB & Z3

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
ABZ16_m5	-	4.47	-	-	1.55	2.28	-	-	-	-	-	-
ABZ16_m6	-	9.81	-	-	1.61	2.22	-	-	-	-	-	-
ABZ16_m7	-	9.56	-	-	1.59	2.18	-	-	-	-	-	-
ABZ16_m8	-	-	-	-	32.81	13.93	-	-	-	-	-	-
ABZ16_m9	-	-	-	-	34.11	13.92	-	-	-	-	-	-
ABZ16_m910	-	-	-	-	37.43	18.13	-	-	-	-	-	-
Coloring	2.55	2.46	-	-	-	-	3.75	3.28	2.97	2.7	2.38	2.43
Coloring_40	-	-	-	-	-	-	1.64	1.55	1.66	1.59	28.81	16.89
f_m0	2.38	2.16	-	-	2.2	2.44	3.24	-	3.58	-	2.82	2.79
PM_M0_AAI	-	-	-	-	-	-	-	-	-	-	7.18	9.73
PM_M0_AOO	-	-	-	-	48.96	20.64	-	-	-	-	-	2.97
PM_M0_VOO	-	-	-	-	46.72	20.21	-	-	-	-	-	2.75
PM_M0_VVI	-	-	-	-	-	-	-	-	-	-	1.94	3.07
PM_M0_VVT	-	-	-	-	-	-	-	-	-	-	1.84	2.96
PM_M1_AOOR	-	-	-	-	-	-	-	-	-	-	2.88	3.66
PM_M1_VOOR	-	-	-	-	-	-	-	-	-	-	3.41	3.7
PM_M2_AAI	-	-	-	-	-	-	-	-	-	-	-	9.1
PM_M2_AAT	-	-	-	-	-	-	-	-	-	-	-	9.15
PM_M2_VVI	-	-	-	-	-	-	-	-	-	-	-	9.77
PM_M2_VVT	-	-	-	-	-	-	-	-	-	-	-	9.25
PM_M3_AAI	-	-	-	-	-	-	-	-	-	-	-	9.67
PM_M3_AAT	-	-	-	-	-	-	-	-	-	-	-	9.65
PM_M3_VVI	-	-	-	-	-	-	-	-	-	-	-	9.56
PM_M3_VVT	-	-	-	-	-	-	-	-	-	-	-	10.91
PM_M4_AAIR	-	-	-	-	-	-	-	-	-	-	7.25	9.73
PM_M4_AATR	-	-	-	-	-	-	-	-	-	-	8.46	10.49
PM_M4_VVIR	-	-	-	-	-	-	-	-	-	-	10.33	11.28
PM_M4_VVTR	-	-	-	-	-	-	-	-	-	-	6.58	9.99
R4_Handle	-	-	-	-	-	-	-	-	-	-	-	42.71

Compared with [24], we have added a way to construct proof information within ProB itself, using a bridge to the Atelier B provers and using ProB's proving capabilities as presented in Chapter 4. Of course this takes time and does not always pay off.

In [24], as with BMC and k-Induction, the search space itself is never reduced. Search space reduction through using proof techniques is considered in [170] and [152]. For model checking CTL and LTL properties, proof information can be used as well. In [161], the model checker SMV is coupled with theorem proving techniques. In a similar fashion, [10] combines the Alloy Analyzer with the Athena theorem prover.

Last, instead of using theorem provers to support model checking, one can use model checkers for theorem proving. As described in Chapter 4 and in [132, 113], we have done so using ProB.

8.5 Conclusion and Future Work

Our evaluation shows that using symbolic model checking techniques for B and Event-B models is beneficial: Several counterexamples could only be detected by the symbolic algorithms. Furthermore, some models could be model checked exhaustively. As already outlined in [91], symbolic techniques prove to be a valuable addition to explicit techniques. The techniques are actually also directly applicable to TLA$^+$, via ProB's translation from TLA$^+$ to B [93].

The key weakness of employing symbolic model checking techniques lies within the expressiveness of B and Event-B. Even though constraint solvers and SMT solvers have increased their efficiency by a huge margin, the constraints occurring during symbolic model checking of high-level languages like B are still too involved. Among other abstraction techniques, integrating static (proof) information into the constraints is one way to help. It brings down computation times and sometimes enables successful validation.

We have also been working on strengthening the underlying constraint solver, by integrating SMT solvers such as Z3. As you can see in Section 8.3.3, this leads to a significant performance increase. Still, more improvements need to be achieved until full symbolic verification of B and Event-B models becomes viable.

Regarding the different model checking algorithms, especially IC3 seems promising. In contrast to the other two algorithms, its focus on one step reachability keeps occurring constraints simpler. This makes it more suited for symbolic model checking of high-level languages like B and Event-B. Additionally, the integration of proof information can lead to a reduced search space.

As IC3 has originally been developed for hardware model checking, it is not trivial to lift it to the software world. To do so, we will further investigate IC3 for B together with the abstraction technique CTIGAR in Section 9.3. As our preliminary benchmarks show, CTIGAR seems to be suited for model checking high-level languages such as B and Event-B.

Another direction of future work could be to generate missing proof obligations from the model checking run. Analyzing predicates that lead to a timeout, one could find problematic properties and try to prove them externally or in an independent run. Once the constraint solver gets stuck we could ask an external solver, such as the Atelier B provers or the SMT solvers for Rodin, to proof or disproof further invariants. Afterwards, one could extend the set of properties under consideration.

In summary, we have implemented four symbolic model checking algorithms for B and Event-B and have shown how to integrate proof information to improve the algorithms' performance. Our evaluation shows that bounded model checking can effectively find counterexamples in models with large branching factors and

that IC3 is capable of automatically proving models with infinite state spaces correct. Further research is, however, needed to scale up the symbolic techniques to models with more involved events.

To infinity... and beyond!
Buzz Lightyear, "Toy Story"

9 Abstraction Techniques

9.1 Introduction and Motivation

In Chapters 7 and 8 we have argued that symbolic model checking algorithms can indeed be applied to B and Event-B models. However, the complexity of B constrains the performance of the model checking algorithms. By necessity, we had to incorporate assistance by theorem provers to allow them to cope.

In this chapter, we will evaluate different abstraction techniques [52] that can be employed on top of the symbolic model checking techniques. Our main goal is to lift them to the infinite case. Additionally, abstracting away some complexity of B might enable the algorithms to handle more involved models as we suspected in Section 8.5.

As we have discussed in Section 8.3, IC3 is the most promising algorithm for B and Event-B. Hence, we will only briefly introduce abstraction techniques for BMC and k-Induction in Section 9.2, while looking at IC3 in greater detail in Section 9.3.

9.2 BMC and k-Induction

CEGAR [51] is an abstraction strategy that is conservative, i. e., the abstract model used preserves the functionality of the concrete one, possibly adding additional behavior. The added behavior might introduce spurious counterexamples.

In consequence, if a counterexample occurs, CEGAR systems have to refine the abstraction in order to prove the spuriousness. If refining is impossible, the counterexample detected is in fact not spurious; the system under consideration has been proven faulty.

As the whole process is driven forward by finding counterexamples and reacting to them, it has been named *Counter-Example Guided Abstraction Refinement*, or CEGAR for short. CEGAR can be used with various abstraction and refinement strategies. A CEGAR-like approach can be integrated into BMC and k-Induction as shown in [90].

3-valued abstraction, another abstraction strategy for BMC and k-Induction has been suggested by Orna Grumberg. Her approach in [89] is based on using 3-valued logic. Any predicate or property checked on the abstract model can either be true, false or unknown. Additionally, the approach ensures that if a property is true or false on the abstract model, it holds the same value on the concrete model. In consequence, the approach allows to successfully model check properties, even though the model has only partially been evaluated. Unknown results can be used to abstract away details: If the property in question can still be inferred to be true or false, model checking is successful. Otherwise, if the property is unknown, abstraction is too coarse.

Both *CEGAR* and *3-valued abstraction* could be applied to model check B and Event-B as well. With the techniques introduced in Chapter 3 ProB already reports true or false for predicates. An unknown status could for instance be set in case of timeouts.

For B and Event-B, abstraction could be achieved by using predicate abstraction [87]. Furthermore, B already features involved mechanisms for refinement of models as we have elucidated in Section 2.1.1. The remaining problem is how to construct refined predicates. In case of using predicate abstraction, this is often done by computing Craig interpolants [59]. For first-order logic, interpolants always exist and can be computed efficiently for various common theories such as linear integer arithmetic [88]. However, computing interpolants for B predicates is not as easy, as we have already stated in Section 7.4.

9.3 IC3

So far, different abstraction techniques for integration with IC3 have been suggested:

- Cimatti et al. [48] initially discussed replacing the underlying SAT solver by an SMT solver. In addition, they suggest an algorithm called Tree-IC3 that uses the knowledge about the control flow graph of a program to

improve the monolithic transition predicate. Instead of relying on IC3's linear search, the authors extend IC3 to search through an unwinding of the control flow graph called *ART*, the abstract reachability tree.

In consequence, IC3 computes interpolants for various points in the *ART*, effectively resulting in an algorithm performing lazy abstraction with interpolants as outlined in [143].

- In Section 8.3.2 we already hinted at CTIGAR, an abstraction technique on top of IC3 that seems to be particularly suited for B and Event-B. CTIGAR, introduced in [26] by Birgmeier et al., avoids unrolling transitions in case of counterexamples. Instead, local refinement queries are used to improve upon the current abstraction.

  This approach seems to be particularly suited for B and Event-B as its focus on local instead of global queries avoids constructing and solving involved constraints over paths of states.

- In [49], the authors combine IC3 with predicate abstraction to a CEGAR like approach. In contrast to CTIGAR, the approach computes abtraction refinements using counterexample traces rather than local queries. This leads to more complicated constraints, especially for high-level languages such as B and Event-B.

- A combination of IC3 with interpolation has been suggested in [182]. The authors introduce an algorithm called "Avy" that combines local inductive generalization in the spirit of IC3 with (partial) unrolling of the transition relation to search for counterexamples.

We currently assume that, out of the suggested ones, CTIGAR is best suited for high-level languages such as B. This is mostly due to the fact that it does not rely on the unwinding of the transition relation.

As can be seen in the empirical evaluation in Section 8.3.2, unwinding does often lead to highly involved constraints. Combined with the high-level constructs of languages like B our constraint solvers cannot cope.

Instead of unwinding, CTIGAR follows IC3's idea to focus on one step queries. The key to lifting IC3 to CTIGAR is to introduce an abstract domain consisting of a set of predicate over the state variables.

The original paper [26] suggests using first-order predicates obtained using a Karr analysis [106], which infers linear congruence relations between integer variables and constants of the system under evaluation.

In case of B, we can mine the B machine for predicates, collecting them from the invariant, properties, assertions or guards. For our prototypical implementation, we combine three sets of predicates in our abstract domain:

- The conjuncts of the invariant, including gluing invariants and invariants at lower refinement levels where applicable,
- The conjuncts of a predicate describing the initial states,
- Pairwise inequalities of the form $x < y$ for all integer variables and constants in the model,
- Pairwise inequalities of the form $x \neq y$ for other variables.

The concrete counterexamples to inductiveness obtained in IC3's query $F_i \wedge pre(s) \wedge T(s, s') \rightarrow pre(s')$ in the `inductivelyGeneralize` procedure is lifted to an abstract counterexample to induction by evaluating the predicates in the abstract domain over the state variables in s and their valuations given by $pre(s)$. Of course, *pre* might only partially assign the state variables. In consequence, one would have to find all possible solutions to compute the most precise abstraction. Both approaches were compared in [26]. Apparently the overhead of computing the most precise abstraction is often unjustified. In consequence, we include only the predicates evaluating to true instead of computing the most precise abstraction.

In the presence of concrete and abstract counterexamples, CTIGAR has to react to more situations when handling the queue of counterexamples. According to [26], two queries can now fail:

- The query used to lift a counterexample is assumed to be false. While this is true for the concrete case, in presence of abstraction the query can be satisfiable. This is called a *Lifting Abstraction Failure* by [26].
- Abstraction might cause the presence of spurious transitions. If this is the case, properties which are indeed inductive relative to a certain frame are not discovered as such. Birgmeier, et al. call this a *Consecution Abstraction Failure* [26].

Both have to be treated by refining the abstract domain. In the original paper [26], Craig interpolants are used to derive predicates to add to the domain. As we have discussed in Chapter 7, depending on the chosen backend, interpolants are not available for B. ProB's own kernel and the Kodkod integration [160] are currently unable to compute them. While the SMT solver integration presented in Chapter 6 is able to do so, it does not support all of B and Event-B. We will discuss possible replacements of interpolants in CTIGAR in the following section.

9.3.1 Replacing Interpolants

Below, we will show how interpolants are used in CTIGAR to refine the predicate domain. As stated above, one possible error caused by abstraction is the *Lifting Abstraction Failure*. It occurs upon checking whether a counterexample state predicate cs is inductive, i. e., when evaluating if $cs(s) \wedge T_z(s, s') \Rightarrow ct(s')$ holds. Due to abstraction it might be that the query holds, while the one using the abstract counterexample state predicate $\widehat{cs}$ fails: $\widehat{cs}(s) \wedge T_z(s, s') \Rightarrow ct(s')$ does not hold.

To refine the abstraction domain, [26] suggest to compute an interpolant R, such that $cs(s) \Rightarrow R \wedge R \Rightarrow (T_z(s, s') \Rightarrow ct(s'))$ and add it to the abstraction domain. This resolves the abstraction failure, since the query $\widehat{cs}(s) \wedge R \wedge T_z(s, s') \Rightarrow ct(s')$ holds.

In case an interpolant can be computed efficiently using ProB's SMT solver backend, it is added to the abstraction domain. However, if more involved constructs are used in the model under consideration, this might become impossible.

