New Applications and Techniques for Constraint Programming in B

Inaugural-Dissertation

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

vorgelegt von

Joshua Schmidt

aus Solingen

Düsseldorf, November 2023

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

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

Berichterstatter:

1. Prof. Dr. Michael Leuschel
   Heinrich-Heine-Universität Düsseldorf
2. Privatdozent Igor Konnov, PhD, TU Wien
   Principal Scientist, Informal Systems

Tag der mündlichen Prüfung: 24. Juli 2023

Ich versichere an Eides Statt, dass die Dissertation von mir selbständig und ohne unzulässige fremde Hilfe unter Beachtung der „Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Heinrich-Heine-Universität Düsseldorf“ erstellt worden ist. Die Dissertation wurde in der vorgelegten oder in ähnlicher Form noch bei keiner anderen Institution eingereicht. Ich habe bisher keine erfolglosen Promotionsversuche unternommen.

Düsseldorf, den 3. November 2023

Joshua Schmidt

“Testing shows the presence, not the absence of bugs.” [1, p.16]

Edsger Wybe Dijkstra

Abstract

The safety of software systems is gaining importance due to the almost indispensable integration of software in modern everyday life. Formal methods are a fundamental approach for the design and verification of software and hardware systems, one of which is the B-Method. This thesis is a selection of my co-authored manuscripts on the B-Method and the ProB tool.

In the first part, a translation of a popular formal specification language called Alloy to classical B is presented, which is automated by ProB. A difference between both languages is that Alloy's syntax is flexible and resembles object-oriented programming languages while B's syntax is strictly typed and rooted in set theory, mathematics, and logic. In contrast to Alloy, B also allows defining infinite and arbitrarily nested sets and relations. Further, B has operational semantics, which eases the definition of state-based systems. Empirical results have shown benefits for performance and soundness of B and ProB compared to Alloy when solving integer constraints, and benefits for performance of the Alloy Analyzer compared to ProB when solving relational constraints. This work contributed to an ongoing discussion in the Alloy community to integrate state-based concepts in the core language of Alloy and Satisfiability Modulo Theories (SMT) in the Alloy Analyzer. Besides that, this work improved the communication between the Alloy and B communities.

The second part of this thesis deals with constraint solving, which is one of the most important features of any formal verification tool. ProB's constraint solver has proven to be powerful for many problems. Yet, its use of plain saturation-based techniques often prevents finding contradictions, especially when considering infinite domains. We present additional constraint solving backends for ProB using techniques of SMT, which enable to learn from conflicts and leverage the power of Boolean satisfiability solving. In particular, we present an extended translation from B to SMT-LIB and integration of Z3 in ProB as well as a custom implementation of SMT in ProB. Empirical results have shown benefits of clause learning and abstractions to Boolean satisfiability solving compared to ProB's native constraint solver, especially when it comes to finding contradictions. For instance, it can be possible to identify a contradiction in a formula's Boolean abstraction without interpreting any theory constraint for which the satisfiability might be undecidable. Z3's theory solvers have further shown benefits in solving constraints involving infinite domains and integer arithmetic. The overall results, however, have shown that no constraint solver is the best for solving all kinds of constraints. For the verification of formal systems, it is thus beneficial to have a large portfolio of different constraint solving backends, as is the case for the ProB tool.

Zusammenfassung

Die Sicherheit von Softwaresystemen gewinnt durch die fast unverzichtbare Integration von Software in den modernen Alltag immer mehr an Bedeutung. Formale Methoden wie die B-Methode sind ein grundlegender Ansatz für den Entwurf und die Verifikation von Software- und Hardwaresystemen. Diese Arbeit ist eine Auswahl von Manuskripten zu der B-Methode und dem ProB Werkzeug, an denen ich grundlegend mitgewirkt habe.

Im ersten Teil dieser Arbeit wird eine Übersetzung einer populären formalen Spezifikationssprache namens Alloy in die klassische B Sprache vorgestellt, welche mittels ProB automatisiert wird. Ein Unterschied zwischen beiden Sprachen ist, dass die Syntax von Alloy flexibel ist und objektorientierten Programmiersprachen ähnelt, während die Syntax von B streng typisiert ist und auf der Mengenlehre, Mathematik und Logik basiert. Darüber hinaus verfügt B über eine operative Semantik, welche die Definition von zustandsbasierten Systemen erleichtert. Empirische Ergebnisse haben Vorteile für die Leistung und Korrektheit von B und ProB im Vergleich zu Alloy beim Lösen ganzzahliger Arithmetik gezeigt. Der Alloy Analyzer hat eine bessere Leistung im Vergleich zu ProB beim Lösen relationaler Einschränkungen gezeigt. Diese Arbeit hat zu einer bestehenden Diskussion in der Alloy Community bezüglich der Integration von zustandsbasiertem Verhalten in der Kernsprache von Alloy sowie der Verwendung von Satisfiability Modulo Theories (SMT) beigetragen. Außerdem hat diese Arbeit die Kommunikation zwischen der Alloy- und der B-Community verbessert.

Der zweite Teil dieser Arbeit befasst sich mit dem Constraint Solving, was eine der wichtigsten Fähigkeiten eines formalen Verifikationswerkzeug ist. Der Constraint Solver von ProB hat sich als leistungsfähig erwiesen. Die schiere Enumeration von Domänen verhindert jedoch häufig das Auffinden von Widersprüchen, insbesondere für unendliche Domänen. Wir präsentieren zusätzliche Constraint Solving Backends für ProB basierend auf SMT. Diese Techniken ermöglichen es, aus Konflikten zu lernen und die Vorteile des Lösens aussagenlogischer Formeln zu nutzen. Im Einzelnen präsentieren wir eine erweiterte Übersetzung von B nach SMT-LIB und Integration von Z3 in ProB sowie eine manuelle Implementierung von SMT. Empirische Ergebnisse haben Vorteile des Lernens von Klauseln und der Verwendung aussagenlogischer Abstraktionen im Vergleich zu ProB's Constraint Solver gezeigt, insbesondere für die Erkennung von Widersprüchen. Beispielsweise ist es möglich, dass ein Widerspruch in der booleschen Abstraktion einer Formel erkannt wird, ohne eine möglicherweise unentscheidbare Theoriebeschränkung zu interpretieren. Z3 hat außerdem Vorteile für das Lösen von Formeln mit unendlichen Domänen und ganzzahliger Arithmetik gezeigt. Insgesamt wurde jedoch deutlich, dass kein Constraint Solver der Beste ist, um alle Formeln zu lösen. Für die Verifikation formaler Systeme ist es daher von Vorteil über ein Portfolio unterschiedlicher Constraint Solver zu verfügen, wie dies bei dem ProB Werkzeug der Fall ist.

Acknowledgments

First, I would like to thank my doctoral adviser Michael Leuschel for giving me the opportunity and trust to work on the ProB tool as well as for teaching me a lot about logic (programming), constraint solving, formal methods, and many more. Although Michael has always worked a lot himself, he was available at all times and stood by me with advice and action. My deepest thanks go to Sebastian Krings for supervising my bachelor and master theses, introducing me to scientific research and teaching me a lot about logic (programming), constraint solving, writing scientific papers, giving scientific talks, and teaching at the university. Without Sebastian, I wouldn't be who I am today in the academic field. I would like to thank David Schneider for teaching me a lot about software development, clean coding, software life cycles, Git, and software testing as well as giving me the opportunity to work with him on the PlÜS tool. I would like to thank Claudia Kiometzis for the administrative management of my employment contracts, teaching commitments, and courses. I would like to thank Jens Bendisposto for his great commitment to teaching and his excellent lectures on software development and functional programming in particular. I would like to thank Igor Konnov for reviewing my dissertation and for his very thorough and useful feedback that helped improve this work.

Furthermore, I would like to thank all my fellow students and colleagues at the chair of software engineering and programming languages at HHU for the good time, the support, and the fruitful discussions. In no particular order, I especially want to mention: Jannik Dunkelau, Philipp Körner, Fabian Vu, David Geleßus, Dominik Hansen, Lukas Ladenberger, Alexandros Efremidis, Carl-Friedrich Bolz-Tereick, Jonas Benedikt Sven Schneider, Ivaylo Dobrikov, Max Jakub Ried, Mareike Mutz, Kristin Rutenkolk, John Witulski, Philip Höfges, and Joy Heron.

Last but not least, I would like to thank my family and friends for their support and patience.

Contents

List of Tables xvi

List of Figures xix

List of Listings xxi

List of Algorithms xxiii

I. Introduction 1

1. Motivation and Goals 3

2. Background 7
- 2.1. Logic . . . 7
  - 2.1.1. Propositional Logic . . . 7
  - 2.1.2. Predicate Logic . . . 9
  - 2.1.3. Programming in Logic . . . 10
- 2.2. Boolean Satisfiability Solving . . . 14
  - 2.2.1. DPLL Algorithm . . . 14
  - 2.2.2. Watched Literals . . . 18
- 2.3. Satisfiability Modulo Theories . . . 19
  - 2.3.1. Eager SMT Solving . . . 19
  - 2.3.2. Lazy SMT Solving . . . 20
- 2.4. Constraint Logic Programming . . . 21
  - 2.4.1. Domain Consistency . . . 22
  - 2.4.2. Labeling . . . 22
  - 2.4.3. Constraint Reification . . . 23
- 2.5. Formal Methods and Formalisms . . . 23
  - 2.5.1. Linear Time Properties . . . 25
  - 2.5.2. Linear Temporal Logic . . . 26
  - 2.5.3. Model Checking . . . 26
  - 2.5.4. The B-Method . . . 27
  - 2.5.5. Alloy 5 . . . 28
- 2.6. ProB . . . 28

3. Thesis Structure 31

II. New Applications for Constraint Programming in B 35

4. Translating Alloy and Extensions to Classical B 37
4.1. Introduction . . . 37
4.2. Introduction to Alloy and B . . . 38
4.2.1. Primer on Alloy . . . 39
4.2.2. Primer on B . . . 39
4.2.3. Comparing Alloy and B . . . 40
4.3. Translation Example . . . 41
4.3.1. Translating Signatures . . . 41
4.3.2. Translating Facts and Predicates . . . 43
4.4. Formal Description of the Translation . . . 44
4.4.1. Overview of the Semantic Functions . . . 45
4.4.2. Example . . . 46
4.4.3. Signature and Field Declarations . . . 47
4.4.4. Universe and Identity . . . 48
4.4.5. Connectives and Simple Predicates . . . 50
4.4.6. Simple Expressions . . . 51
4.4.7. Representing Tuples . . . 51
4.4.8. Cartesian Product . . . 52
4.4.9. Domain and Range Restriction . . . 52
4.4.10. Join . . . 53
4.4.11. Quantifications, Set Comprehensions and Identifiers . . . 54
4.4.12. Conditionals . . . 55
4.4.13. Fact, Function & Predicate Declaration . . . 55
4.4.14. Assertion Declaration and Run & Check Commands . . . 56
4.4.15. Multiplicity Annotations . . . 57
4.4.16. Post-Processing Optimization Rules . . . 58
4.5. Translation of Alloy Extensions . . . 58
4.5.1. Integers and Natural Numbers . . . 59
4.5.2. Relations . . . 62
4.5.3. Orderings . . . 63
4.5.4. Enumerations . . . 64
4.5.5. Sequences . . . 65
4.6. Tooling . . . 68
4.6.1. Overview . . . 68
4.6.2. Prolog Encoding of Translation Rules . . . 69
4.7. Empirical Evaluation . . . 70
4.8. Improvements Over Existing Alloy Tools . . . 75
4.8.1. Integers . . . 75
4.8.2. Higher-Order Quantification . . . 76
4.8.3. Proof . . . 76
4.9. Related and Future Work . . . 77
4.10. Conclusion . . . 79

5. Additional Experiments and Considerations **81**
5.1. Extended Verification of Commands . . . 81
5.2. Additional Empirical Evaluation . . . 83
5.3. Translating Alloy 6 to B . . . 86

III. New Techniques for Constraint Programming in B **91**

6. SMT Solving for the Validation of B and Event-B Models **93**
6.1. Introduction . . . 93
6.2. Background . . . 96
6.2.1. Primer on B . . . 96
6.2.2. Primer on SMT-LIB . . . 97
6.3. Former Z3 Integration . . . 98
6.3.1. High-Level Translation . . . 99
6.3.2. Workflow . . . 100
6.4. New Z3 Integration . . . 101
6.4.1. High-Level Translation . . . 101
6.4.2. New Workflow . . . 107
6.4.3. Decomposition of Constraints . . . 109
6.5. SMT Solving in ProB . . . 110
6.5.1. SMT Workflow in ProB . . . 111
6.5.2. Preprocessing . . . 111
6.5.3. SAT Solving . . . 115
6.5.4. SMT Solving . . . 119
6.6. Additional Theory Solver . . . 124
6.6.1. Integer Difference Logic Solver . . . 124
6.6.2. Theory Solver Integration . . . 126
6.7. Empirical Evaluation . . . 126
6.7.1. Integration of Z3 . . . 126
6.7.2. Direct Implementation of SMT Solving in ProB . . . 134
6.8. Related Work . . . 142
6.9. Future Work . . . 145
6.10. Conclusion . . . 146

7. Additional Experiments and Considerations **149**
7.1. Unfixed Deferred Sets . . . 149
7.2. Memory Consumption of Z3 . . . 151
7.3. Well-Definedness . . . 152
7.4. Revisiting Empirical Evaluation . . . 153
7.5. Quantifier Instantiation . . . 155
7.6. Additional Empirical Evaluation . . . 158
7.6.1. Bounded Model Checking . . . 159
7.6.2. Deadlock and Inductive Invariant Checking . . . 162

7.7. Soundness 166

IV. Conclusion **169**

8. Translating Alloy and Extensions to Classical B **171**

9. SMT Solving for the Validation of B and Event-B Models **175**

10. Future Work **181**

Appendix **183**

A. Alternative Alloy 6 Model of the Chameleon Puzzle **185**

B. Transformation Rules for Integer Difference Logic **187**

C. Discussion on Github about Z3's Memory Consumption **189**

D. Detailed Results of Additional Constraint Solving Benchmarks **191**

Bibliography **195**

List of Tables

4.1. Performance evaluation of the Alloy Analyzer and ProB's constraint solver. 74

5.1. Performance evaluation of the Alloy Analyzer and different constraint solving backends of ProB. 84

6.1. Performance evaluation of ProB's constraint solver and its backends to Z3 using constraints from BMC. 129
6.2. Performance evaluation of ProB's constraint solver, its parallel backend to Z3, and its different SMT solver configurations using constraints from BMC. 134
6.3. Detailed statistics of the different configurations of ProB's SMT solver considering all constraints presented in Table 6.2. 136
6.4. A subset of the classical B and Event-B models from Table 6.1 and Table 6.2, but checking the inductivity of invariants. 138
6.5. Detailed statistics of the different configurations of ProB's SMT solver considering all constraints for inductive invariant proofs presented in Table 6.4. 139
6.6. A subset of the classical B and Event-B benchmarks used in Table 6.1 and Table 6.2 but solving constraints to prove deadlock freedom. 141
6.7. Detailed statistics of the different configurations of ProB's SMT solver considering all constraints for deadlock freedom proofs presented in Table 6.6. 142
6.8. A set of benchmarks from BMC of an Event-B model of a pacemaker [2] comparing the different configurations of ProB's SMT solver using the new theory solver for IDL with ProB's constraint solver and the new parallel integration of Z3. 143
6.9. Detailed statistics of the different configurations of ProB's SMT solver using the additional theory solver for IDL considering all BMC constraints presented in Table 6.8. 145

7.1. The BMC constraints presented in Table 6.1 but using updated versions of ProB, Z3, and SICStus Prolog. 154
7.2. Performance evaluation of different configurations of the n queens problem comparing ProB's constraint solver, its parallel integration of Z3, and its SMT solver with and without quantifier instantiation. 156
7.3. Performance comparison of ProB's constraint solver, its backend to Kodkod, its parallel integration Z3, and its SMT solver for checking Alloy models with and without quantifier instantiation. 157

D.1. First part of detailed results of the BMC benchmarks used in Section 7.6. 191

D.2. Second part of the BMC benchmarks presented in Table D.1. In total, 393 constraints were solved. 192

D.3. Detailed results of the benchmarks from deadlock freedom proofs used in Section 7.6. In total, 112 constraints were solved. 193

D.4. Detailed results of the inductive invariant checking benchmarks used in Section 7.6. In total, 1302 constraints were solved. 194

List of Figures

2.1. An exemplary Prolog predicate computing the length of a list using an accumulator for tail-recursion and its corresponding representation in first-order logic. . . . 10
2.2. An exemplary SLD resolution tree for the Prolog query `len([1,2], 0, L)` using the predicate `len/3` shown in Figure 2.1. . . . 13
2.3. Example of the DPLL algorithm to find a solution for a propositional logic formula. . . . 17
2.4. Hypergraph representation of the constraint $x \in 1..10 \wedge y \in 3..5 \wedge z \in 10..15 \wedge x > 2 \wedge y < x \wedge x + y > z$ with and without consistent domains. 21
2.5. A program graph of the Euclidean algorithm using division [3] as well as a transition system that has been unfolded from the program graph. . . . 24

4.1. Signatures and fields of the Own Grandpa model in Alloy and B. . . . 42
4.2. A visualization of the different phases of the Translation from Alloy to B. 69
4.3. Visualization of the Alloy Analyzer's and ProB's constraint solver's runtimes for finding a single solution for the n queens puzzle with varying n. . . . 72