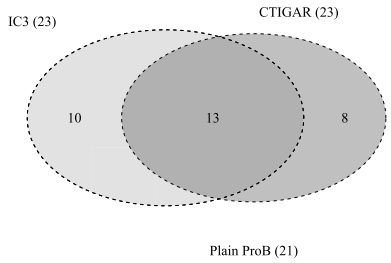
To overcome this limitation, we can use the weakest precondition calculus briefly introduced in Section 2.1.1. We compute $[T_z]c(s')$, the weakest precondition ensuring $c(s')$ holds after executing the transition encoded by T_z.

In contrast to the interpolant, $[T_z]c(s')$ is not guaranteed to be implied by $c(s)$. We can use ProB's proving capabilities sketched in Chapter 4 to try to prove $c(s) \Rightarrow [T_z]c(s')$. If the proof can be performed, we have found a replacement for the interpolant R. Otherwise, we currently have to add the full concrete counterexample to the abstract domain. In the worst case, this continuously reduces the degree of abstraction, degrading CTIGAR to plain IC3.

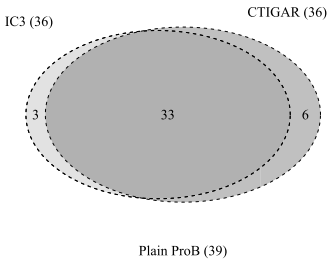
Consecution Abstraction Failures are handled in the same way. However, the queries in question contain IC3's frames F_i, providing a set of hypotheses which can be used by the provers in order to show that the weakest precondition is indeed fulfilled.

9.3.2 Empirical Evaluation

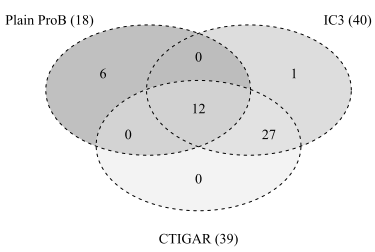
To assert that CTIGAR is suitable for B and Event-B, we implemented the algorithm as outline in Section 9.3. Due to the lack of efficient computation of interpolants, we replaced them by weakest preconditions as described in Section 9.3.1. As benchmarks we relied on the same models already used in the previous chapter. The timings are given in Tables 9.2a and 9.4a, speedup by using proof information is given in Tables 9.2b and 9.4b. The performance with respect to the different backends we used in the previous chapter is again presented using the Venn diagrams in Fig. 9.1.



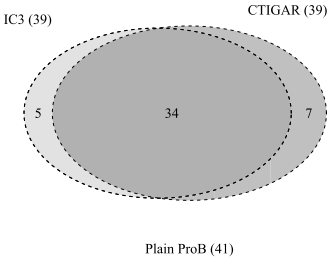
(a) ProB's Solver Alone



(b) Kodkod Backend



(c) Z3 Backend



(d) Portfolio of Solvers

Figure 9.1: CTIGAR and IC3 vs. ProB MC by Backends

As you can see, our prototypical implementation of CTIGAR is currently performing just as the plain IC3 algorithm does. However, if the Z3 backend is used, CTIGAR can only successfully model check a proper subset of the models IC3 can check. Using Z3, as depicted in Fig. 9.1c, the *LargeBranching* model cannot be checked anymore. Again, this is due to the scheduling problems between ProB and Z3 we already discussed in Section 6.5 and Section 8.3.3.

Independent of the backend used, the lack of efficiently computable interpolants impedes CTIGAR playing out its full potential. The ongoing evaluation of the predicates in the abstraction domain over concrete counterexample states is responsible for a performance loss as well. When compared to plain IC3, CTIGAR as implemented in ProB is on average ~ 1.11 times slower if proof information is not used. If it is used, it is ~ 1.07 times slower. For larger models, the difference in performance will increase.

The largest slowdown occurs checking the f_m1 which is ~ 1.48 times slower if CTIGAR is used without proof information. For other models, we can report a speedup. For instance, checking $ABZ16\_m1$, CTIGAR only takes ~ 0.88 of the time IC3 took.

Sadly, using CTIGAR rather than IC3 does not enable ProB to verify new models. The models checkable by ProB's explicit state model checker still result in too complicated constraints. In fact, adding abstraction often made things worse: As concrete values are removed, ProB has to enumerate more variables,

Table 9.1: CTIGAR on Models with Invariant Violations

(a) Runtimes in Seconds

Model	CTIGAR	
use proof info	no	yes
LargeBranching	1.76	1.76
SearchEvents	1.76	1.72
CountersWrong	1.52	1.58

(b) Speedup in Percent

Model	Speedup
LargeBranching	0.0%
SearchEvents	2.27%
CountersWrong	-3.95%

rather than using given values. This leads a larger number of constraints to causing timeouts.

9.4 Related Work

Of course integrating abstraction techniques into model checkers is not only relevant in case of the symbolic model checking algorithms considered in this thesis. In fact, different techniques have been suggested for explicit state model checking or BDD-based model checking as well:

- For the temporal logics LTL and CTL, a combination of model checking and abstract interpretation [58] has been suggested in [52].
- Instead of using abstraction on the model under evaluation, [129] uses partial evaluation to generate a model checker specialized on a certain model. As a result, the specialized checker is not only faster; it might also optimize away certain aspects of the model. In consequence, if the combination of model and model checker permits, the reduction can allow for infinite-state models to be checked by an explicit state model checker.

9.5 Conclusion and Future Work

In this chapter we have shown that CTIGAR is a suitable way to integrate abstraction into a symbolic model checking algorithm for B and Event-B. However, performance does not live up to our expectations. As in the previous chapters, the available backends for solving and proving are often too weak.

In the future, we want to evaluate the effect the different options of CTIGAR have on its performance when applied to B and Event-B. In particular, this includes lazy rather than immediate treatment of counterexamples. We hope

that a more fine-grained selection of which counterexamples to treat immediately can help reduce stress put on the solvers.

Furthermore, benchmarks suggest that our replacement of interpolants by weakest preconditions is not optimal. Another future research direction could be to extend the work done in Chapter 6 in regard to interpolant computation. Of course this can only provide us with interpolants for the subset of B that can be handled by common SMT solvers efficiently. However, handling some counterexamples using interpolants and others using weakest preconditions can already account for a performance increase. We hope to find a way to combine both even for a single counterexample, for instance by computing interpolants only for parts of a predicate.

Table 9.3: CTIGAR on Models without Invariant Violations

(a) Runtimes in Seconds

Model	CTIGAR	
use proof info	no	yes
*ABZ*16_*m*0	1.41	1.4
*ABZ*16_*m*1	1.55	1.6
*ABZ*16_*m*2	1.73	1.58
*ABZ*16_*m*3	1.91	1.83
*ABZ*16_*m*4	1.67	2.0
Coloring	2.95	3.04
*Coloring*_40	3.77	3.86
Counters	1.82	1.69
*f*_*m*0	1.84	1.86
*f*_*m*1	2.07	2.03
*PM*_*M*0_*AAI*	1.96	17.35
*PM*_*M*0_*AAT*	16.57	1.51
*PM*_*M*0_*AOO*	16.69	16.84
*PM*_*M*0_*VOO*	16.86	17.1
*PM*_*M*0_*VVI*	1.63	16.83
*PM*_*M*0_*VVT*	1.41	16.65
*PM*_*M*1_*AOOR*	3.13	16.56
*PM*_*M*1_*VOOR*	3.08	16.96
*PM*_*M*2_*AAI*	16.66	31.93
*PM*_*M*2_*AAT*	32.89	32.79
*PM*_*M*2_*VVI*	17.23	32.94
*PM*_*M*2_*VVT*	17.02	31.59
*PM*_*M*3_*AAI*	18.32	32.02
*PM*_*M*3_*AAT*	33.48	31.86
*PM*_*M*3_*VVI*	18.55	32.23
*PM*_*M*3_*VVT*	19.3	32.49
*PM*_*M*4_*AAIR*	19.8	32.77
*PM*_*M*4_*AATR*	20.02	33.22
*PM*_*M*4_*VVIR*	19.14	31.97
*PM*_*M*4_*VVTR*	15.68	31.88
*R*0_*GearDoor*	1.43	1.37
*R*1_*Valve*	1.44	1.44
*R*2_*Outputs*	1.37	1.39
*R*3_*Sensors*	1.77	1.5
*R*5_*Switch*	12.27	1.68
*R*6_*Lights*	4.8	1.94

(b) Speedup in Percent

Model	Speedup
*ABZ*16_*m*0	0.71%
*ABZ*16_*m*1	-3.23%
*ABZ*16_*m*2	8.67%
*ABZ*16_*m*3	4.19%
*ABZ*16_*m*4	-19.76%
Coloring	-3.05%
*Coloring*_40	-2.39%
Counters	7.14%
*f*_*m*0	-1.09%
*f*_*m*1	1.93%
*PM*_*M*0_*AAI*	-785.2%
*PM*_*M*0_*AAT*	90.89%
*PM*_*M*0_*AOO*	-0.9%
*PM*_*M*0_*VOO*	-1.42%
*PM*_*M*0_*VVI*	-932.52%
*PM*_*M*0_*VVT*	-1080.85%
*PM*_*M*1_*AOOR*	-429.07%
*PM*_*M*1_*VOOR*	-450.65%
*PM*_*M*2_*AAI*	-91.66%
*PM*_*M*2_*AAT*	0.3%
*PM*_*M*2_*VVI*	-91.18%
*PM*_*M*2_*VVT*	-85.61%
*PM*_*M*3_*AAI*	-74.78%
*PM*_*M*3_*AAT*	4.84%
*PM*_*M*3_*VVI*	-73.75%
*PM*_*M*3_*VVT*	-68.34%
*PM*_*M*4_*AAIR*	-65.51%
*PM*_*M*4_*AATR*	-65.93%
*PM*_*M*4_*VVIR*	-67.03%
*PM*_*M*4_*VVTR*	-103.32%
*R*0_*GearDoor*	4.2%
*R*1_*Valve*	0.0%
*R*2_*Outputs*	-1.46%
*R*3_*Sensors*	15.25%
*R*5_*Switch*	86.31%
*R*6_*Lights*	59.58%

The trick to forgetting the big picture is to look at everything close-up.

Chuck Palahniuk, "Lullaby"

10 Applicability to other Formalisms

Even though Chapters 7 to 9 were focused on B and Event-B, the techniques presented in it can easily be applied to other state-based formal languages. However, the commonly used specification languages show different characteristics and limitations when it comes to applying symbolic model checking techniques. In the following section we will briefly describe how symbolic model checking can be done for other specification languages, discussing impacts of both languages and supporting tools.

10.1 TLA$^+$

The techniques developed in this thesis can also be applied on TLA$^+$ models. For example, Listing 10.1 contains a TLA$^+$ version of the running example of the Chapter 8 with the non-inductive invariant $c \geq -2 \wedge c \neq -1$. The accompanying configuration file (not shown here) declares `Invariant1` and `Invariant2` as invariants.

One way to apply our symbolic model checking techniques to TLA$^+$ models is to translate them to B models using an automated translator bundled with ProB [93]. Afterwards, we can use the same constraint solver we did above and benefit from the techniques introduced in Chapters 3 and 6. This way, the model can be proven correct using the k-Induction or IC3 algorithm.

The translation approach however is somewhat limited: it only supports TLA$^+$ models that can be typed translatable, e. g., there is no support for sets whose

Listing 10.1: Simple Example in TLA$^+$

```
---- MODULE Counter64_ok ----
EXTENDS Integers
VARIABLES m, c
Invariant1 == c >= -2
Invariant2 == c # -1
Init ==
        /\ m \in {127, 255}
        /\ c = 0

incby(i) == c' = c + i /\ UNCHANGED <<m>>

Next == \/ \E i \in (1 .. 64) : incby(i)
====
```

members are of different types. Additionally, ProB's constraint solving kernel does not handle all TLA$^+$ expressions efficiently.

TLA$^+$ has a proof system [43] that can be used to perform invariant preservation proofs as we did in Section 8.2. Different backend provers are available. Among others, there exists an integration between TLA$^+$ and Isabelle [44] and a bridge to SMT solvers like Z3 [149]. Judging from our experience in Section 8.3.3 we agree with [147, 149] regarding the expected performance of SMT solvers.

In contrast to B, state changes of a TLA$^+$ model are defined using a single next-state relation. There is no split into different operations or events. Thus, in order to fully apply our techniques directly to TLA$^+$ models, the user has to decompose the next-state predicate. The same approach is commonly used when working with the TLA$^+$ proof system [57].

10.2 VDM

Listing 10.2 shows a simplified version of the running example written in VDM-SL. To allow for execution / debugging in Overture [120], we remove the non-deterministic initialization of m. Instead, we initialize it to 255 by default.

All algorithms introduced in 8.2 can be applied directly to model check specifications written in VDM-SL, the state-based VDM specification language. Explicitly given pre- and post-conditions allow us to set up proof obligations similar to the ones we used for B and Event-B.

Indeed, there appears to be only one drawback: to our knowledge there is currently no integrated constraint solver for VDM expressions. However, two

Listing 10.2: Simple Example in VDM-SL

```
module Counter
exports all
definitions

state State of
  m : nat
  c : nat
inv mk_State(m,c) ==
  c >= 0 and c <= m
init s == s = mk_State(255,0)
end

operations
incby(i:nat)
        ext wr c : nat
        pre i in set {1,...,64}
        post c = c~ + i
end Counter
```

different lines of work are followed in order to provide one. In [121] the authors implement a translation from VDM to Alloy and eventually to SAT. In contrast, in [122] the authors suggest an integration of Overture and ProB.