5.1. Checking an Alloy command in ProB Tcl/Tk via the main menu. . . . 82

6.1. A formal description of our syntax-directed translation rules for translating a subset of B's set operators to SMT-LIB as understood by Z3. . . . 104
6.2. A workflow diagram of the new integration of Z3 in ProB running two Z3 constraint solvers in parallel using the former and new translation from B to SMT-LIB. . . . 107
6.3. A generalized workflow diagram of the direct implementation of SMT solving in ProB. . . . 111
6.4. A colored graph to find static symmetry breaking predicates for the formula $x < y \wedge y < x$ by computing graph automorphisms. . . . 113
6.5. Exemplary conflict analysis for SAT solving using resolution. . . . 116
6.6. A graph representation of the IDL constraint $x \in \mathbb{Z} \wedge x > y \wedge y > x$ as suggested by Wang et al. [4]. . . . 125
6.7. A Venn diagram to visualize and compare the amount of BMC constraints that can be solved by ProB's constraint solver and Z3 using the axiomatic and the constructive translation from B to SMT-LIB as can be seen in Table 6.1. . . . 130

6.8. A visualization of the BMC benchmark results presented in Table 6.1 showing the amount of constraints that can be solved by a constraint solver within a specific amount of time. 131

6.9. A visualization of the BMC benchmark results presented in Table 6.2 showing the amount of constraints that can be solved by a constraint solver within a specific amount of time. 133

6.10. A Venn diagram to visualize and compare the amount of BMC constraints that can be solved by ProB's constraint solver, the parallel integration of Z3 in ProB, and the best performing configuration of ProB's SMT solver. 135

6.11. A Venn diagram to visualize and compare the amount of BMC constraints that can be solved by ProB's constraint solver, ProB's SMT solver with static syntax analysis, and the same SMT solver configuration but additionally using the theory solver for IDL. 144

7.1. A formal description of our syntax-directed rules for adding all necessary well-definedness conditions to B or Event-B predicates. 152

7.2. A Venn diagram showing the amount of BMC constraints that can be solved by ProB's constraint solver and the different configurations of Z3 using version `4.12.3` built from commit `1d62964c` as can be seen in Table 7.1. 155

7.3. A Venn diagram showing the amount of BMC constraints that can be solved by ProB's constraint solver, its parallel integration of Z3, and its SMT solver using the additional static analysis and symmetry breaking as can be seen in Table D.1 and Table D.2. 160

7.4. A visualization of the BMC benchmark results presented in Table D.1 and Table D.2 showing the amount of constraints that can be solved by a constraint solver within a specific amount of time. 162

7.5. A Venn diagram showing the amount of constraints from deadlock freedom proofs that can be solved by ProB's constraint solver, its parallel integration of Z3 that decomposes constraints into independent components, and its SMT solver without any additional syntax analysis as can be seen in Table D.3. 163

7.6. Venn diagrams showing the amount of constraints from inductive invariant proofs that can be solved by ProB's constraint solver, its integrations of Z3, and its SMT solver without any additional static analysis as can be seen in Table D.4. 164

B.1. Syntax-directed rules for rewriting a subset of integer constraints into IDL as implemented in ProB's graph-based constraint solver for IDL. 188

C.1. Excerpt from a comment on Github about a memory leak in Z3, especially when using many Z3 solvers in parallel [5]. 189

C.2. An answer of Nikolaj Bjørner, one of Z3's core developers, on the issue regarding the memory leak in Z3 as can be seen in Figure C.1 [6]. 189

List of Listings

4.1. Signatures of the Own Grandpa model in Alloy. 41
4.2. Signatures of the Own Grandpa model translated to B. 42
4.3. Facts and predicates of the Own Grandpa model in Alloy. 43
4.4. Facts and predicates of the Own Grandpa model translated to B. 44
4.5. Parts of the Prolog code that translates Alloy's set union to B. 70
4.6. An exemplary Alloy model for checking of mathematical laws. 71
4.7. An Alloy model demonstrating the unsoundness of integers. 75
4.8. An exemplary Alloy model using higher-order quantification. 76
4.9. An exemplary Alloy model to prove an assertion in AtelierB. 77
4.10. A translated Alloy model to prove an assertion in AtelierB. 78

5.1. An idiomatic formal specification of the „Chameleon Puzzle“ [7] in classical B. 87
5.2. A formal specification of the „Chameleon Puzzle“ [7] in Alloy 6 presented by Peter Kriens [8]. 88
5.3. A manual translation of the Alloy 6 specification presented in Listing 5.2 to classical B. 89

6.1. Prolog code for the implementation of the theory propagation for a "less than" comparison in ProB's constraint solver using coroutines. 122

7.1. B encoding of the n queens problem using two universal quantifiers that can be transformed by quantifier instantiation. 156

A.1. Alloy 6 model of the „Chameleon Puzzle“ [7]. 185
A.2. A manual translation of the Alloy 6 model presented in Listing A.1 to classical B. 185

List of Algorithms

2.1. Pseudocode of the unification algorithm by Robinson [9, 10]. 11
2.2. Pseudocode of the DPLL algorithm's unit propagation [11, 12]. 15
2.3. Pseudocode of the DPLL algorithm's resolution-based simplification procedure for propositional logic formulas [11, 12]. 15
2.4. Pseudocode of the DPLL algorithm [11, 12]. 16

6.1. Pseudocode of the constraint solving routine for the integration of Z3 in ProB that decomposes B constraints into independent components. . . . 110

Part I.
Introduction

1. Motivation and Goals

Software has become a part of everyday life and is used in almost all areas. While this is definitely an advantage and simplifies many tasks, the quality assurance often suffers due to the high demand for software systems. Concrete reasons can be time and cost pressure, ignorance, lack of experience or human errors. There are many possible software bugs, each with a different severity regarding specific requirements. In the context of software that interacts with humans, special attention must be paid to the safety of software systems. Safety generally means that a software is guaranteed to avoid any state that violates specific requirements. For instance, software used in autonomous driving, railway or aerospace systems must meet precise formal requirements to ensure the avoidance of potentially life-threatening situations.

Formal methods are a fundamental approach for the development and verification of software (and hardware) systems in software engineering. The origins of the discipline of software engineering date back to the 1970s [13]. Up to the present time, language and tool support for formal methods have improved significantly. Aside from being an active area of research, formal methods have also been used in many practical areas. For instance, the software of the first driverless metro in the city of Paris is an example for a successful application of formal methods [14]. In a similar approach, Siemens designed and verified a software that has been installed on many international metro lines, *e.g.*, on the Canarsie Line in New York [15]. Another recent example for the use of formal methods in industry is the design and verification of a new approach to railway interlocking, namely the ETCS Hybrid Level 3 [16]. In recent years, formal methods have also been used for the verification of systems based on artificial intelligence such as neural networks [17–20] or machine learning in general [21–24].

There are many formal specification languages with different approaches for the design and verification of systems. Most languages are based on the concepts of mathematics and logic. While the syntax and semantics of a specification language are important for the acceptance and use in the formal methods community, the most important feature of a formal specification language is its tool support. In particular, efficient tools for the actual verification of requirements are necessary. Constraint programming is a fundamental technique used in almost all tools for formal verification, which generally aims at proving or disproving logical and mathematical formulas. Many different approaches for solving constraints exist, each having its pros and cons. Verification tools thus often provide different constraint solving backends.

The B-Method [25, 26] is a formal method for software development, which is actively used in academia and industry. Its foundation is an expressive formal specification language which is rooted in typed set theory, integer arithmetic, and first-order logic. One feature of the B-Method is that a system's amount of states can be infinite. While this allows for the specification of manifold systems, the use of unbounded domains often aggravates constraint solving. ProB [27–29] is an animator, model checker, and constraint solver for the B-Method. Its constraint solver applies constraint logic programming (CLP) [30] and defines a rich set of propagation rules for the B language.

ProB, and in particular the B-Method, has been used in many industrial applications [14–16, 31–37]. Further, ProB is certified T2 SIL4 according to the EN 50128 [38] standard defined by the European Committee for Electrotechnical Standardization (CENELEC). Yet, the use of the B-Method in the formal methods community could be greater. One criticism is that the B language is difficult to learn and requires deep knowledge of mathematics and logic. Especially in the United States, other formal methods and tools such as Alloy [39, 40] and the Alloy Analyzer, TLA$^+$ and the TLC model checker [41] or Promela and the SPIN model checker [42] are more popular.

The first goal of this thesis is therefore to bring the formal methods community closer together. For this, we deem a translation between Alloy and B to be suited. First research questions (RQ) are:

> **RQ1:** Which steps are necessary to automatically translate Alloy models into B?
>
> **RQ2:** How does ProB compete in checking Alloy models compared to the Alloy toolchain?
>
> **RQ3:** What are significant strengths and weaknesses of SAT solving (Alloy Analyzer) compared to CLP (ProB)?
>
> **RQ4:** Which main use cases can be singled out in which both languages complement each other?

To answer these questions, we started with manually translating Alloy models to B and working out their differences and similarities. This work finally manifested in a formalization of a complete set of translation rules from Alloy 5 to classical B [43]. Based on that, we presented an automated integration of Alloy 5 in ProB enabling to use its features for animation and verification [43, 44]. The performance of the Alloy Analyzer and ProB were evaluated empirically using a set of publicly available Alloy models.

At the heart of ProB is its constraint solver which is used for many tasks. ProB also has backends to external constraint solvers such as Kodkod [45, 46], which translates constraints to propositional logic and applies satisfiability (SAT) solving, or Z3 [47, 48], which uses a modern constraint solving approach called satisfiability modulo theories (SMT). Z3 has shown benefits for disproving B formulas but often fails to prove such due to complex translations to SMT-LIB [48]. While ProB's constraint solver implements many sophisticated features such as coroutines for a delayed constraint propagation or memoization, there are still many techniques which could improve constraint solving for the B-Method such as conflict-driven clause learning (CDCL) or backjumping instead of chronological backtracking.

The second goal of this thesis is thus to improve the performance and coverage of ProB's portfolio of constraint solving backends. We first ask:

RQ5: How can the performance and coverage of ProB's integration of Z3 be improved?

To answer this question, we extended the existing integration of Z3 in ProB to optionally use lambda functions for the translation of specific B and Event-B operators to SMT-LIB [49, 50]. Additionally, a parallel integration of different Z3 constraint solvers as well as a decomposition of constraints into independent components prior to the translation was presented. Inspired by benefits of Z3, we further ask the following questions:

RQ6: Which steps are necessary to use ProB's constraint solver as a theory solver for SMT solving of B and Event-B constraints?

RQ7: What are significant benefits of a direct implementation of SMT solving in ProB compared to using Z3?

RQ8: What are significant strengths and weaknesses of CLP and SMT for solving B and Event-B constraints?

RQ9: How can the performance of ProB's constraint solver in disproving integer constraints over unbounded domains be improved?

For this, we presented a direct implementation of SMT solving in ProB's Prolog core to combine the strengths of ProB's constraint solver for finding solutions with a CDCL scheme to improve the identification of contradictions [50]. Empirical evaluations have shown weaknesses of ProB's constraint solver for disproving formulas over unbounded integer domains. To counter this, we further presented an additional constraint solver for the integer difference logic (IDL) and its integration in ProB's new SMT solver [50].

We decided to implement the SMT solver in Prolog since ProB's constraint solver is implemented in Prolog. The use of Prolog thus allows for a fast and intuitive combination of the SAT solver and ProB's constraint solver for SMT solving. This decision leads to the following final research question:

RQ10: What are significant features of Prolog that are specifically suitable or unsuitable for implementing an SMT solver?

2. Background

In this chapter, we provide an introduction to the most important background knowledge which is necessary for this thesis. We start by introducing propositional logic and predicate logic as well as a programming paradigm called logic programming. Afterward, we introduce different concepts of satisfiability solving for propositional logic and predicate logic followed by an introduction to formal methods in computer science. Here, we focus on two prominent formal methods called the B-Method and Alloy. Furthermore, we introduce the concepts of model checking and constraint solving which are essential in the field of formal verification. Finally, we give an overview of the ProB tool, which is the main tool used in this thesis.

2.1. Logic

Logic is a foundation of human knowledge. We humans gather knowledge in the form of facts and experiences during our lifetime and are able to derive new knowledge by combining existing one. This is called logical reasoning. In particular, logical reasoning is the action of deducing knowledge from existing premises. There are different manifestations of logic while propositional logic and predicate logic are most important in mathematics and computer science.

2.1.1. Propositional Logic

In propositional logic, it is possible to reason over statements that can have exactly one truth value, *e.g.*, true ($\top$) or false ($\bot$) [51–53]. Relations between statements can be expressed using sentential connectives such as logical conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$) or equivalence ($\Leftrightarrow$) [51, 54]. The logical conjunction and disjunction are commutative. Besides exact truth values, propositional logic allows defining variables for truth values. Such variables are usually referred to as literals and can have a polarity which is either positive or negative [51–53]. For instance, $\neg$A is a literal with negative polarity.

Let ϕ be a propositional logic formula and $V(\phi)$ be the set of variables occurring in ϕ. An interpretation of a propositional logic formula is an assignment of its variables $\mu(\phi)$ which can be generally defined as a partial function $V(\phi) \nrightarrow \{\top, \bot\}$ [51–55]. A propositional logic formula ϕ is satisfiable if there exists an interpretation $\mu(\phi)$ which makes the formula true. Such an interpretation $\mu(\phi)$ is called a model if it makes the formula ϕ true and is a total function, *i.e.*, all variables in $V(\phi)$ are assigned a truth value. The interpretation $\mu(\phi)$ is referred to as a partial model if only the variables which are necessary to make ϕ true are assigned, *i.e.*, $\mu(\phi)$ is a partial function.

For processing, logical formulas are usually transformed to a normal form [56–58]. The disjunctive normal form (DNF) is a disjunction of conjunctions while the conjunctive normal form (CNF) is a conjunction of disjunctions [56–58]. The individual conjuncts of the conjunctive normal form, *i.e.*, the disjunctions, are called clauses. In order to rewrite propositional logic formulas to either disjunctive or conjunctive normal form, DeMorgan's laws for propositional logic can be applied exhaustively [57–59]. Afterward, the laws of distributivity can be applied to distribute conjunctions over disjunctions or vice versa.

The disjunctive normal form of a propositional logic formula describes the assignments of variables which make the formula true within the conjunctions. It is thus possible to directly read all (partial) models from a disjunctive normal form. However, for a propositional logic formula with n variables, there exists 2^n different assignments of truth values. This leads to an exponential blowup in the size of a disjunctive normal form in the worst case [56], which makes the generation of a disjunctive normal form not scalable in general. The conjunctive normal form is more concise and is thus preferably used as a canonical form in propositional logic.

Nevertheless, the conjunctive normal form can also experience an exponential blowup in the amount of clauses related to the amount of subformulas when distributing disjunctions over conjunctions by applying the law of distributivity [57, 60]. Furthermore, the expansion of nested equivalences to disjunctions also leads to an exponential increase in the amount of clauses, *i.e.*, formulas of the form $\mathrm{A}_1 \Leftrightarrow \mathrm{A}_2 \Leftrightarrow \cdots \Leftrightarrow \mathrm{A}_k$, $k \in \mathbb{N}$. Tseitin [60] thus proposed to introduce new variables for each subformula of a propositional logic formula excluding single variables, which is also known as renaming. The new variables are each set to be equivalent to their corresponding subformula while all of these equivalences are conjoined. Afterward, each equivalence is transformed to conjunctive normal form independently. This transformation results in an equisatisfiable but not equivalent propositional logic formula, *i.e.*, it provides the same models for the original variables. For instance, consider the following formula in conjunctive normal form:

$$\begin{aligned} &(\mathrm{A} \wedge \mathrm{B}) \vee (\mathrm{C} \wedge \mathrm{D} \wedge \mathrm{E} \wedge \mathrm{F}) \\ \equiv\ &(\mathrm{A} \vee \mathrm{C}) \wedge (\mathrm{A} \vee \mathrm{D}) \wedge (\mathrm{A} \vee \mathrm{E}) \wedge (\mathrm{A} \vee \mathrm{F}) \wedge \\ &(\mathrm{B} \vee \mathrm{C}) \wedge (\mathrm{B} \vee \mathrm{D}) \wedge (\mathrm{B} \vee \mathrm{E}) \wedge (\mathrm{B} \vee \mathrm{F}) \end{aligned} \tag{2.1}$$

The Tseitin transformation would introduce one new variable U for the complete formula and two variables V and W for the two subformulas. This results in the following equisatisfiable propositional logic formula:

$$\mathrm{W} \wedge (\mathrm{W} \Leftrightarrow (\mathrm{U} \vee \mathrm{V})) \wedge (\mathrm{U} \Leftrightarrow (\mathrm{A} \wedge \mathrm{B})) \wedge (\mathrm{V} \Leftrightarrow (\mathrm{C} \wedge \mathrm{D} \wedge \mathrm{E} \wedge \mathrm{F}))$$

Each conjunct is then transformed to the conjunctive normal form. This ensures that the amount of a CNF's clauses increases linearly related to the amount of subformulas in the worst case. Yet, the amount of clauses is not necessarily smaller than without rewriting.

To that effect, several optimizations of the Tseitin transformation have been proposed to date [61–65]. Plaisted and Greenbaum [63] observed that new introduced variables do not necessarily have to be equivalent to their corresponding propositional logic formula. The use of an implication can be sufficient depending on the polarity of subformulas [63]. However, the authors still introduced new variables for each subformula. Boy de la Tour has proven several transformations to be useless in specific arrangements of subformulas [66] and proposed to only introduce new variables for subformulas if this transformation reduces the final amount of clauses in the conjunctive normal form [61]. For this, functions computing the amount of clauses of a logical formula after rewriting to CNF with and without renaming were defined. Nonnengart et al. [64] presented efficient algorithms for the improved transformation of Boy de la Tour [61]. Jackson and Sheridan [65] also presented a compact conversion to conjunctive normal form with renaming.