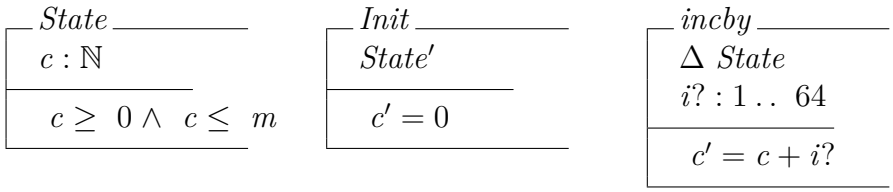
10.3 Z

A possible encoding of the simple running example in Z could look as follows. First, we introduce the constant m using an axiomatic definition:

$$\begin{array}{|l} m : \mathbb{N} \\ \hline m : \{127, 255\} \end{array}$$

Valid states are described using the Z schema called “State”, the initial value of the counter is given in the schema called “Init”.

The `incby` event of the Event-B model is then given by a schema that modifies the state as done in the B version:



ProB is able to load Z files [158]. Again, the approach is based on a systematic translation of Z specifications to an extended version of B. In order to support more Z specifications, Z specific constructs like freetypes have been added to ProB's interpreter. As these features are integrated natively, ProB can be used as prover for Z specifications in the same way as we did for B. Other provers for Z are the commercial ICL Proofpower [108] and Z/EVES [164]. As for VDM, a translation of a subset of Z to Alloy has been suggested [137].

In contrast to TLA$^+$ there is no single next-state relation. Proof obligations could be computed separately for the different schemas.

Slightly adapting the algorithms is still necessary due to a central difference between B and Z in the handling of invariants. When working with Z, one commonly puts the invariants into the enabling condition of a schema. In the example above, this is done by adding Δ *State* to *incby*. In consequence, a schema cannot be executed if the invariant would be violated in the successor state.

In order to apply our symbolic model checking algorithms, one could think about two approaches, both of which can be integrated with proof information as well:

- Follow [158], in which the authors suggested including a specific schema called "Invariant". Comparable to the *State* schema above, it would be used to report the predicates to be verified to ProB.
- Check for deadlocks instead of invariant violations. In this case, the predicate to check would be the disjunct of all enabling conditions of a specification.

Part IV

Conclusion

11

Overall Conclusion and Future Work

In Chapter 1 several goals were stated for this thesis:

1. Strengthen ProB's constraint solver to enable it to solve infinite domain constraint satisfaction problems. This is crucial in order to use it as one of the solving engines for a symbolic model checker.

2. Assess the performance of the extended constraint solver. Based on benchmark data, find ways to improve. Additionally, integrate other solvers or provers if necessary and possible.

3. Evaluate and decide on symbolic model checking algorithms that are suited for B and Event-B.

4. Implement the selected algorithms in ProB using ProB's strengthened constraint solver.

5. Improve performance by integrating abstraction techniques where needed.

6. Perform an empirical evaluation of said algorithms comparing it to ProB's explicit state model checking algorithm.

Goal 1 was achieved by two different lines of work. ProB's internal constraint solver was lifted from the finite to the infinite case using new techniques for enumeration and analysis such as tracking of enumeration and randomized enumeration as introduced in Chapter 3. High-level reasoning over infinite domains was implemented in different ways. For a first approach we implemented the CHR rules given in Chapter 3.

As discussed in Chapter 3, extending our CLP(FD) based solver by high-level reasoning rules written in CHR could in theory have been a solution. However, implementing a somewhat complete high-level reasoner was quickly realized to be a cumbersome task. In addition to the overall complexity, integration between Prolog and CHR was not always optimal. Especially tool support was lacking when it came to debugging.

Yet, the evaluation performed in Chapter 4 exceeded our expectations. Judging by the numbers alone, ProB performed comparably to other well-established provers. A more detailed look at the results revealed that ProB is in fact an orthogonal addition: For certain kinds of proof obligations ProB is superior to both the Atelier B provers as well as the SMT solvers for Rodin.

We also learned that ProB was performing poorly for other kinds of proof obligations. In particular, those involving variables with large or even infinite domains were hard to solve for ProB. Thus, we came back to the idea of improving high-level reasoning.

After trying to use CHR, the focus shifted to integrating an SMT solver into ProB as suggested in goal 2. In Chapter 6 we discussed the theoretical aspects of a translation from B and Event-B to SMT-LIB. Here, especially well-definedness issues proved difficult to solve. We implemented an integration between ProB and Z3 and thoroughly benchmarked it on the benchmarks already used in Chapter 4. The results are promising and show that an integrated approach can be superior to both Z3 and ProB alone.

There is, however, still an issue remaining. The integrated approach was able to discharge proof obligations not dischargeable by the other approaches. Yet, a significant amount could not be discharged anymore. The problem lies within fine-tuning the interaction between ProB and Z3 to avoid that one steals computation time the other is then missing. In the future, we want to implement heuristics to decide when to send a constraint to Z3 and when not to. More involved heuristics could decide on the amount of time allocated to Z3 before ProB takes over again and vice versa.

As an alternative to heuristics, machine learning techniques could in theory be used to classify input predicates and assign them to ProB or Z3 based on some characteristics. An approach to do so has been suggested for instance in [133].

During this thesis, ProB's improved constraint solving capabilities have found several applications aside from symbolic model checking. Most prominently, ProB is used for data validation tasks [95] which greatly benefit from the improvements introduced in Chapter 3. The challenge here is to efficiently handle large relations and sets, e. g., representing track topologies, and at the same time effectively solving constraints and dealing with certain infinite functions which are used to manipulate data. Notable applications come from the railway industry [76] or university timetabling [167]. While the Z3 integration has not been used for data validation, we hope to do so in the future.

Another line of work that has seen performance gains thanks to improving ProB's kernel is the automatic debugging technique introduced in [166]. Here, ProB's solving capabilities are used to synthesize patches for B machines exhibiting errors like invariant violations. Both the improved constraint solver and the Z3 integration have considerably improved the performance of the debugging

assistant. In particular, the ability to detect the absence of counterexamples has been put to use in order to improve user information.

Before writing Chapter 8, the evaluation of different symbolic model checking algorithms given in Chapter 7 was performed. Based on this evaluation we decided upon three algorithms implemented in Chapter 8: bounded model checking, k-Induction and IC3. Furthermore, we incorporated the results of prior proof attempts into the algorithms, effectively reducing the search space by providing static information regarding invariant preservation properties. This lead to performance improvements. Summarizing, this accounts for goals 3, 4 and 6.

The empirical evaluation however has made different shortcomings of our approach obvious:

- The expressiveness of B and Event-B accounts for involved constraints that are hard to handle by the solvers.
- Integrating static information concerning invariant preservation properties is one way of increasing applicability of the symbolic model checking algorithms.
- Especially IC3 seems promising, as its one-step nature avoids constraints from pilling up.

In the future, we need to investigate two different directions. One is to deepen the integration of model checking and proof by generating missing proof obligations from the model checking runs. As suggested, by analyzing predicates that lead to a timeout one could find problematic properties and try to prove them externally or in an independent run.

The second direction is to further examine abstraction techniques like the one in [48], a combination with predicate abstraction as outlined in [49] or CTIGAR [26]. As of now, goal 5 remains open. Our implementation and evaluation did not make clear which abstraction technique is the most promising to be used for B and Event-B. Further case studies will have to be executed to decide between the algorithms.

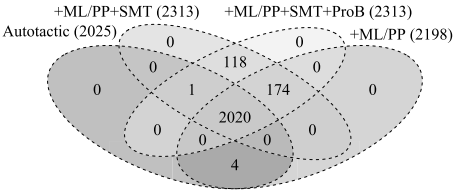
Part V

Appendices

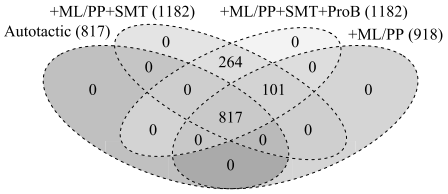
A

Additional Data & Visualizations

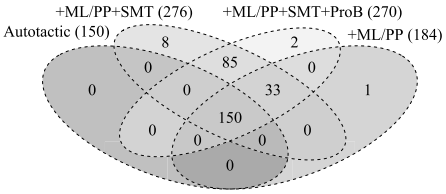
A.1 ProB Disprover: Individual Landing Gears



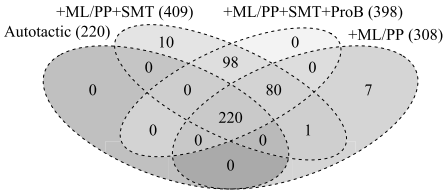
(a) Su and Abrial, Version 1



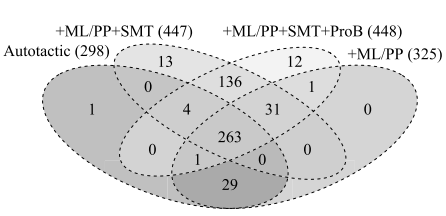
(b) Su and Abrial, Version 2



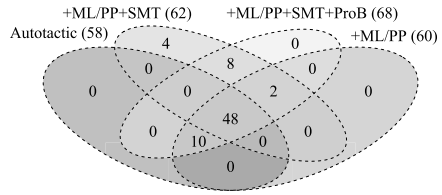
(c) Su and Abrial, Version 3



(d) Mammar and Laleau

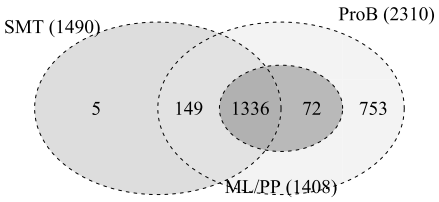


(e) André, Attiogbé and Lanoix

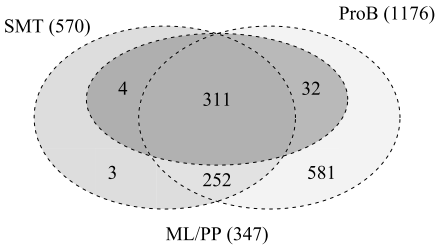


(f) Hansen, Ladenberger, Wiegard, Bendisposto and Leuschel

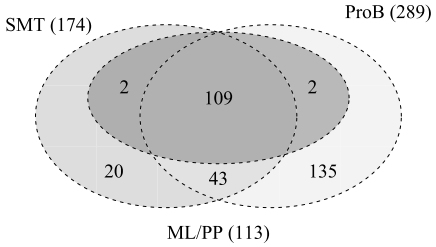
Figure A.1: Landing Gears: Results of Tactics



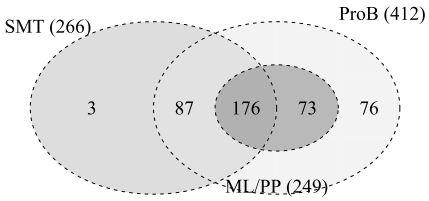
(a) Su and Abrial, Version 1



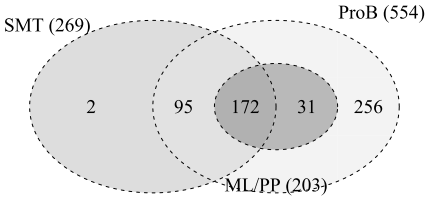
(b) Su and Abrial, Version 2



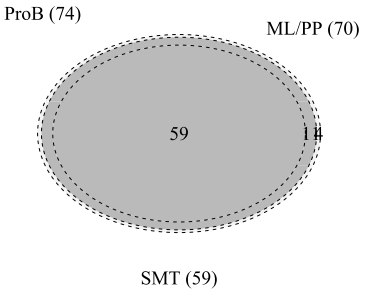
(c) Su and Abrial, Version 3



(d) Mammar and Laleau



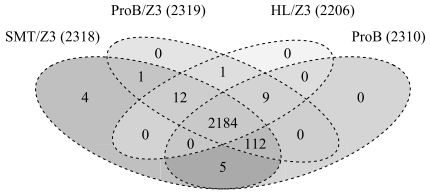
(e) André, Attiogbé and Lanoix



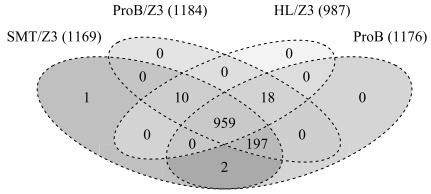
(f) Hansen, Ladenberger, Wiegard, Bendisposto and Leuschel

Figure A.2: Landing Gears: Results of Provers alone

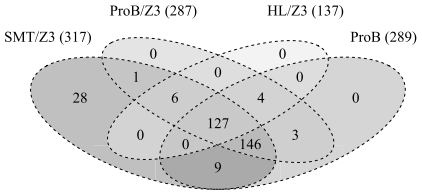
A.2 SMT Solver Integration



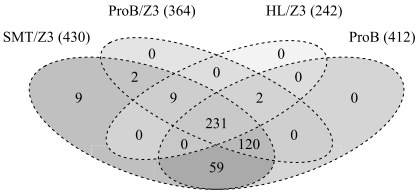
(a) Su and Abrial, Version 1



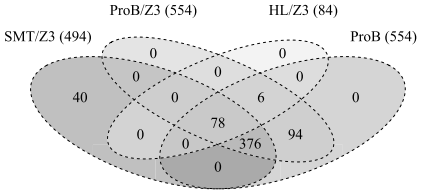
(b) Su and Abrial, Version 2



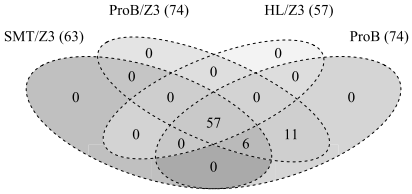
(c) Su and Abrial, Version 3



(d) Mammar and Laleau

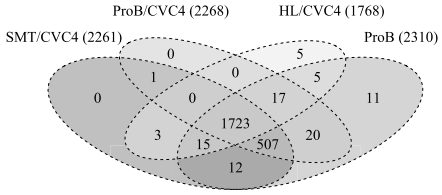


(e) André, Attiogbé and Lanoix

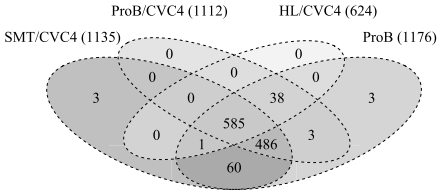


(f) Hansen, Ladenberger, Wiegard, Bendisposto and Leuschel

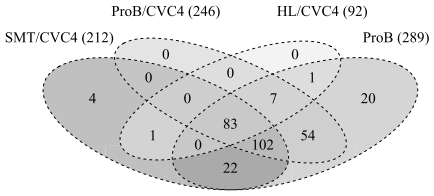
Figure A.3: Z3-based Provers on Landing Gear Models