For the original formula in Equation (2.1), it suffices to rename the right-hand side of the disjunction introducing one fresh variable U as follows:

$$\begin{aligned} &((\mathrm{A} \land \mathrm{B}) \lor \mathrm{U}) \land (\mathrm{U} \Leftrightarrow (\mathrm{C} \land \mathrm{D} \land \mathrm{E} \land \mathrm{F})) \\ \equiv\ &(\mathrm{A} \lor \mathrm{U}) \land (\mathrm{B} \lor \mathrm{U}) \land (\neg\mathrm{C} \lor \neg\mathrm{D} \lor \neg\mathrm{E} \lor \neg\mathrm{F} \lor \mathrm{U}) \land (\mathrm{C} \lor \neg\mathrm{U}) \land (\mathrm{D} \lor \neg\mathrm{U}) \land \\ &(\mathrm{E} \lor \neg\mathrm{U}) \land (\mathrm{F} \lor \neg\mathrm{U}) \end{aligned}$$

The CNF contains one clause less than without renaming.

2.1.2. Predicate Logic

The predicate logic is an extension of the propositional logic which allows reasoning over domain specific objects, function symbols, and predicates rather than just truth values [54, 67–69]. A predicate describes properties and relations of one or more objects and evaluates to either true or false. For instance, we are able to reason over the integers and $\mathrm{eq}(1, 2)$ can be a predicate describing the equality between two integers with an arity of two and the functor eq. Furthermore, predicate logic introduces the concept of existential ($\exists$) and universal ($\forall$) quantification, which allows quantifying over domain specific variables or predicates [54, 67–69]. First-order logic (FOL) is a restricted form of predicate logic which allows quantifying over domain specific objects, *e.g.*, integers, but not predicates [54, 67, 68]. For instance, the formula $\exists(x, \mathrm{p}) : x \in \mathbb{Z} \land \mathrm{p}(x)$ uses second-order logic since it quantifies a predicate.

A logic is decidable if all valid and invalid solutions can be enumerated for each formula. FOL is undecidable in general [70–72]. For instance, it is possible to describe the behavior of an arbitrary Turing machine in FOL as well as the property that the Turing machine halts for each input [72], which is known as the halting problem. Turing has proven that the halting problem is undecidable leading to the conclusion that FOL is also undecidable [72]. Church came to the same conclusion and presented an independent and different proof [71]. Yet, for a given finite set of consistent axioms and a formula that is implied by these axioms, it is possible to prove that the formula is valid under these axioms by finite enumeration. Therefore, FOL is said to be complete and semi-decidable [73].

```
len([], Acc, Acc).
len([_|T], Acc, Len) :-
        NAcc is Acc + 1,
        len(T, NAcc, Len).
```

$$\forall \texttt{Acc}.(\texttt{len([], Acc, Acc)})$$
$$\forall(\texttt{T}, \texttt{Acc}, \texttt{NAcc}, \texttt{Len}).(\texttt{len([\_|T], Acc, Len)} \lor \neg(\texttt{NAcc is Acc + 1}) \lor \neg\texttt{len(T, NAcc, Len)})$$

Figure 2.1.: An exemplary Prolog predicate computing the length of a list using an accumulator for tail-recursion (left) and its corresponding representation in first-order logic (right).

2.1.3. Programming in Logic

Logic programming is a programming paradigm that allows defining relations between logical facts. Solutions can be found by using a corresponding proof engine. In contrast to that, one usually defines a sequence of instructions in imperative programming languages or transformations on data structures in functional programming languages.

The first ideas of a logic programming language date back to the early 1970s [74] which finally manifested in the Prolog programming language [75]. A core language has first been standardized in 1995 [76, 77]. An overview of the last 50 years of Prolog has recently been presented by Körner et al. [78].

Prolog is a dynamic and interpreted programming language. The main characteristics of Prolog are the application of linear resolution restricted to horn clauses for prove, and homoiconity. A horn clause is a clause with at most one positive literal. Prolog allows defining formulas in first-order logic. Prolog rules consist of a body predicate implying a head statement such as `r :- p, q` which is equivalent to $r \Leftarrow p \wedge q$ in propositional logic [79, 80]. Furthermore, it is possible to define Prolog facts which are rules with an empty body (*i.e.*, the body is just a truth statement). A predicate can have arguments which can be used for input, output or both. In the documentation of predicates, arguments are prefixed with a `+` for inputs, `-` for outputs, and `?` for arguments that can be both [79, 80]. For instance, `append([1,2], [2,3], C)` is a standard Prolog predicate with arguments asking which list results when appending the lists `[1,2]` and `[2,3]`. The implementation of `append/3` is implemented strictly logical which allows calling the predicate with any combination of arguments such as `append(A, B, [1,2,3])` asking which two lists `A` and `B` can be appended to the list `[1,2,3]`. Figure 2.1 shows an example of a Prolog predicate computing the length of a list using an accumulator for tail-recursion and its corresponding representation in first-order logic.

The concept of homoiconity states that a programming language defines a single data type while its syntax is implemented in this data type. In Prolog, everything is a term. A term has a functor and an arity which defines the amount of the term's arguments [79, 80]. For instance, abbreviated as `p/2` for a predicate `p` with two arguments. Terms are further divided in simple terms (no arguments) and compound terms (one or more arguments) [79, 80]. Although there is only one data type, one usually distinguishes between atoms, numbers, and lists in Prolog. Atoms are simple terms which are not

Algorithm 2.1 Pseudocode of the unification algorithm by Robinson [9, 10].

Input two Prolog terms to be unified

Output a variable substitution (most-general unifier) making both terms equal, or false

```
1: procedure UNIFY(A, B)
2:     $k \leftarrow 0$
3:     $\theta_0 \leftarrow \varnothing$
4:     while true do
5:         if $A\theta_k = B\theta_k$ then
6:             return $\theta_k$
7:         else
8:             find disagreement tuple $< a_k, b_k >$ of $A\theta_k$ and $B\theta_k$
9:         $k \leftarrow k + 1$
10:        if $a_k$ is a variable that does not occur in $b_k$ then
11:            $\theta_k \leftarrow \theta_{k-1}\{a_k/b_k\}$
12:        else if $b_k$ is a variable that does not occur in $a_k$ then
13:            $\theta_k \leftarrow \theta_{k-1}\{b_k/a_k\}$
14:        else
15:            return false
```

numbers. Lists can be written using square brackets such as `[a,2]`, which is syntactical sugar only. Furthermore, a head-tail concatenation can be used to access one or more front elements and a list of the tail elements. For instance, `[a,2] = [H|T]` results in assigning `H` with `a` and `T` with `[2]`. The tail of a list can also be the empty list. Internally, lists are represented as Prolog terms. For instance, the above list is represented as the term `'.'(a,'.'(2,[]))` in SICStus Prolog [81]. One benefit of homoiconity is that code can be seen as data and executable code. For instance, one is able to dynamically adapt or generate Prolog code and execute it. Another example for a homoiconic programming language is LISP, where everything is a list.

A Prolog program generally consists of a list of horn clauses (aka knowledge base) [79, 80]. The user makes queries to a Prolog interpreter in the form of predicates. Prolog facts and rules can be dynamically added to or removed from the knowledge base during the interpretation using `assert/1` or `retract/1`. In order to find a solution, a Prolog interpreter applies linear resolution for first-order logic [9, 11] with unification [10] as is described in the following.

Unification

The unification algorithm by Robinson [9, 10] is essential to apply the concept of resolution to first-order logic. A pseudocode representation of the unification algorithm can be seen in Algorithm 2.1. The algorithm's input are two terms and the output is a substitution of variables that makes both terms equal or false if both terms cannot be made equal. Substitutions allow substituting variables occurring in terms with other values. For instance, the substitution $\{\mathrm{A}/1, \mathrm{B}/2\}$ can be applied to the term $\mathrm{p}(\mathrm{A}, \mathrm{B})$, *i.e.*, $\mathrm{p(A, B)\{A/1, B/2\}}$, resulting in $\mathrm{p(1, 2)}$. The algorithm starts with the empty substitution and checks if both terms are already equal. If this is not the case, a so-called disagreement tuple is searched. A disagreement tuple $\langle a_k, b_k\rangle$, $k \in \mathbb{N}$, is a pair of subterms a_k occurring in A and b_k occurring in B that are not equal but at the same position in both terms. The search for a disagreement tuple starts at the leftmost position in both terms. For instance, the first disagreement tuple of the term $\mathrm{f(A, 1)}$ and $\mathrm{f(a, B)}$ is $\langle \mathrm{A, a}\rangle$ while $\langle \mathrm{1, B}\rangle$ is the second one. Afterward, the algorithm tries to unify both terms either by assigning a_k to b_k if a_k is a variable that does not occur in b_k, or b_k to a_k if b_k is a variable that does not occur in a_k. The check for occurrence of a variable is usually called the occurs-check. If the variables in a disagreement tuple cannot be made equal, the algorithm fails indicating that both input terms cannot be unified. Otherwise, the algorithm searches for disagreement tuples and assigns variables as long as both terms are not equal. The final substitution is guaranteed to be minimal and is called the most-general unifier. In general, a substitution Θ is more general than another substitution Φ if there exists a substitution σ such that $\Phi = \Theta\sigma$ [9].

The occurs-check guarantees that no cyclic dependencies exist [9]. For instance, the unification $\mathrm{A = f(A)}$ fails since the disagreement tuple is $\langle \mathrm{A, f(A)}\rangle$ which would result in a cyclic assignment f(f(f(f(f(...))))). By default, most Prolog implementations do not apply the occurs-check because it is additional overhead. This is not a problem in practice since one usually does not write cyclic variable dependencies. If this is done, it can lead to unexpected behavior in the future due to failing unifications. However, when debugging such code, it should be easy to recognize the cyclic dependency of variables.

Selective Linear Definite Clause Resolution

Most Prolog interpreters prove queries by contradiction using linear resolution [9, 11, 82, 83] with unification [10]. Resolution is the process of deducing a formula $\mathrm{A} \vee \mathrm{C}$ from two clauses $(\mathrm{A} \vee \mathrm{B}) \wedge (\neg\mathrm{B} \vee \mathrm{C})$ containing the same literal with positive and negative polarity. This can be lifted to predicate logic by applying unification. For instance, we can deduce the formula $\mathrm{p(a)} \vee \mathrm{q(a)}$ from the clauses $(\mathrm{p(a)} \vee \mathrm{q(b)}) \wedge (\neg\mathrm{q(B)} \vee \mathrm{q(a)})$ using the unification $\{\mathrm{B/b}\}$. A sequence of resolution steps is linear if the last deduced clause of each step is directly used for the next deduction (depth-first search). As described before, a Prolog program consists of a list of horn clauses, which can potentially be used for linear resolution. The Prolog interpreter enforces an order of Prolog rules and facts by selecting from top to bottom (selection rule), which is why the process is called selective linear definite (SLD) clause resolution [79, 80].

Figure 2.2 shows an exemplary SLD resolution tree for the Prolog query `len([1,2], 0, L)` using the predicate `len/3` shown in Figure 2.1. Note that the resolution of the built-in Prolog predicate `is/2` is not shown in the figure. First, a Prolog query is negated for the proof by contradiction. Afterward, the first clause for resolution is searched top down in the list of horn clauses as can be seen in Figure 2.1. The first clause of `len/3` cannot be used for resolution since the first argument of the query, *i.e.*, `[1,2]`, is not equal to the empty list. The second clause of `len/3` is the first clause for resolution with the query deducing the formula ¬(`NAcc is Acc + 1`) ∨ ¬`len([2], NAcc, L)`. When

```
len([], Acc, Acc).
len([_|T], Acc, Len) :-
        NAcc is Acc + 1,
        len(T, NAcc, Len).
```

$$\begin{tikzcd}
\boxed{\neg\texttt{len([1,2], 0, L)}} \arrow[d, "{\{\_/1, \mathrm{T}/[2], \mathrm{Acc}/0, \mathrm{Len}/\mathrm{L}, \mathrm{NAcc}/1\}}"] \\
\boxed{\neg\texttt{len([2], 1, L)}} \arrow[d, "{\{\_/2, \mathrm{T}'/[], \mathrm{Acc}'/1, \mathrm{Len}'/\mathrm{L}, \mathrm{NAcc}'/2\}}"] \\
\boxed{\neg\texttt{len([], 2, L)}} \arrow[d, "{\{\mathrm{Acc}''/2, \mathrm{L}/\mathrm{Acc}''\}}"] \\
\boxed{\bot}
\end{tikzcd}$$

Figure 2.2.: An exemplary SLD resolution tree (right) for the Prolog query `len([1,2], 0, L)` using the predicate `len/3` (left) shown in Figure 2.1. The resolution of the built-in Prolog predicate `is/2` is not shown in the tree.

skipping the evaluation of the built-in Prolog predicate `is/2`, this results in deducing ¬`len([2], 1, L)`. This resolution induced the unification of several variables as can be seen on the edges of the tree in Figure 2.2. Afterward, the deduced clause is used for the next resolution (linear) which again selects the second clause of `len/3`. Finally, the formula ¬`len([], 2, L)` is contradictory to the base case of the predicate `len/3`. We have thus found a solution since we enforce a proof by contradiction. The most-general unifier can be extracted from the edges when following back the path to the root node. In this example, the most-general unifier for the Prolog query is $\{\mathrm{L}/2\}$, which does not provide any choice points.

A choice point occurs if more than one Prolog rule is applicable for a specific goal or subgoal [79, 80, 84]. Backtracking means to return to a previous choice point in a search tree [79, 80, 84]. In chronological backtracking, the search always returns to the last choice point. In Prolog, backtracking is either triggered by a search path that does not result in a solution or by the user who requests to search for another solution. The last choice point is searched bottom-up starting from the current state of the Prolog interpreter. The search is performed lazily, *i.e.*, paths of the search tree possibly leading to a further solution are only evaluated upon request. Possible choice points can be removed by using a so-called cut (`!/0`) in Prolog, which effectively prevents backtracking over this cut on the same level in the search tree [79, 80, 84].

Coroutines

Coroutines in Prolog can be used to suspend the execution of a predicate until a certain condition is met. For instance, one can use `when(+Condition, +Goal)` to delay the call of `+Goal` until the condition is met. In a condition, only a small subset of Prolog can be used, which is `nonvar/1`, `ground/1`, conjunction, disjunction, and unification (`=/2`). For instance, `p(X,Y) :- when(nonvar(X), q(X,Y))` delays the call of `q(X,Y)` until `X` is ground. In SICStus Prolog, it is possible to assign a block declaration for a predicate using `block/?`, where only non-variable checks can be used in a condition [81]. For instance, the above example predicate `p/1` can be encoded using a block declaration as follows:

```
:- block p(-,?).
p(X,Y) :- q(X,Y).
```

The dash indicates that the execution of the predicate is delayed until this argument is ground. A question mark indicates that this argument is not involved in blocking a predicate's execution. While the conditions are more restricted, block declarations are usually more efficient than using `when/2` in SICStus Prolog [81].

2.2. Boolean Satisfiability Solving

The process of deciding the satisfiability of a propositional logic formula is called Boolean satisfiability solving (SAT). This problem was the first one that has been proven to be NP-complete by Stephen Cook [85] and Leonid Levin [86]. The NP-completeness states that there does not exist an efficient decision procedure that is able to solve all Boolean satisfiability problems in polynomial time or less. Nevertheless, there exist decision procedures for the Boolean satisfiability problem which scale for most problems in practice. One of the main foundations in SAT solving is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [11, 12]. Many improvements have been suggested to date such as conflict-driven clause learning [87, 88], variable selection heuristics for branching decisions [89, 90] or restarts [91] with phase saving [92].

In the following, we introduce the DPLL algorithm as well as the concept of watched-literals. Other important features of modern SMT solvers are explained in Section 6.5 alongside our implementation in ProB.

2.2.1. DPLL Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [11, 12] is the foundation of most SAT solvers. Its input is a propositional logic formula in conjunctive normal form while the output is either a model of the input formula, *i.e.*, an interpretation that makes the formula true, or a statement indicating that the input formula is not satisfiable. In its classical form, the key features of the DPLL algorithm are the so-called unit propagation, variable selection, and resolution-based simplification of formulas.

Algorithm 2.2 Pseudocode of the DPLL algorithm's unit propagation [11, 12].

Input/Output a propositional logic formula ϕ in CNF set representation and a partial assignment of variables μ

```
1: procedure UNIT-PROPAGATION($\phi$, $\mu$)
2:     for all $c \in \phi$ do
3:         if $c = \{l\}$ then
4:             $l \leftarrow \top$
5:             $\phi \leftarrow \phi \setminus c$
6:             $\mu \leftarrow \mu \cup \{l = \top\}$
7:         else if $c = \{\neg l\}$ then
8:             $l \leftarrow \bot$
9:             $\phi \leftarrow \phi \setminus c$
10:            $\mu \leftarrow \mu \cup \{l = \bot\}$
11:    return $\phi$, $\mu$
```

Algorithm 2.3 Pseudocode of the DPLL algorithm's resolution-based simplification procedure for propositional logic formulas [11, 12].