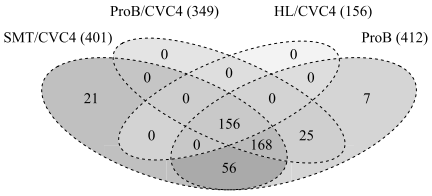
(a) Su and Abrial, Version 1



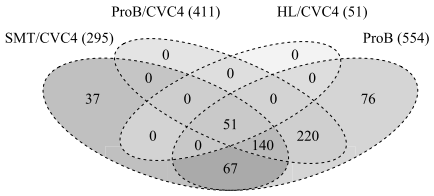
(b) Su and Abrial, Version 2



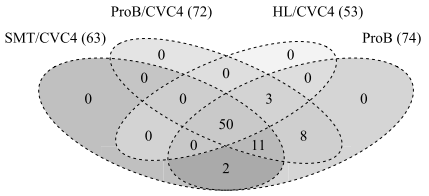
(c) Su and Abrial, Version 3



(d) Mammar and Laleau

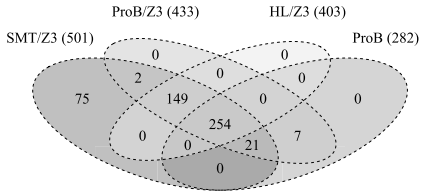


(e) André, Attiogbé and Lanoix

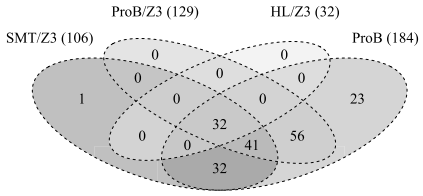


(f) Hansen, Ladenberger, Wiegard, Bendisposto and Leuschel

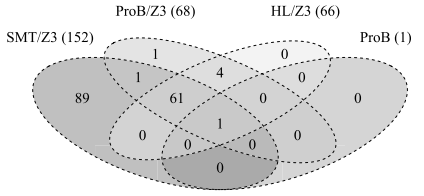
Figure A.4: CVC4-based Provers on Landing Gear Models



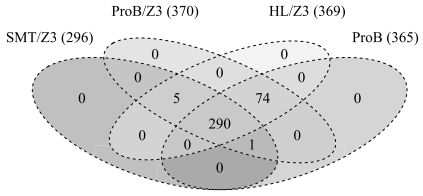
(a) Colley, CAN Bus



(b) Wiegard, Stuttgart 21

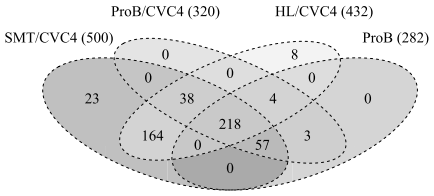


(c) Andriamiarina and Mèry, Graph Coloring Algorithm

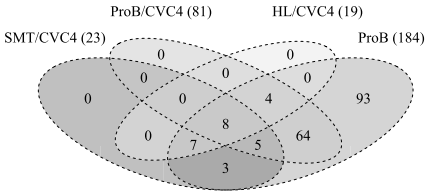


(d) Singh, Pacemaker

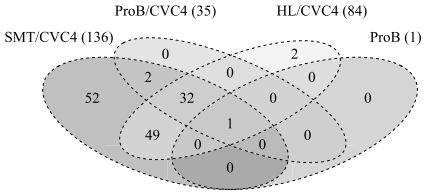
Figure A.5: Z3-based Provers on Miscellaneous Models



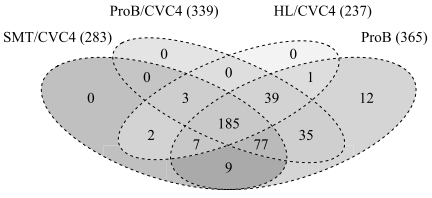
(a) Colley, CAN Bus



(b) Wiegard, Stuttgart 21



(c) Andriamiarina and Mèry, Graph Coloring Algorithm



(d) Singh, Pacemaker

Figure A.6: CVC4-based Provers on Miscellaneous Models

A.3 Symbolic Model Checking

A.3.1 Model Checking Results using plain ProB

Table A.1: Models (without Invariant Violations) Checked Using Plain ProB

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
ABZ16_m0	1.57	1.51	-	-	1.65	1.59	1.63	1.52	1.84	1.46	1.6	1.6
ABZ16_m1	1.89	1.71	-	-	2.32	2.31	1.5	1.49	1.61	1.87	1.83	1.74
ABZ16_m2	2.0	2.1	-	-	2.25	2.27	1.73	1.72	1.72	1.8	1.67	2.02
ABZ16_m3	1.69	1.64	-	-	-	7.07	1.68	1.7	1.49	1.48	1.9	1.66
ABZ16_m4	1.56	1.48	-	-	2.1	2.29	1.5	1.44	1.5	1.54	1.76	1.75
ABZ16_m5	2.28	1.92	-	-	-	-	-	-	-	-	-	-
ABZ16_m6	1.6	1.59	-	-	-	-	-	-	-	-	-	-
ABZ16_m7	1.7	1.64	-	-	-	-	-	-	-	-	-	-
ABZ16_m8	-	-	-	-	-	-	-	-	-	-	-	-
ABZ16_m9	-	-	-	-	-	-	-	-	-	-	-	-
ABZ16_m910	-	-	-	-	-	-	-	-	-	-	-	-
Coloring	1.55	1.5	-	-	-	-	-	-	-	-	-	-
Coloring_40	-	-	-	-	-	-	-	-	-	-	-	-
Counters	-	-	-	-	-	-	1.49	1.43	1.45	1.34	1.37	1.37
f_m0	1.32	1.36	-	-	1.35	1.3	-	-	-	-	1.73	1.64
f_m1	1.6	1.54	-	-	1.59	1.76	-	-	-	-	2.01	2.37
PM_M0_AAI	-	-	-	-	-	-	-	-	-	-	1.7	-
PM_M0_AAT	-	-	-	-	-	-	-	-	-	-	-	4.63
PM_M0_AOO	-	-	-	-	-	-	-	-	-	-	-	47.84
PM_M0_VOO	-	-	-	-	-	-	-	-	-	-	-	33.68
PM_M0_VVI	-	-	-	-	-	-	-	-	-	-	1.36	-
PM_M0_VVT	-	-	-	-	-	-	-	-	-	-	1.51	-
PM_M1_AOOR	-	-	-	-	-	-	-	-	-	-	-	-
PM_M1_VOOR	-	-	-	-	-	-	-	-	-	-	-	-
PM_M2_AAI	-	-	-	-	-	-	-	-	-	-	-	-
PM_M2_AAT	-	-	-	-	-	-	-	-	-	-	-	-
PM_M2_VVI	-	-	-	-	-	-	-	-	-	-	-	-
PM_M2_VVT	-	-	-	-	-	-	-	-	-	-	-	-
PM_M3_AAI	-	-	-	-	-	-	-	-	-	-	-	-
PM_M3_AAT	-	-	-	-	-	-	-	-	-	-	-	-
PM_M3_VVI	-	-	-	-	-	-	-	-	-	-	-	-
PM_M3_VVT	-	-	-	-	-	-	-	-	-	-	-	-
PM_M4_AAIR	-	-	-	-	-	-	-	-	-	-	-	-
PM_M4_AATR	-	-	-	-	-	-	-	-	-	-	-	-
PM_M4_VVIR	-	-	-	-	-	-	-	-	-	-	-	-
PM_M4_VVTR	-	-	-	-	-	-	-	-	-	-	-	-
R0_GearDoor	1.4	1.4	-	-	-	-	1.47	1.37	1.5	1.55	1.78	1.39
R1_Valve	1.46	1.43	-	-	-	-	7.12	1.37	7.02	1.37	1.47	1.48
R2_Outputs	2.31	2.3	-	-	-	-	1.38	1.37	1.35	1.38	1.45	1.45
R3_Sensors	3.46	3.3	-	-	-	-	59.94	1.41	-	1.44	1.78	1.5
R4_Handle	30.99	26.09	-	-	-	-	-	-	-	-	-	-
R5_Switch	57.91	58.07	-	-	-	-	-	15.93	-	16.17	11.21	1.69
R6_Lights	-	-	-	-	-	-	-	-	-	-	4.65	1.83

Table A.2: Models (with Invariant Violations) Checked Using Plain ProB

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
LargeBranching	-	-	1.83	-	1.64	1.67	-	-	-	-	1.62	1.56
Search	-	-	-	-	-	-	-	-	-	-	-	-
SearchEvents	-	-	1.82	1.8	1.74	1.73	-	-	-	-	1.73	1.72
TravelAgency	2.07	1.83	-	-	13.47	21.26	-	-	-	-	-	-
*ABZ*16_*m*910_*i*1	1.61	1.55	-	-	-	-	-	-	-	-	-	-
*ABZ*16_*m*910_*i*2	1.63	1.6	-	-	-	-	-	-	-	-	-	-
CountersWrong	1.14	1.16	1.23	1.26	1.15	1.16	1.28	1.22	1.22	1.2	1.26	1.34

A.3.2 Model Checking Results using Kodkod

Table A.3: Models (without Invariant Violations) Checked Using ProB & Kodkod

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
ABZ16_m0	1.6	1.51	-	-	1.66	1.57	2.21	2.31	2.48	2.42	2.43	1.93
ABZ16_m1	1.71	1.61	-	-	1.7	1.66	2.93	2.46	2.38	2.39	2.92	2.83
ABZ16_m2	1.55	1.74	-	-	2.03	1.94	2.58	2.4	2.38	2.49	3.06	3.03
ABZ16_m3	1.58	1.64	-	-	-	6.94	2.09	2.06	2.07	2.04	2.74	2.65
ABZ16_m4	1.43	1.36	-	-	1.83	1.8	2.32	2.13	2.21	2.14	2.98	2.8
ABZ16_m5	1.93	1.79	-	-	-	-	-	-	-	-	-	-
ABZ16_m6	1.64	1.93	-	-	-	-	-	-	-	-	-	-
ABZ16_m7	1.74	1.68	-	-	-	-	-	-	-	-	-	-
ABZ16_m8	-	-	-	-	-	-	-	-	-	-	-	-
ABZ16_m9	-	-	-	-	-	-	-	-	-	-	-	-
ABZ16_m910	-	-	-	-	-	-	-	-	-	-	-	-
Coloring	2.83	2.74	-	-	2.68	2.68	-	-	-	-	-	-
Coloring_40	-	-	-	-	-	-	-	-	-	-	-	-
Counters	-	-	-	-	-	-	4.97	4.99	4.94	4.9	5.01	5.07
f_m0	4.14	4.07	-	-	4.02	4.02	-	-	-	-	4.79	4.86
f_m1	4.67	4.93	-	-	4.94	5.36	-	-	-	-	5.55	5.53
PM_M0_AAI	4.08	4.1	-	-	4.21	4.43	-	-	-	-	1.45	2.62
PM_M0_AAT	1.25	1.3	-	-	1.64	1.46	-	-	-	-	1.58	2.78
PM_M0_AOO	1.31	1.36	-	-	1.26	1.29	-	-	-	-	1.43	1.37
PM_M0_VOO	1.22	1.21	-	-	1.22	1.19	-	-	-	-	1.53	1.57
PM_M0_VVI	1.36	1.36	-	-	1.34	1.33	-	-	-	-	1.47	2.63
PM_M0_VVT	1.28	1.25	-	-	1.25	1.25	-	-	-	-	1.49	2.72
PM_M1_AOOR	1.26	1.29	-	-	1.25	1.27	-	-	-	-	1.53	1.51
PM_M1_VOOR	1.31	1.29	-	-	1.26	1.26	-	-	-	-	1.45	1.55
PM_M2_AAI	1.3	1.29	-	-	1.2	1.29	-	-	-	-	1.45	1.44
PM_M2_AAT	1.24	1.21	-	-	1.21	1.2	-	-	-	-	1.54	1.55
PM_M2_VVI	1.27	1.24	-	-	1.21	1.29	-	-	-	-	1.48	1.53
PM_M2_VVT	1.25	1.25	-	-	1.23	1.27	-	-	-	-	1.56	1.66
PM_M3_AAI	1.58	1.32	-	-	1.34	1.55	-	-	-	-	2.21	2.19
PM_M3_AAT	1.28	1.26	-	-	1.22	1.23	-	-	-	-	2.19	2.16
PM_M3_VVI	1.31	1.32	-	-	1.27	1.28	-	-	-	-	2.15	2.1
PM_M3_VVT	1.28	1.45	-	-	1.4	1.24	-	-	-	-	2.24	2.16
PM_M4_AAIR	1.37	1.34	-	-	1.33	1.33	-	-	-	-	2.28	2.18
PM_M4_AATR	1.29	1.32	-	-	1.35	1.26	-	-	-	-	2.3	2.24
PM_M4_VVIR	1.36	1.31	-	-	1.32	1.34	-	-	-	-	2.34	2.25
PM_M4_VVTR	1.36	1.37	-	-	1.31	1.34	-	-	-	-	2.57	2.4
R0_GearDoor	1.27	1.53	-	-	-	-	2.05	2.2	2.08	2.02	2.47	2.29
R1_Valve	1.41	1.58	-	-	-	-	2.12	2.05	2.16	2.23	2.48	2.38
R2_Outputs	2.51	2.39	-	-	-	-	2.13	2.26	2.39	2.34	2.55	2.65
R3_Sensors	3.83	3.26	-	-	-	-	2.22	2.22	2.26	2.18	2.88	2.7
R4_Handle	31.48	25.71	-	-	-	-	-	-	-	-	-	-
R5_Switch	56.14	52.35	-	-	-	-	2.25	2.26	2.25	2.23	3.59	3.4
R6_Lights	-	-	-	-	-	-	2.53	2.43	2.34	2.36	4.46	4.04

Table A.4: Models (with Invariant Violations) Checked Using ProB & Kodkod

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
LargeBranching	-	-	1.76	-	1.58	1.6	-	-	-	-	1.6	1.62
Search	-	-	-	-	-	-	-	-	-	-	-	-
SearchEvents	-	-	-	-	-	-	-	-	-	-	-	-
TravelAgency	2.48	2.29	-	-	17.01	21.73	-	-	-	-	-	-
*ABZ*16_*m*910_*i*1	-	-	-	-	-	-	-	-	-	-	-	-
*ABZ*16_*m*910_*i*2	-	-	-	-	-	-	-	-	-	-	-	-
CountersWrong	1.65	1.59	2.32	2.24	1.64	1.59	2.26	2.42	3.05	2.65	2.86	2.88

A.3.3 Model Checking Results using Z3

Table A.5: Models (without Invariant Violations) Checked Using ProB & Z3

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
ABZ16_m0	1.58	1.51	-	-	1.64	1.67	1.69	1.64	1.8	1.59	1.64	1.85
ABZ16_m1	-	1.91	-	-	2.54	1.91	1.69	1.53	2.25	1.59	1.96	2.43
ABZ16_m2	-	2.03	-	-	2.03	2.18	2.37	1.66	2.4	1.79	3.88	2.15
ABZ16_m3	-	1.8	-	-	1.61	1.83	1.89	1.82	2.04	2.07	4.6	1.99
ABZ16_m4	-	1.7	-	-	1.83	1.87	2.11	1.7	1.96	1.65	4.84	2.34
ABZ16_m5	-	4.47	-	-	1.55	2.28	-	-	-	-	-	-
ABZ16_m6	-	9.81	-	-	1.61	2.22	-	-	-	-	-	-
ABZ16_m7	-	9.56	-	-	1.59	2.18	-	-	-	-	-	-
ABZ16_m8	-	-	-	-	32.81	13.93	-	-	-	-	-	-
ABZ16_m9	-	-	-	-	34.11	13.92	-	-	-	-	-	-
ABZ16_m910	-	-	-	-	37.43	18.13	-	-	-	-	-	-
Coloring	2.55	2.46	-	-	-	-	3.75	3.28	2.97	2.7	2.38	2.43
Coloring_40	-	-	-	-	-	-	1.64	1.55	1.66	1.59	28.81	16.89
Counters	-	-	-	-	-	-	2.22	2.24	2.18	2.2	2.19	2.27
f_m0	2.38	2.16	-	-	2.2	2.44	3.24	-	3.58	-	2.82	2.79
f_m1	2.74	2.67	-	-	2.61	2.64	-	-	-	-	4.51	4.51
PM_M0_AAI	-	-	-	-	-	-	-	-	-	-	7.18	9.73
PM_M0_AAT	-	-	-	-	-	-	-	-	-	-	-	1.77
PM_M0_AOO	-	-	-	-	48.96	20.64	-	-	-	-	-	2.97
PM_M0_VOO	-	-	-	-	46.72	20.21	-	-	-	-	-	2.75
PM_M0_VVI	-	-	-	-	-	-	-	-	-	-	1.94	3.07
PM_M0_VVT	-	-	-	-	-	-	-	-	-	-	1.84	2.96
PM_M1_AOOR	-	-	-	-	-	-	-	-	-	-	2.88	3.66
PM_M1_VOOR	-	-	-	-	-	-	-	-	-	-	3.41	3.7
PM_M2_AAI	-	-	-	-	-	-	-	-	-	-	-	9.1
PM_M2_AAT	-	-	-	-	-	-	-	-	-	-	-	9.15
PM_M2_VVI	-	-	-	-	-	-	-	-	-	-	-	9.77
PM_M2_VVT	-	-	-	-	-	-	-	-	-	-	-	9.25
PM_M3_AAI	-	-	-	-	-	-	-	-	-	-	-	9.67
PM_M3_AAT	-	-	-	-	-	-	-	-	-	-	-	9.65
PM_M3_VVI	-	-	-	-	-	-	-	-	-	-	-	9.56
PM_M3_VVT	-	-	-	-	-	-	-	-	-	-	-	10.91
PM_M4_AAIR	-	-	-	-	-	-	-	-	-	-	7.25	9.73
PM_M4_AATR	-	-	-	-	-	-	-	-	-	-	8.46	10.49
PM_M4_VVIR	-	-	-	-	-	-	-	-	-	-	10.33	11.28
PM_M4_VVTR	-	-	-	-	-	-	-	-	-	-	6.58	9.99
R0_GearDoor	1.79	1.95	-	-	-	-	3.32	1.96	2.8	1.9	3.52	2.17
R1_Valve	4.13	4.23	-	-	-	-	7.37	1.57	7.42	1.57	8.84	2.48
R2_Outputs	-	54.53	-	-	1.41	1.57	1.48	1.34	1.48	1.5	1.52	1.55
R3_Sensors	-	-	-	-	1.45	2.69	-	2.18	-	2.2	36.61	1.74
R4_Handle	-	-	-	-	-	-	-	-	-	-	-	42.71
R5_Switch	-	-	-	-	-	-	-	15.88	-	15.66	-	2.32
R6_Lights	-	-	-	-	-	-	-	-	-	-	24.23	2.31

Table A.6: Models (with Invariant Violations) Checked Using ProB & Z3

Model	MC		BMC		BMC*		k-Induction		t-Induction		IC3	
use proof info	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
LargeBranching	-	-	-	-	1.65	1.66	-	-	-	-	1.83	1.87
Search	-	-	-	-	-	-	-	-	-	-	-	-
SearchEvents	-	-	3.67	4.24	-	-	-	-	-	-	2.43	2.4
TravelAgency	1.96	2.02	-	-	-	-	-	-	-	-	-	-
*ABZ*16_*m*910_*i*1	2.43	11.53	-	-	-	-	-	-	-	-	-	-
*ABZ*16_*m*910_*i*2	2.52	26.4	-	-	-	-	-	-	-	-	-	-
CountersWrong	1.26	1.25	1.48	1.49	1.32	1.35	1.64	1.58	1.63	1.59	1.72	1.74

B

Source Code Listings

B.1 Infinite Domain Constraint Solver

Listing B.1: Interpreter

```
:- module(infinite_domain_solver,[solve_constraint/2,
                                  enum_warning/0]).

:- use_module(high_level_reasoner).

:- use_module(library(clpfd)).
:- use_module(library(lists)).

:- op(870,xfy,'=>').
:- op(820,xfy,'<=>').
:- op(860,xfy,'&').
:- op(850,xfy,'or').

:- dynamic enum_warning/0.

solve_constraint(Constraint,TopLevelVars) :-
    retractall(enum_warning),
    solve(Constraint,ExistsWF,AllWF),
    ground_vars(TopLevelVars,ExistsWF,AllWF).

ground_vars(TopLevelVars,ExistsWF,AllWF) :-
    %alarm(15,throw(time_out),Id,[remove(true)]),
    maplist(enumerate_exists_aux,TopLevelVars),
    ground_waitflags(ExistsWF,AllWF).

solve(A & B,EWF,AWF) :- solve(A,EWF,AWF), solve(B,EWF,AWF).
solve(A or B,EWF,AWF) :- solve(A,EWF,AWF) ; solve(B,EWF,AWF).
solve(A <=> B,EWF,AWF) :-
    solve(A,EWF,AWF), solve(B,EWF,AWF) ;
    solve_not(A,EWF,AWF), solve_not(B,EWF,AWF).
solve(A => B,EWF,AWF) :- solve_not(A,EWF,AWF) ; solve(B,EWF,AWF).
solve(not(A),EWF,AWF) :- solve_not(A,EWF,AWF).
solve(V in D,_,_) :- V in D.
solve(A = B,_,_) :-
    compute_exprs(A,B,AE,BE),
    AE #= BE.
```

```
solve(A >= B,_,_) :-
    compute_exprs(A,B,AE,BE),
    leq(B,A),
    AE #>= BE.
solve(A > B,_,_) :-
    compute_exprs(A,B,AE,BE),
    lt(B,A),
    AE #> BE.
solve(A =< B,_,_) :-
    compute_exprs(A,B,AE,BE),
    leq(A,B),
    AE #=< BE.
solve(A < B,_,_) :-
    compute_exprs(A,B,AE,BE),
    lt(AE,BE),
    AE #< BE.

solve(forall(X,LHS => RHS),_EWF,AWF) :-
    when(ground(AWF),enumerate_forall(X,LHS,RHS)).
solve(exists(X,RHS),EWF,_AWF) :-
    when(ground(EWF),enumerate_exists(X,RHS)).

compute_exprs(A,B,AE,BE) :- compute_expr(A,AE), compute_expr(B,BE).
compute_expr(X,X) :- var(X), !.
compute_expr(X,X) :- number(X), !.
compute_expr(A + B,E) :- !,
    compute_expr(A,AE),
    compute_expr(B,BE),
    E #= AE + BE.
compute_expr(A - B,E) :- !,
    compute_expr(A,AE),
    compute_expr(B,BE),
    E #= AE - BE.
compute_expr(A * B,E) :- !,
    compute_expr(A,AE),
    compute_expr(B,BE),
    E #= AE * BE.
compute_expr(A / B,E) :- !,
    compute_expr(A,AE),
    compute_expr(B,BE),
    E #= AE / BE.
compute_expr(A mod B,E) :- !,
    compute_expr(A,AE),
    compute_expr(B,BE),
    E #= mod(AE,BE).

enumerate_forall(Var,LHS,RHS) :-
    LHS = (_ in Min .. Max),
    % setup of inner constraints: contains a choicepoint
    % to allow for different solutions to inner variables
    solve(LHS & RHS,NewEWF,NewAWF), !,
    ground_waitflags(NewEWF,NewAWF),
```

```
    enumerate_forall_aux(Min,Max,Var).
% exhaustively enumerate infinite domain? -> exception
enumerate_forall_aux(_,sup,_) :- throw(enum_infinite).
enumerate_forall_aux(inf,_,_) :- throw(enum_infinite).
% domain is finite, try all elements
enumerate_forall_aux(Current,Max,Var) :-
    Current =< Max, !,
    try_forall_value(Current,Var), % does not bind Var
    Current2 is Current + 1,
    enumerate_forall_aux(Current2,Max,Var).
enumerate_forall_aux(_,_,_).

try_forall_value(Current,Var) :-
    \+ \+ (Current = Var).

enumerate_exists(Var,RHS) :-
    % setup inner constraints
    solve(RHS,NewEWF,NewAWF), !,
    ground_waitflags(NewEWF,NewAWF),
    enumerate_exists_aux(Var).
enumerate_exists_aux(Var) :-
    fd_size(Var,sup), !,
    % non-exhaustively enumerate infinite domain
    % need to find just one element!
    assert(enum_warning),
    fd_inf(Var,Min), fd_sup(Var,Max),
    enumerate_infinite(Var,0,Min,Max).
enumerate_exists_aux(Var) :-
    indomain(Var).

enumerate_infinite(Var,Cur,_Min,Max) :-
    sup_inf_safe_lt(Cur,Max),
    Var = Cur.
enumerate_infinite(Var,Cur,Min,_Max) :-
    sup_inf_safe_lt(Min,-Cur),
    Var is -Cur.
enumerate_infinite(Var,Cur,Min,Max) :-
    Cur2 is Cur + 1,
    enumerate_infinite(Var,Cur2,Min,Max).

sup_inf_safe_lt(_,sup) :- !.
sup_inf_safe_lt(inf,_) :- !.
suf_inf_safe_lt(X,Y) :- X < Y.

solve_not(A & B,EWF,AWF) :-
    solve_not(A,EWF,AWF) ; solve_not(B,EWF,AWF).
solve_not(A or B,EWF,AWF) :-
    solve_not(A,EWF,AWF), solve_not(B,EWF,AWF).
solve_not(A <=> B,EWF,AWF) :-
    solve(A,EWF,AWF), solve_not(B,EWF,AWF) ;
    solve_not(A,EWF,AWF), solve(B,EWF,AWF).
solve_not(A => B,EWF,AWF) :-
```

```
    solve(A,EWF,AWF), solve_not(B,EWF,AWF).
solve_not(A in Inf..Sup,EWF,AWF) :-
    solve(A < Inf or A > Sup,EWF,AWF).
solve_not(not(A),EWF,AWF) :- solve(A,EWF,AWF).
solve_not(A = B,_,_) :-
    compute_exprs(A,B,AE,BE),
    AE #\= BE.
solve_not(A >= B,EWF,AWF) :- solve(A < B,EWF,AWF).
solve_not(A > B,EWF,AWF) :- solve(A =< B,EWF,AWF).
solve_not(A =< B,EWF,AWF) :- solve(A > B,EWF,AWF).
solve_not(A < B,EWF,AWF) :- solve(A >= B,EWF,AWF).
solve_not(forall(X,LHS => RHS),EWF,AWF) :-
    solve(exists(X,LHS & not(RHS)),EWF,AWF).
solve_not(exists(X,RHS),EWF,_AWF) :-
    when(ground(EWF),\+(enumerate_exists(X,RHS))).

ground_waitflags(E,A) :-
    E = ground, A = ground.
```