Input/Output a propositional logic formula ϕ in CNF set representation

```
1: procedure SIMPLIFY($\phi$)
2:     for all $c \in \phi$ do
3:         for all $v \in c$ do
4:             if $v = \top$ then
5:                 $\phi \leftarrow \phi \setminus c$
6:                 break
7:             else if $v = \bot$ then
8:                 $c \leftarrow c \setminus \{v\}$                    ▷ resolution
9:     return $\phi$
```

The unit propagation states that all clauses of a conjunctive normal form that contain a single literal should lead to the propagation of the literal occurring in a unit clauses with its corresponding polarity. Such clauses are called unit clauses. The justification for unit propagation is that there exists no other assignment for a unit clause to make it true. A pseudocode algorithm of the unit propagation can be seen in Algorithm 2.2. The input of the algorithm is a propositional logic formula in CNF represented as a set of sets as described in Section 2.1.1 as well as a partial assignment of variables. The output of the algorithm is a CNF without unit clauses and an updated partial assignment of variables that have been set by unit propagation. The unit propagation traverses the complete set of clauses. We assume that the variables occurring in a CNF are replaced by their truth value by reference once they are propagated. For instance, the CNF $\{\{\mathrm{A}\}, \{\neg\mathrm{A}, \mathrm{B}\}\}$ becomes $\{\{\top\}, \{\bot, \mathrm{B}\}\}$ after setting the variable A to true.

After setting the truth value of a variable, the DPLL algorithm simplifies a CNF of a propositional logic formula by applying Boolean resolution as can be seen in Algo-

Algorithm 2.4 Pseudocode of the DPLL algorithm [11, 12].

Input a propositional logic formula ϕ in CNF set representation and a partial assignment of variables μ
Output a model μ of ϕ or $\bot$

```
1: procedure DPLL($\phi$, $\mu$)
2:     $\phi, \mu \leftarrow$ UNIT-PROPAGATION($\phi, \mu$)
3:     $\phi \leftarrow$ SIMPLIFY($\phi$)
4:     if $\varnothing \in \phi$ then
5:         return $\bot$
6:     else if $\phi = \varnothing$ then
7:         return $\mu$
8:     else
9:         select any variable $v$ occurring in $\phi$          ▷ choice point
10:        $a \leftarrow \top \lor a \leftarrow \bot$                ▷ choice point
11:        $v \leftarrow a$
12:        $\mu \leftarrow \mu \cup \{v = a\}$
13:        $\phi \leftarrow$ SIMPLIFY($\phi$)
14:        return DPLL($\phi, \mu$)
```

rithm 2.3. This results in removing falsities from clauses since variables are set to their truth values by reference after propagation. Clauses that contain a variable that is true are removed from the set of clauses since they are already satisfied. For instance, the CNF $\{\{\top\}, \{\bot, \mathrm{B}\}\}$ is simplified to $\{\{\mathrm{B}\}\}$.

Algorithm 2.4 shows a recursive pseudocode implementation of the DPLL algorithm for Boolean satisfiability solving. The algorithm's input is a propositional logic formula in CNF represented as a set of sets as well as a partial assignment of variables. Its output is either a model of the formula or $\bot$, which indicates that the input formula is contradictory. The DPLL algorithm starts by applying unit propagation and simplifying a formula afterward. If the set of clauses contains an empty clause, the algorithm has found a contradiction and returns a false statement as can be seen in line 5 of Algorithm 2.4. An empty clause originates from the simplification of a unit clause by resolution as can be seen in line 8 of Algorithm 2.3. For instance, the unit clause $\{\mathrm{A}\}$ results in an empty clause after setting the variable A to false. The DPLL algorithm has found a model of a satisfiable formula if the set of clauses is empty and returns the current set of assignments as can be seen in line 7 of Algorithm 2.4. Otherwise, an arbitrary variable is selected and assigned with an arbitrary truth value. Both of these decisions provide a choice point which makes the DPLL algorithm complete when applying backtracking.

In the following, we give an example of the DPLL algorithm to find a solution for the propositional logic formula $(\mathrm{C} \lor \mathrm{B}) \land (\neg\mathrm{A} \lor \neg\mathrm{B} \lor \mathrm{E}) \land (\neg\mathrm{A} \lor \mathrm{C} \lor \neg\mathrm{E}) \land (\neg\mathrm{B} \lor \neg\mathrm{C}) \land (\mathrm{B} \lor \neg\mathrm{C}) \land (\mathrm{C} \lor \mathrm{D}) \land \mathrm{D}$. For this, we use the set representation of the CNF as well as a binary tree representation of the DPLL algorithm as can be seen in Figure 2.3. Variables are selected lexicographically and first assigned to true for branching decisions. For variable

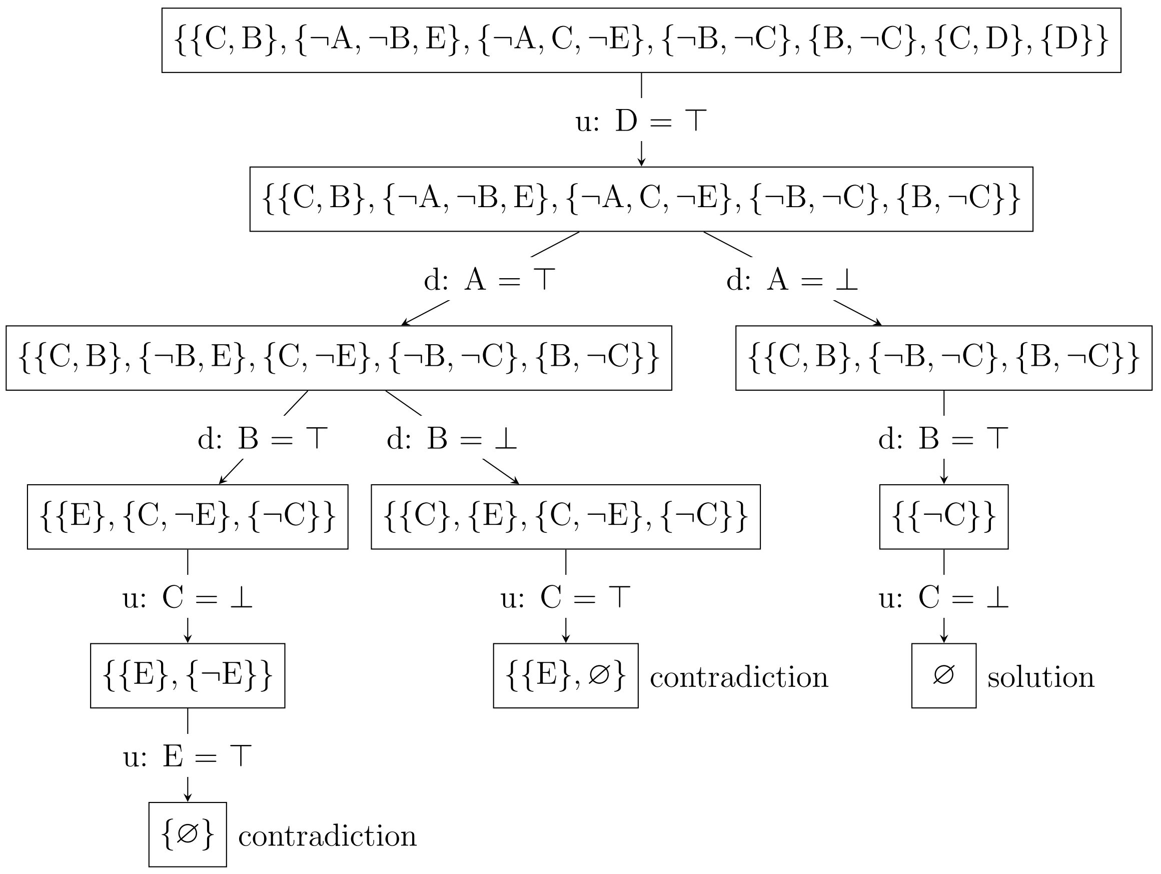

{{C, B}, {¬A, ¬B, E}, {¬A, C, ¬E}, {¬B, ¬C}, {B, ¬C}, {C, D}, {D}}
u: D = ⊤
{{C, B}, {¬A, ¬B, E}, {¬A, C, ¬E}, {¬B, ¬C}, {B, ¬C}}
d: A = ⊤
d: A = ⊥
{{C, B}, {¬B, E}, {C, ¬E}, {¬B, ¬C}, {B, ¬C}}
{{C, B}, {¬B, ¬C}, {B, ¬C}}
d: B = ⊤
d: B = ⊥
d: B = ⊤
{{E}, {C, ¬E}, {¬C}}
{{C}, {E}, {C, ¬E}, {¬C}}
{{¬C}}
u: C = ⊥
u: C = ⊤
u: C = ⊥
{{E}, {¬E}}
{{E}, ∅}
contradiction
∅
solution
u: E = ⊤
{∅}
contradiction

Figure 2.3.: Example of the DPLL algorithm to find a solution for the propositional logic formula $(C \lor B) \land (\neg A \lor \neg B \lor E) \land (\neg A \lor C \lor \neg E) \land (\neg B \lor \neg C) \land (B \lor \neg C) \land (C \lor D) \land D$. Variables are selected lexicographically and first assigned with true for branching decisions.

assignments in Figure 2.3, a prefixed "u:" stands for unit propagation and "d:" for decision.