Listing B.2: Random Enumeration

```
:- module(random_permutations, [get_num_bits/3, get_masks/3,
                                 random_permutation_element/10]).

get_num_bits(Length,NextPower,NumBits) :-
    NumBitsP4 is ceil(log10(Length) / log10(4)),
    NextPower is 4**NumBitsP4,
    NumBits is (floor(log10(NextPower) / log10(2)) + 1) // 2.

get_masks(HalfNumBits,LeftMask,RightMask) :-
    RightMask is (1 << HalfNumBits) - 1,
    LeftMask is RightMask << HalfNumBits.

random_permutation_element(Index,MaxIndex,From,To,Seed,NumBits,
                           LeftMask,RightMask,
                           RandomElement,NextIndex) :-
    draw_index(Index,MaxIndex,Seed,NumBits,LeftMask,RightMask,From,
               To,DrawnElement,NextIndex),
    % working on a 4^x long interval.
    % thus, we might pick a number that is too large
    % if this happens, we just pick a new one
    % to avoid context switching overhead,
    % this is now done inside the C code
    RandomElement is DrawnElement + From.

draw_index(Idx,MaxIdx,Seed,HalfNumBits,LeftMask,RightMask,
           From,To,Rnd,NextIdx) :-
    draw_index_loop(Idx,MaxIdx,Seed,HalfNumBits,LeftMask,RightMask,
                    From,To,IdxOut,Rnd),
    (IdxOut > MaxIdx -> fail ; NextIdx = IdxOut).

draw_index_loop(Idx,MaxIdx,Seed,HalfNumBits,LeftMask,RightMask,
                From,To,IdxOut,RndOut) :-
```

```
    Left2 is (Idx /\ LeftMask) >> HalfNumBits,
    Right2 is Idx /\ RightMask,
    feistel_rounds(Left2,Right2,Seed,RightMask,Left3,Right3),
    Rnd is (Left3 << HalfNumBits) \/ Right3,
    Idx2 is Idx + 1,
    (Rnd > To - From, Idx2 =< MaxIdx
     -> draw_index_loop(Idx2,MaxIdx,Seed,HalfNumBits,LeftMask,
                        RightMask,From,To,IdxOut,RndOut)
     ;  IdxOut = Idx2, RndOut = Rnd).

feistel_rounds(Left,Right,Seed,RightMask,LeftOut,RightOut) :-
    LeftOut = Right,
    term_hash(Right,Hash),
    RightOut is Left xor Hash /\ RightMask.
```

Listing B.3: CHR Rules

```
:- module(high_level_reasoner, [lt/2, leq/2]).

:- use_module(library(chr)).

:- chr_constraint leq/2, lt/2.
:- chr_constraint eq/3.

reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).

antireflexivity @ lt(X,X) <=> fail.
idempotence     @ lt(X,Y) \ lt(X,Y) <=> true.
transitivity    @ lt(X,Y), leq(Y,Z) ==> lt(X,Z).
transitivity    @ leq(X,Y), lt(Y,Z) ==> lt(X,Z).
transitivity    @ lt(X,Y), lt(Y,Z)  ==> lt(X,Z).
```

C

Own Publications Used in this Thesis

C.1 Outline

This thesis is based on several publications written during my PhD studies. The main matter makes use of four articles already published in different venues. In the sections below, the authors and abstracts of the articles are given and general publication information is presented. The individual contributions of each author are listed. As far as known, impact factors and conference rankings are included in the information.

Furthermore, the significance of each article is discussed. In particular, this includes the contribution of each article towards the overall goal of providing a symbolic model checker for B and Event-B. Last, articles are put into relation to each other.

The golden thread is as follows: The first article "Constraint Logic Programming over Infinite Domains with an Application to Proof" lays the ground for any symbolic analysis as it deals with improvements to ProB's constraint solving capabilities.

The following articles, "From Failure to Proof: The ProB Disprover for B and Event-B" evaluates ProB's constraint solver empirically.

As a result of the evaluation, in order to overcome weaknesses discovered, an integration between ProB and Z3 has been developed in "SMT Solvers for Validation of B and Event-B models".

Last, "Proof Assisted Symbolic Model Checking for B and Event-B" uses the foundations provided by the other articles to back the implementation of different symbolic model checking algorithms.

C.2 Constraint Logic Programming over Infinite Domains with an Application to Proof

3.2.1 Abstract

We present a CLP(FD)-based constraint solver able to deal with unbounded domains. It is based on constraint propagation, resorting to enumeration if all other methods fail. An important aspect is detecting when enumeration was complete and if this has an impact on the soundness of the result. We present a technique which guarantees soundness in the following way: if the constraint solver finds a solution it is guaranteed to be correct; if the constraint solver fails to find a solution it can either return the result "definitely false" in case it knows enumeration was exhaustive, or "unknown" in case it was aborted. The technique can deal with nested universal and existential quantifiers. It can easily be extended to set comprehensions and other operators introducing new quantified variables. We show applications in data validation and proof.

3.2.2 Significance

The article describes our first attempts at lifting ProB's constraint solving kernel from finite to infinite domains. As already stated, this is a precondition for it to be useful as the backend of symbolic model checking algorithms. Besides, the techniques presented in this article also improve ProB's performance when used for data validation tasks.

Regarding user feedback, improved tracking of the enumeration status of different variables can be used for more precise error messages which can aid in resolving performance bottlenecks.

3.2.3 Relation to Other Articles

From Failure to Proof: The ProB Disprover for B and Event-B: The paper evaluates the performance of ProB when used as a prover. Tracking enumerations is crucial to do so: the absence of a counterexample, i. e., a proof, can only be detected if the status of variable enumerations is known. Thus, "From Failure to Proof: The ProB Disprover for B and Event-B" could be seen as an in-depth evaluation of the techniques presented in this article.

SMT Solvers for Validation of B and Event-B models: "Constraint Logic Programming over Infinite Domains with an Application to Proof" and "SMT Solvers for Validation of B and Event-B models" describe two orthogonal approaches to reach the same goal. In both papers, we try to add high-level

reasoning to ProB's kernel, improving propagation and detection of unsatisfiability. Instead of improving ProB's kernel, the article describes how ProB and Microsoft's SMT solver Z3 can be integrated into a single solving procedure.

Proof Assisted Symbolic Model Checking for B and Event-B: As described in the introduction, symbolic model checking algorithms for high-level languages such as B and Event-B put a lot of stress on the underlying constraint solvers and provers. Therefore, improvements like the one described in this article render ProB more suited to be used for the algorithms adapted and developed in "Proof Assisted Symbolic Model Checking for B and Event-B".

3.2.4 Publication Information

The article "Constraint Logic Programming over Infinite Domains with an Application to Proof" [115] was originally published in the "Electronic Proceedings in Theoretical Computer Science" series, to which no impact factor is assigned yet.

The article was presented at the "Workshop on (Constraint) Logic Programming" in Leipzig, Germany on September 12–13, 2016. The workshop itself was part of the "Leipzig Week of Declarative Programming". Before acceptance, all submissions to the workshop went through a full peer review process. No CORE Conference Ranking is available for the WLP workshop.

The authors of this article are Sebastian Krings and Michael Leuschel.

S. Krings' contributions are:

- Enumeration tracking algorithm
- Randomized enumeration
- Main body of the article
- Implementation work inside ProB

M. Leuschel's contributions are:

- Introduction
- Section on related work
- Section on data validation
- General implementation work and benchmarks

C.3 From Failure to Proof: The ProB Disprover for B and Event-B

3.3.1 Abstract

The ProB disprover uses constraint solving to find counterexamples for B proof obligations. As the ProB kernel is now capable of determining whether a search was exhaustive, one can also use the disprover as a prover. In this paper, we explain how ProB has been embedded as a prover into Rodin and Atelier B. Furthermore, we compare ProB with the standard automatic provers and SMT solvers used in Rodin. We demonstrate that constraint solving in general and ProB in particular are able to deal with classes of proof obligations that are not easily discharged by other provers and solvers. As benchmarks, we use medium-sized specifications such as landing gear systems, a CAN bus specification and a railway system. We also present a new method to check proof obligations for inconsistencies, which has helped uncover various issues in existing (sometimes fully proven) models.

3.3.2 Significance

The article describes key extensions to ProB's constraint solving kernel. The main goal was to show that ProB can now be used not only to find counterexamples but detect the absence of them. This has been made possible by observing the enumeration status of different variables as explained in "Constraint Logic Programming over Infinite Domains with an Application to Proof".

The empirical evaluation focuses on proof obligations. However, the extension also enabled ProB's kernel to be used as the backend of symbolic model checking algorithms. This is due to the fact that the constraints occurring in symbolic model checking often resemble those existing in proof attempts, as we will show below.

3.3.3 Relation to Other Articles

Constraint Logic Programming over Infinite Domains with an Application to Proof: Using ProB as a prover has been made possible by the enumeration tracking technique described in "Constraint Logic Programming over Infinite Domains with an Application to Proof". Without the improvements done, ProB could only have been used as a disprover, i. e., to find counterexamples. Aside from obvious cases, the absence of a counterexample could not have been detected.

SMT Solvers for Validation of B and Event-B models: As was mentioned in Section 2.5, SMT solvers have different characteristics than solvers based on constraint logic programming. In "From Failure to Proof: The ProB Disprover for B and Event-B" we learned that certain constraints cannot be solved easily by ProB due to its lack of high-level reasoning. In "Constraint Logic Programming over Infinite Domains with an Application to Proof" we tried to overcome this limitation by introducing CHR rules. However, this proved to be and error-prone task. Instead, in "SMT Solvers for Validation of B and Event-B models", we integrate ProB with Z3 in order to overcome the weaknesses we spotted.

Proof Assisted Symbolic Model Checking for B and Event-B: This article implements different symbolic model checking algorithms for B and Event-B. To do so it makes use of ProB's extended constraint solver and prover in two ways. Most prominently, constraints occurring in the algorithms are solved using ProB's kernel and the tools it integrates with. Additionally, ProB's prover is used to compute static information later added to constraints in order to ease solving.

3.3.4 Publication Information

The article "From Failure to Proof: The ProB Disprover for B and Event-B" [113] was originally published in the "Lecture Notes in Computer Science" series by Springer, Berlin, Germany. No impact factor is assigned to the series.

The article was presented at the "Software Engineering and Formal Methods" conference in York, UK on September 7–11, 2015. It was given a "Best Paper Award". Before acceptance, all submissions to the conference went through a full peer review process. The conference is rated "B — good conference, and well regarded in a discipline area" according to the CORE Conference Ranking.

The authors of this article are Sebastian Krings, Jens Bendisposto and Michael Leuschel.

S. Krings' contributions are:

- Empirical evaluation including results and conclusion
- Discussion of differences between ProB disprover and SMT solvers
- Rodin integration of the disprover
- Implementation work needed to lift ProB from disprover to prover

J. Bendisposto's and M. Leuschel's contributions are:

- Co-developers of initial disprover (without support for proof)
- Detection of inconsistencies in hypothesis

- Section on constraint solving kernel
- Atelier B integration together with ClearSy and Alstom
- Implementation work needed to lift ProB from disprover to prover
- Helpful discussion during research and implementation

C.4 SMT Solvers for Validation of B and Event-B models

3.4.1 Abstract

We present an integration of the constraint solving kernel of the ProB model checker with the SMT solver Z3. We apply the combined solver to B and Event-B predicates, featuring higher-order datatypes and constructs like set comprehensions. To do so we rely on the finite set logic of Z3 and provide a new translation from B to Z3, better suited for constraint solving. Predicates can then be solved by the two solvers working hand in hand: constraints are set up in both solvers simultaneously and (intermediate) results are transferred. We thus combine a constraint logic programming based solver with a DPLL($\mathcal{T}$) based solver into a single procedure. The improved constraint solver finds application in many validation tasks, from animation of implicit specifications, to test case generation, bounded and symbolic model checking on to disproving of proof obligations. We conclude with an empirical evaluation of our approach focusing on two dimensions: comparing low and high-level encodings of B as well as comparing pure ProB to ProB combined with Z3.

3.4.2 Significance

The integration of Z3 into ProB is strengthening the constraint solving kernel. As we have seen in different empirical evaluations, ProB's kernel and Z3 are orthogonal extensions to each other. They are able to solve different kinds of constraints. In symbolic model checking, we often have to deal with constraints that are not particularly suited for solving using constraint logic programming. Thanks to Z3 these cannot be solved by the integrated solver, enabling symbolic model checking of a wider range of models.

3.4.3 Relation to Other Articles

Constraint Logic Programming over Infinite Domains with an Application to Proof: As stated above, techniques like enumeration tracking are needed to use ProB as a prover. This holds true when using ProB as an SMT solver as well.

From Failure to Proof: The ProB Disprover for B and Event-B: The article made several weaknesses of ProB obvious. Furthermore, the comparison to the existing SMT integration for Rodin showed that in using SMT solvers can be a better approach to constraint solving than constraint logic programming. The SMT translation however had its own choice of restrictions we outlined. In "SMT Solvers for Validation of B and Event-B models" we thus learned our lessons and suggested an integrated approach combining both in order to overcome the limitations spotted before.

Proof Assisted Symbolic Model Checking for B and Event-B: The article shows that although ProB has been strengthened it is often still too weak to handle full-blown symbolic model checking. Integrating SMT solvers is a further mean towards a more capable constraint solver. Thus, it can enable symbolic model checking of models that ProB could not handle without.

3.4.4 Publication Information

The article "SMT Solvers for Validation of B and Event-B models" [117] was originally published in the "Lecture Notes in Computer Science" series by Springer, Berlin, Germany. No impact factor is assigned to the series.

The article was presented at the "Integrated Formal Methods" conference in Reykjavík, Island on June 1–5, 2016. Before acceptance, all submissions to the conference went through a full peer review process. The conference is rated "B — good conference, and well regarded in a discipline area" according to the CORE Conference Ranking.

The authors of this article are Sebastian Krings and Michael Leuschel.

S. Krings' contributions are:

- Implementation of integration between ProB and Z3
- Small experiments in paper and empirical evaluation
- Theoretical aspects of translation

M. Leuschel's contributions are:

- Additions to related and future work

- Updates to discussion
- Description of ProB's kernel
- Discussion of different implementation and translation techniques

C.5 Proof Assisted Symbolic Model Checking for B and Event-B

3.5.1 Abstract

We have implemented various symbolic model checking algorithms, like BMC, k-Induction and IC3 for B and Event-B. The high-level nature of B and Event-B accounts for complicated constraints arising in these symbolic analysis techniques. In this paper we suggest using static information stemming from proof obligations to simplify occurring constraints. We show how to include proof information in the aforementioned algorithms. Using different benchmarks we compare explicit state to symbolic model checking as well as techniques with and without proof assistance. In particular for models with large branching factor, e. g., due to complicated data values being manipulated, the symbolic techniques fare much better than explicit state model checking. The inclusion of proof information results in further clear performance improvements.

3.5.2 Significance

In the article, we implement the symbolic model checking algorithms selected in Section 7.4 for B and Event-B. An empirical evaluation is performed on a selection of models. Additionally, it introduces proof assistance, a technique that uses static information stemming from successful proof attempts to strengthen constraints and make them easier to handle.

3.5.3 Relation to Other Articles

Constraint Logic Programming over Infinite Domains with an Application to Proof: All symbolic algorithms introduced in the paper make heavy use of ProB's improved constraint solver. Without the additional improvements, several benchmarks could not be run successfully.

From Failure to Proof: The ProB Disprover for B and Event-B: The proof capabilities can be used to recompute static information where missing and to perform further proofs if needed. Thus, a tighter integration between

model checking on the one side and prove as well as constraint solving on the other side seems desirable.

SMT Solvers for Validation of B and Event-B models: We hope that the integration of SMT solvers into ProB's kernel will allow for more models to be checked symbolically. During implementation and evaluation for "Proof Assisted Symbolic Model Checking for B and Event-B" we saw that the relies upon the negation of properties under consideration often leads to highly involved constraints. We hope that integrating Z3 enables us to handle these better than relying ProB alone.

3.5.4 Publication Information

The article "Proof Assisted Symbolic Model Checking for B and Event-B" [116] was originally published in the "Lecture Notes in Computer Science" series by Springer, Berlin, Germany. No impact factor is assigned to the series.

An extended version has been submitted to the Elsevier journal "Science of Computer Programming".

The article was presented at the "ASM, Alloy, B, TLA, VDM, Z (ABZ)" conference in Linz, Austria on May 23–27, 2016. Before acceptance, all submissions to the conference went through a full peer review process. No CORE Conference Ranking is available for the ABZ conference.

The authors of this article are Sebastian Krings and Michael Leuschel.

S. Krings' contributions are:

- Implementation and description of algorithms
- Discussion and conclusion
- Empirical evaluation

M. Leuschel's contributions are:

- Improvements to running example and in-depth comparison to ProB
- Discussion related to non-inductive invariants

Bibliography

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[4] J.-R. Abrial and L. Mussat. On Using Conditional Definitions in Formal Theories. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Proceedings ZB*, volume 2272 of *LNCS*, pages 242–269. Springer, 2002.

[5] J.-R. Abrial, W. Su, and H. Zhu. Formalizing hybrid systems with Event-B. In *Proceedings ABZ*, volume 7316 of *LNCS*, pages 178–193. Springer, 2012.

[6] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment. In *Proceedings CHARME*, volume 3725 of *LNCS*, pages 254–268. Springer, 2005.

[7] André, Attiogbé, and Lanoix. Modelling and Analysing the Landing Gear System: a Solution with Event-B/Rodin. Solution Submitted to the Case Study Track of ABZ-2014.

[8] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[9] K. R. Apt and P. Zoeteweij. An Analysis of Arithmetic Constraints on Integer Intervals. *Constraints*, 12(4):429–468, 2007.

[10] K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating Model Checking and Theorem Proving for Relational Reasoning. In R. Berghammer, B. Möller, and G. Struth, editors, *Relational and Kleene-Algebraic Methods in Computer Science*, volume 3051 of *LNCS*, pages 21–33. Springer, 2004.

[11] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Proceedings SPIN*, pages 146–162. Springer, 2006.

[12] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings IJCAI*, IJCAI, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[13] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[14] K. Bansal, A. Reynolds, C. Barrett, and C. Tinelli. A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT. In *Proceedings IJCAR*, volume 9706 of *LNAI*, pages 82–98. Springer, 2016.

[15] S. Bardin, P. Herrmann, and F. Perroud. An Alternative to SAT-Based Approaches for Bit-Vectors. In J. Esparza and R. Majumdar, editors, *Proceedings TACAS*, volume 6015 of *LNCS*, pages 84–98. Springer, 2010.

[16] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[17] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. Rajamani, editors, *Proceedings CAV*, volume 3576 of *LNCS*, pages 20–23. Springer, 2005.

[18] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[19] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, Feb. 2009.

[20] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, July 2007.

[21] J. Bendisposto. *Directed and Distributed Model Checking of B-Specifications*. PhD thesis, Heinrich-Heine-University Duesseldorf, 2015.

[22] J. Bendisposto, S. Krings, and M. Leuschel. Who watches the watchers: Validating the ProB Validation Tool. In *Proceedings F-IDE*, volume 149 of *EPTCS*. Electronic Proceedings in Theoretical Computer Science, 2014.

[23] J. Bendisposto, P. Körner, M. Leuschel, J. Meijer, J. van de Pol, H. Treharne, and J. Whitefield. Symbolic Reachability Analysis of B through ProB and LTSmin. In *Proceedings iFM*, volume 9681 of *LNCS*. Springer, 2016.

[24] J. Bendisposto and M. Leuschel. Proof assisted model checking for B. In *Proceedings ICFEM*, volume 5885 of *LNCS*, pages 504–520. Springer, 2009.

[25] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[26] J. Birgmeier, A. Bradley, and G. Weissenbacher. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In A. Biere and R. Bloem, editors, *Proceedings CAV*, volume 8559 of *LNCS*, pages 831–848. Springer, 2014.

[27] J. Black and P. Rogaway. Ciphers with Arbitrary Finite Domains. In B. Preneel, editor, *Proceedings CT-RSA*, volume 2271 of *LNCS*, pages 114–130. Springer, 2002.

[28] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT Solvers. In *Proceedings CADE*, volume 6803 of *LNCS*, pages 116–130. Springer, 2011.

[29] J. C. Blanchette and T. Nipkow. *Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.

[30] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Comput.*, 45(9):993–1002, Sept. 1996.

[31] F. Boniol and V. Wiels. The Landing Gear System Case Study. In *Proceedings ABZ: The Landing Gear Case Study*, volume 433 of *CCIS*, pages 1–18. Springer, 2014.

[32] R. T. Boute. The Euclidean Definition of the Functions Div and Mod. *ACM Trans. Program. Lang. Syst.*, 14(2):127–144, Apr. 1992.

[33] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Proceedings CADE*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.

[34] A. R. Bradley. SAT-based Model Checking Without Unrolling. In *Proceedings VMCAI*, volume 6538 of *LNCS*, pages 70–87, Berlin, Heidelberg, 2011. Springer.

[35] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang. An Incremental Approach to Model Checking Progress Properties. In *Proceedings FMCAD*, FMCAD '11, pages 144–153, Austin, Texas, 2011. FMCAD Inc.

[36] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In S. Kowalewski and A. Philippou, editors, *Proceedings TACAS*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.

[37] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986.

[38] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, Sept. 1992.

[39] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[40] D. Cantone and C. Zarba. A New Fast Tableau-Based Decision Procedure for an Unquantified Fragment of Set Theory. In R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *LNCS*, pages 126–136. Springer, 2000.

[41] M. Carlsson and T. Fruehwirth. *Sicstus PROLOG User's Manual 4.3*. Books On Demand - Proquest, 2014.

[42] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proceedings PLILP*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.

[43] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA$^+$ Proof System. *CoRR*, abs/0811.1914, 2008.

[44] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying Safety Properties with the TLA$^+$ Proof System. In *Proceedings IJCAR*, volume 6173 of *LNCS*, pages 142–148. Springer, 2010.

[45] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental Formal Verification of Hardware. In *Proceedings FMCAD*, FMCAD '11, pages 135–143, Austin, TX, 2011. FMCAD Inc.

[46] J. Christiansen and S. Fischer. EasyCheck — Test Data for Free. In *Proceedings FLOPS*, volume 4989 of *LNCS*, pages 322–336. Springer, 2008.

[47] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings CAV*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[48] A. Cimatti and A. Griggio. Software model checking via IC3. In *Proceedings CAV*, volume 7358 of *LNCS*, pages 277–293. Springer, 2012.

[49] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Proceedings TACAS*, volume 8413 of *LNCS*, pages 46–61. Springer, 2014.

[50] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *Proceedings TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[51] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[52] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, Sept. 1994.

[53] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[54] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and Complexity of Bounded Model Checking. In *Proceedings VMCAI*, volume 2937, pages 85–96. Springer, 2004.

[55] ClearSy. *Atelier B 4.1 Release Notes*. Aix-en-Provence, France, 2009. Available at `http://www.atelierb.eu/`.

[56] ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2016. Available at `http://www.atelierb.eu/`.

[57] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA+ proofs. In *Proceedings FM*, volume 7436 of *LNCS*, pages 147–154, 2012.

[58] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings POPL*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[59] W. Craig. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *The Journal of Symbolic Logic*, 22(3):250–268, Sept. 1957.

[60] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[61] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design*, pages 45–52, 2009.

[62] L. de Moura, H. Rueß, and M. Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *Proceedings CADE*, volume 2392 of *LNCS*. Springer, 2002.

[63] D. Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In *Proceedings ASM*, volume 5977 of *LNCS*, pages 217–230. Springer, 2010.

[64] D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ*, volume 7316 of *LNCS*, pages 194–207. Springer, 2012.

[65] D. Delahaye, C. Dubois, C. Marché, and D. Mentré. The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations. In *Proocedings ABZ*, volume 8477 of *LNCS*, pages 290–293. Springer, 2014.

[66] I. Dobrikov and M. Leuschel. Optimising the ProB Model Checker for B using Partial Order Reduction. In *Proceedings SEFM*, volume 8702 of *LNCS*, pages 220–234, 2014.

[67] M. Dowson. The Ariane 5 Software Failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, Mar. 1997.

[68] C. Dross, S. Conchon, and A. Paskevich. Reasoning with Triggers. Research Report RR-7986, INRIA, June 2012.

[69] G. J. Duck. SMCHR: satisfiability modulo constraint handling rules. *CoRR*, abs/1210.5307, 2012.

[70] B. Dutertre. Yices 2.2. In *Proceedings CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, July 2014.

[71] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT Solvers for Rodin. In *Proceedings ABZ*, volume 7316 of *LNCS*, pages 194–207. Springer, 2012.

[72] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 94, Part 2(0):130–143, 2014.

[73] N. Eén, A. Mishchenko, and R. Brayton. Efficient Implementation of Property Directed Reachability. In *Proceedings FMCAD*, FMCAD '11, pages 125–134, Austin, Texas, 2011. FMCAD Inc.

[74] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[75] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B)*, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[76] J. Falampin, H. Le-Dang, M. Leuschel, M. Mokrani, and D. Plagge. Improving railway data validation with ProB. In *Industrial Deployment of System Engineering Methods*, pages 27–44. Springer, 2013.

[77] F. Ferrandi, M. Rendine, and D. Sciuto. Functional Verification for SystemC Descriptions Using Constraint Solving. In *Proceedings DATE*, DATE '02, pages 744–, Washington, DC, USA, 2002. IEEE Computer Society.

[78] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research.* Oliver and Boyd, 3 edition, 1953.

[79] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[80] A. Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, Italy, 2010.

[81] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *The Journal of Logic Programming*, 37(1–3):95–138, 1998.

[82] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[83] A. Fröhlich, G. Kovásznai, and A. Biere. Efficiently Solving Bit-Vector Problems Using Model Checkers. In *Proceedings SMT*, pages 6–15, 2013.

[84] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Proceedings CAV*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[85] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *Proceedings CAV*, volume 697 of *LNCS*, pages 438–449. Springer, 1993.

[86] C. P. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings AAAI and IAAI*, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[87] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[88] A. Griggio, T. T. H. Le, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic. In *Proceedings TACAS*, volume 6806 of *LNCS*, pages 143–157. Springer, 2011.

[89] O. Grumberg. 3-Valued Abstraction for (Bounded) Model Checking. In *Proceedings ATVA*, volume 5799 of *LNCS*, pages 21–21. Springer, 2009.

[90] A. Gupta and O. Strichman. Abstraction Refinement for Bounded Model Checking. In K. Etessami and S. K. Rajamani, editors, *Proceedings CAV*, volume 3576 of *LNCS*, pages 112–124. Springer, 2005.

[91] S. Hallerstede and M. Leuschel. Constraint-based deadlock checking of high-level specifications. *TPLP*, 11(4–5):767–782, 2011.

[92] D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. Validation of the ABZ Landing Gear System Using ProB. In *Proceedings ABZ: The Landing Gear Case Study*, volume 433 of *CCIS*, pages 66–79. Springer, 2014.

[93] D. Hansen and M. Leuschel. Translating TLA$^+$ to B for validation with ProB. In *Proceedings iFM*, volume 7321 of *LNCS*, pages 24–38. Springer, 2012.

[94] D. Hansen and M. Leuschel. Translating B to TLA$^+$ for validation with TLC. In *Proceedings ABZ*, volume 8477 of *LNCS*, pages 40–55, 2014.

[95] D. Hansen, D. Schneider, and M. Leuschel. Using B and ProB for Data Validation Projects. In *Proceedings ABZ*, volume 9675 of *LNCS*. Springer, 2016.

[96] Z. Hassan, A. R. Bradley, and F. Somenzi. Incremental, Inductive CTL Model Checking. In *Proceedings CAV*, volume 7358 of *LNCS*, pages 532–547, Berlin, Heidelberg, 2012. Springer.

[97] Z. Hassan, A. R. Bradley, and F. Somenzi. Better generalization in IC3. In *Proceedings FMCAD*, FMCAD '13, pages 157–164, Austin, Texas, 2013. FMCAD Inc.

[98] P. V. Hentenryck, H. Simonis, and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artif. Intell.*, 58(1-3):113–159, 1992.

[99] T. S. Hoang, C. Snook, L. Ladenberger, and M. Butler. Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-Simulation. In *Proceedings ABZ*, volume 9675 of *LNCS*, pages 360–375. Springer, 2016.

[100] C. Hoare. The Verifying Compiler, a Grand Challenge for Computing Research. In *Proceedings VMCAI*, volume 3385 of *LNCS*, pages 78–78. Springer, 2005.

[101] J. M. Howe and A. King. A Pearl on SAT Solving in Prolog. In *Proceedings FLOPS*, volume 6009 of *LNCS*, pages 165–174, Berlin, Heidelberg, 2010. Springer.

[102] J. M. Howe and A. King. A Pearl on SAT and SMT Solving in Prolog. *Theor. Comput. Sci.*, 435:43–55, June 2012.

[103] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[104] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Proceedings ICLP*, pages 196–218. MIT Press, 1987.

[105] R. Jhala and K. L. McMillan. Interpolant-Based Transition Relation Approximation. *Logical Methods in Computer Science*, 3(4), 2007.

[106] M. Karr. Affine Relationships Among Variables of a Program. *Acta Inf.*, 6(2):133–151, June 1976.

[107] R. Kindermann, T. Junttila, and I. Niemelä. SMT-Based Induction Methods for Timed Systems. In *Proceedings FORMATS*, volume 7595 of *LNCS*, pages 171–187. Springer, 2012.

[108] D. King, R. Arthan, and I. Winnersh. Development of practical verification tools. *ICL Systems Journal*, 11:106–122, 1996.

[109] J. C. King. *A Program Verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.

[110] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[111] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 1997.

[112] G. Kovásznai, A. Fröhlich, and A. Biere. bv2epr: A Tool for Polynomially Translating Quantifier-Free Bit-Vector Formulas into EPR. In *Proceedings CADE*, volume 7898 of *LNCS*, pages 443–449. Springer, 2013.

[113] S. Krings, J. Bendisposto, and M. Leuschel. From Failure to Proof: The ProB Disprover for B and Event-B. In *Proceedings SEFM*, volume 9276 of *LNCS*. Springer, 2015.

[114] S. Krings and M. Leuschel. Inferring Physical Units in Formal Models. *Software & Systems Modeling*, pages 1–23, 2015.

[115] S. Krings and M. Leuschel. Constraint Logic Programming over Infinite Domains with an Application to Proof. In *Proceedings WLP*, volume 234 of *EPTCS*. Electronic Proceedings in Theoretical Computer Science, 2016.

[116] S. Krings and M. Leuschel. Proof Assisted Symbolic Model Checking for B and Event-B. In *Proceedings ABZ*, volume 9675 of *LNCS*. Springer, 2016.

[117] S. Krings and M. Leuschel. SMT Solvers for Validation of B and Event-B models. In *Proceedings iFM*, volume 9681 of *LNCS*. Springer, 2016.

[118] P. Körner. An Integration of ProB and LTSmin. Master's thesis, Heinrich-Heine-Universität Düsseldorf, 2017.

[119] L. Ladenberger, J. Bendisposto, and M. Leuschel. Visualising Event-B Models with B-Motion Studio. In *Proceedings FMICS*, volume 5825 of *LNCS*, pages 202–204. Springer, 2009.

[120] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The Overture Initiative - Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, Jan. 2010.

[121] K. Lausdahl. Translating VDM to Alloy. In *Proceedings iFM*, volume 7940 of *LNCS*, pages 46–60, 2013.

[122] K. Lausdahl, H. Ishikawa, and P. Larsen. Interpreting Implicit VDM Specifications using ProB. *University of Newcastle-upon-Tyne. Computing Science. Technical Report Series*, CS-TR-1446:1–15, 1 2015. Proceedings of the 12th Overture Workshop.

[123] C. Lecoutre. *Constraint Networks*. ISTE, 2010.

[124] C. Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[125] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings ACM SAC*, pages 615–622, 2009.

[126] M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, and D. Plagge. From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In J.-L. Boulanger, editor, *Formal Methods Applied to Complex Systems: Implementation of the B Method*, chapter 14, pages 427–446. Wiley ISTE, Hoboken, NJ, 2014.

[127] M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[128] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, Feb. 2008.

[129] M. Leuschel and T. Massart. Infinite State Model Checking by Abstract Interpretation and Program Specialisation. In *Selected Papers from the 9th International Workshop on Logic Programming Synthesis and Transformation*, volume 1817 of *LNCS*, pages 62–81. Springer, 2000.

[130] N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[131] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, and M. Deters. An efficient SMT solver for string constraints. *Formal Methods in System Design*, 48(3):206–234, 2016.

[132] O. Ligot, J. Bendisposto, and M. Leuschel. Debugging Event-B Models using the ProB Disprover Plug-in. *Proceedings AFADL*, Juni 2007.

[133] A. Loreggia, Y. Malitsky, H. Samulowitz, and V. A. Saraswat. Deep Learning for Algorithm Portfolios. In *Proceedings AAAI*, pages 1280–1286, 2016.

[134] M. Luby and C. Rackoff. How to Construct Pseudorandom Permutations from Pseudorandom Functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[135] I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In *Proceedings ABZ*, volume 5977 of *LNCS*. Springer, 2010.

[136] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[137] P. Malik, L. Groves, and C. Lenihan. Translating Z to Alloy. In *Proceedings ABZ*, volume 5977 of *LNCS*, pages 377–390. Springer, 2010.

[138] A. Mammar and R. Laleau. Modeling a Landing Gear System in Event-B. In *Proceedings ABZ: The Landing Gear Case Study*, volume 433 of *CCIS*, pages 80–94. Springer, 2014.

[139] Z. Manna and C. Zarba. Combining Decision Procedures. In B. K. Aichernig and T. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 381–422. Springer, 2003.

[140] A. Mashkoor. The Hemodialysis Machine Case Study. In *Proceedings ABZ*, volume 9675 of *LNCS*, pages 329–343. Springer, 2016.

[141] P. J. Matos, B. Fischer, and J. Marques-Silva. A Lazy Unbounded Model Checker for Event-B. In *Proceedings ICFEM*, volume 5885 of *LNCS*, pages 485–503. Springer, 2009.

[142] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Proceedings CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[143] K. L. McMillan. Lazy Abstraction with Interpolants. In *Proceedings CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[144] K. L. McMillan. Interpolants from Z3 Proofs. In *Proceedings FMCAD*, FMCAD '11, pages 19–27, Austin, TX, 2011. FMCAD Inc.

[145] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[146] D. Méry and N. K. Singh. Formal Specification of Medical Systems by Proof-Based Refinement. *ACM Trans. Embed. Comput. Syst.*, 12(1):15:1–15:25, Jan. 2013.

[147] S. Merz and H. Vanzetto. Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers. In *Proceedings LPAR*, volume 7180 of *LNCS*, pages 289–303. Springer, 2012.

[148] S. Merz and H. Vanzetto. Encoding TLA$^+$ set theory into many-sorted first-order logic. *CoRR*, abs/1508.03838, 2015.

[149] S. Merz and H. Vanzetto. Encoding TLA ˆ+ + into Many-Sorted First-Order Logic. In *Proceedings ABZ*, volume 9675 of *LNCS*, pages 54–69. Springer, 2016.

[150] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, apr 1984.

[151] M. Moskal. Programming with Triggers. In *Proceedings SMT*, SMT '09, pages 20–29, New York, NY, USA, 2009. ACM.

[152] O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In E. Brinksma, W. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 1–16. Springer, 1995.

[153] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, Oct. 1979.

[154] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0. *JSAT*, 9:53–58, 2015.

[155] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer, Berlin, Heidelberg, 2002.

[156] Nist. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology, Building 101, Room A1000 Gaithersburg, MD 20899-0001, May 2002.

[157] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 377–390, London, UK, UK, 1994. Springer.

[158] D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *LNCS*, pages 480–500. Springer, 2007.

[159] D. Plagge and M. Leuschel. Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more. *STTT*, 12(1):9–21, 2 2010.

[160] D. Plagge and M. Leuschel. Validating B, Z and TLA$^+$ using ProB and Kodkod. In D. Giannakopoulou and D. Méry, editors, *Proceedings FM*, LNCS, pages 372–386. Springer, 2012.

[161] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 82–97, London, UK, UK, 2001. Springer.

[162] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *The Journal of Symbolic Logic*, 62:981–998, 9 1997.

[163] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'07, pages 346–362, Berlin, Heidelberg, 2007. Springer.

[164] M. Saaltink. The Z/EVES system. In *Proceedings ZUM 1997*, volume 1212 of *LNCS*, pages 72–85, Berlin, Heidelberg, 1997. Springer.

[165] A. Savary, M. Frappier, and M. Leuschel. Model-Based Robustness Testing in Event-B using Mutation. In R. Calinescu and B. Rumpe, editors, *Proceedings SEFM'15*, volume 9276 of *LNCS*, pages 132–147, 2015.

[166] J. Schmidt, S. Krings, and M. Leuschel. Interactive Model Repair by Synthesis. In *Proceedings ABZ*, volume 9675 of *LNCS*. Springer, 2016.

[167] D. Schneider, M. Leuschel, and T. Witt. Model-Based Problem Solving for University Timetable Validation and Improvement. In *Proceedings FM*, volume 9109 of *LNCS*, pages 487–495. Springer, 2015.

[168] S. Schneider. *The B-method: An Introduction*. Cornerstones of computing. Palgrave, 2001.

[169] T. Schuele and K. Schneider. Bounded model checking of infinite state systems. *Formal Methods in System Design*, 30(1):51–81, 2007.

[170] N. Shankar. Combining Theorem Proving and Model Checking through Symbolic Analysis. In *CONCUR'00: Concurrency Theory*, number 1877 in LNCS, pages 1–16, State College, PA, aug 2000. Springer.

[171] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Proceedings FMCAD*, FMCAD '00, pages 127–144. FMCAD Inc, Austin, Texas, 2000.

[172] C. Spermann and M. Leuschel. ProB gets Nauty: Effective Symmetry Reduction for B and Z Models. In *Proceedings TASE*, pages 15–22. IEEE, June 2008.

[173] A. Stephenson, L. LaPiana, D. Mulville, P. Rutledge, F. Bauer, D. Folta, G. Dukeman, R. Sackheim, and P. Norvig. Mars Climate Orbiter - Mishap Investigation Report - Phase I Report, 1999.

[174] A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. In *Automated Reasoning*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

[175] W. Su and J.-R. Abrial. Aircraft Landing Gear System: Approaches with Event-B to the Modeling of an Industrial System. In *Proceedings ABZ: The Landing Gear Case Study*, volume 433 of *CCIS*, pages 19–35. Springer, 2014.

[176] A. Sülflow, U. Kühne, R. Wille, D. Große, and R. Drechsler. Evaluation of SAT-like Proof Techniques for Formal Verification of Word-Level Circuits. In *Proceedings IEEE WRTLT*, Beijing, China, Oct. 2007. IEEE Computer Society Press.

[177] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In *Proceedings TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

[178] M. Triska. The Finite Domain Constraint Solver of SWI-Prolog. In *Proceedings FLOPS*, volume 7294 of *LNCS*, pages 307–316, 2012.

[179] M. Triska. *Correctness Considerations in CLP(FD) Systems*. PhD thesis, Vienna University of Technology, 2014.

[180] A. M. Turing. Checking a Large Routine. In *Report on a Conference on High Speed Automatic Computation, June 1949*, pages 67–69, Cambridge, UK, 1949. University Mathematical Laboratory, Cambridge University.

[181] A. Valmari. A Stubborn Attack On State Explosion. In *Proceedings CAV*, volume 531 of *LNCS*, pages 156–165. Springer, 1990.

[182] Y. Vizel and A. Gurfinkel. Interpolating Property Directed Reachability. In *Proceedings CAV*, volume 8559 of *LNCS*, pages 260–276. Springer, 2014.

[183] G. Weissenbacher, D. Kröning, and P. Rümmer. A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-LIB Standard. In *Proceedings SMT*, SMT'09, 2009.

[184] T. Welp and A. Kuehlmann. QF_BV Model Checking with Property Directed Reachability. In *Proceedings DATE*, DATE '13, pages 791–796, San Jose, CA, USA, 2013. EDA Consortium.

[185] J. Witulski and M. Leuschel. Checking Computations of Formal Method Tools - A Secondary Toolchain for ProB. In *Proceedings F-IDE*, volume 149 of *EPTCS*. Electronic Proceedings in Theoretical Computer Science, 2014.

[186] F. Yang, J. P. Jacquot, and J. Souquieres. JeB: Safe Simulation of Event-B Models in JavaScript. In *Proceedings APSEC*, volume 1, pages 571–576. IEEE Computer Society, Dec 2013.

[187] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA$^+$ Specifications. In *Proceedings CHARME*, volume 1703 of *LNCS*, pages 54–66. Springer, 1999.

[188] Z. Zeng, M. Ciesielski, and B. Rouzeyre. Functional Test Generation Using Constraint Logic Programming. In *SOC Design Methodologies*, volume 90 of *IFIP*, pages 375–387. Springer, 2002.

[189] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2):249–288, Jun 2017.

List of Figures

List of Listings

List of Algorithms

List of Tables