The DPLL algorithm first applies unit propagation since the clause $\{D\}$ is a unit clause and simplifies the formula by removing the clause $\{C, D\}$ which is true due to the propagation of $D = \top$. Afterward, we can decide for a variable to be assigned next and choose to assign $A = \top$. Again, the current formula is simplified which results in removing the negative literal A from the second and third clauses by resolution. The resulting formula does not contain a unit clause and we thus decide to assign the variable B with true next. This assignment causes the unit propagation $C = \bot$ which itself causes the unit propagation of $E = \top$ in the next step of the algorithm. This propagation results in an empty clause, *i.e.*, a contradiction has been found. The DPLL algorithm now backtracks to the last choice point, which was the decision of $B = \top$. This backtracking causes the decision of $B = \bot$. The resulting simplified formula contains a unit clause which causes the unit propagation of $C = \top$. Again, a contradiction has been found since this propagation results in an empty clause in the set of clauses. The DPLL algorithm now backtracks to the decision of $A = \bot$ and simplifies the formula. Afterward, we decide to assign $B = \top$ according to our variable selection heuristic. This results in a single unit clause which propagates $C = \bot$. The set of clauses is now empty and we thus have found a solution for the propositional logic input formula. The model found by the DPLL algorithm can be extracted from the edges of the binary tree, *i.e.*, the sequence of assigned variables, which is the set of assignments $\{D = \top, A = \bot, B = \top, C = \bot\}$. The other variables do not participate in this partial model and can be selected arbitrarily in order to derive a complete model.

It can be seen that the decision for the variable to assign as well as its polarity influences the performance of the DPLL algorithm. For instance, we would have found the same solution without any backtracking if we had decided to assign $A = \bot$ first instead of $A = \top$. One simple improvement of the DPLL algorithm is the pure literal elimination [11, 12]. A literal is said to be pure if it occurs with the same polarity in all clauses. In this case, a pure literal is set to its polarity assuming that this decision leads to the most amount of simplifications. The application of the pure literal heuristic would have lead to finding the solution of the example presented in Figure 2.3 without any backtracking. Yet, the identification of pure literals is costly since the complete set of clauses has to be traversed after each propagation of a variable. In general, the decision for the assignment of variables (that are no pure literals) is one of the main culprits for a possibly bad performance of the DPLL algorithm since it guides the overall search. In Section 6.5.3, different variable selection heuristics are presented alongside our implementation in ProB.

2.2.2. Watched Literals

Other bottlenecks for performance in the DPLL algorithm are the search for unit clauses as well as the simplification of the set of clauses after the propagation of a variable. The unit propagation and the clause simplification both traverse the complete set of clauses as can be seen in line 2 of Algorithm 2.2 and line 2 of Algorithm 2.3. To improve this, Moskewicz et al. [89] proposed the watched literals scheme. The idea is to store pointers to a subset of variables for each clause. After the propagation of a variable, the affected clauses can be simplified by accessing the corresponding pointers of this variable by reference without the need for iterating over the complete set of clauses. A unit clause is identified if there is only one pointer to a variable in a clause. In practice, it is sufficient to watch two literals for each clause to identify unit clauses which is referred to as the two watched literals scheme [89]. The initial literals to be watched are usually chosen arbitrarily for each clause.

It should be noted that the watched literals scheme does not guarantee the most amount of simplifications for each clause. If a variable is propagated but not watched in a specific clause, this clause will not be simplified. However, this is not an issue since a clause will be simplified as soon as a variable that has already been propagated is selected to be watched. In this case, the variable is not actually watched but the clause gets simplified accordingly. For instance, consider the set of clauses $\{\{\underline{A}, \underline{B}\}, \{\underline{\neg A}, \underline{E}\}, \{\underline{A}, \underline{C}, D\}, \{\underline{B}, \underline{C}, D, E\}\}$ where the two watched literals of each clause are underlined. For this, we have to iterate over the set of clauses once to set up the watched literals. Since there are no unit clauses, we can start by deciding to assign the variable A to true. This causes the first and third clause to be removed from the set of clauses since the variable A is watched which makes these clauses evaluate to true. In the second clause, the watched literal A has a negative polarity so that it is removed from this clause by resolution. This results in the simplified set of clauses $\{\{\underline{\mathrm{E}}\}, \{\underline{\mathrm{B}}, \underline{\mathrm{C}}, \mathrm{D}, \mathrm{E}\}\}$. The watched literals of each simplified clause are updated. Here, this is the case for the first clause where it is noted that there exists no other literal to watch than E. Since this literal is already watched, a unit clause is identified. The unit propagation assigns E to true which results in the simplified set of clauses $\{\{\underline{\mathrm{B}}, \underline{\mathrm{C}}, \mathrm{D}, \top\}\}$. This assignment causes no further simplification since the literal E, which is now true, is not watched in the last remaining clause. We can thus decide to assign the literal B to true which provides an overall solution for the satisfiability of this formula. The fact that clauses are not necessarily simplified if a propagated literal is not watched does not affect the completeness since the DPLL algorithm will find all solutions when backtracking.

SAT solvers that implement the two watched literals scheme do not apply the pure literal elimination since this would again add the overhead of traversing the complete set of clauses after each propagation, or watch the complete set of literals to achieve the most amount of simplifications [93]. However, the identification of a pure literal would then again require a linear search over all literals that a pointer refers to. Furthermore, watching all literals thwarts the initial goal to improve the performance of unit propagation.

2.3. Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the process of deciding the satisfiability of a Boolean formula for some background theories, *e.g.*, the integers or sets of integers. The literature usually distinguishes between eager and lazy SMT solving, which will be discussed in the following.

2.3.1. Eager SMT Solving

In eager SMT solving (aka bit-blasting), all theories are translated to propositional logic while preserving the satisfiability. Afterward, a SAT solver is used to solve a Boolean formula. If a solution exists, it is translated back to its corresponding theories (see, *e.g.*, [94–97]). One benefit of an eager approach is that one can always use state-of-the-art SAT solvers [98]. For instance, the Alloy language [40, 45] is translated to SAT and has been successfully used for many applications[1]. Plagge and Leuschel [46] presented a translation from a subset of the B language to Kodkod [45] which showed benefits for finite relational constraints such as computing the relational closure. Finite sets can be encoded as bit vectors. Therefor, one bit is introduced for each possible element of a set to indicate its presence or absence in the set. Logic operations such as bitwise-and

[1] *e.g.*, see `https://alloytools.org/applications.html`

or bitwise-or can then be used to implement the necessary semantics of set theory. For instance, a set $\{1, 2\} \in \mathbb{P}(\{1, 2, 3\})$ can be encoded as a bit vector with three bits $[1, 1, 0]$, where the zero bit indicates the absence of the integer 3.

Yet, the eager approach to SMT solving is not always beneficial. One downside is that usually only finite domains are supported, which are bounded by a predefined bit width. Besides a lower expressiveness, finite domains entail the occurrence of integer overflows. Despite that, it can be a noticeable overhead to create a CNF of a Boolean formula for SAT solving, especially when encoding several full adders for integer arithmetic [43].

2.3.2. Lazy SMT Solving

In lazy SMT solving (aka DPLL(T) or CDCL(T)) [98–101], SAT solving is combined with theory specific constraint solving. Therefore, a formula in predicate logic is abstracted to a Boolean formula by replacing theory specific predicates with Boolean variables. For instance, the formula $x \in \mathbb{Z} \wedge x > y \wedge y > x$ can be abstracted to $\mathrm{A} \wedge \mathrm{B} \wedge \mathrm{C}$ with $\mathrm{A} \equiv x \in \mathbb{Z}$, $\mathrm{B} \equiv x > y$, and $\mathrm{C} \equiv y > x$. Afterward, a SAT solver is used to solve the Boolean abstraction. If a solution is found, it is translated back to predicate logic to be sent to theory specific constraint solvers. For instance, we need a constraint solver for linear integer arithmetic in the example above. Here, a SAT solver finds the only solution $\mathrm{A} = \top \wedge \mathrm{B} = \top \wedge \mathrm{C} = \top$. If a Boolean variable is assigned a negative polarity, its corresponding predicate is negated in predicate logic. The translated model is then sent to the constraint solver for integer arithmetic which in this case would refute the formula since it is not satisfiable. The SMT solver then backtracks into the SAT solver and uses the found model for CDCL to prevent finding the same solution again in an ongoing search [87, 88]. If the SAT solver finds a different solution, this interaction between the SAT solver and the theory solver repeats until the theory solver reports satisfiability. In this example, there exists no other Boolean model so that the formula is found to be unsatisfiable.

Modern lazy SMT solvers implement different improvements such as early pruning or theory propagation, which are discussed in Section 6.5.4 along our implementation in ProB. Furthermore, SAT and SMT solvers do not implement the pure-literal elimination of the classic DPLL algorithm anymore due to efficiency reasons [93, 99]. Techniques such as non-chronological backtracking (backjumping) have proven to be more critical for the performance of constraint solving.

Lazy SMT solving has been successfully applied for many different theories[2]. The main benefit compared to other constraint solving approaches is that lazy SMT solvers benefit from advantages in SAT solving such as CDCL. Furthermore, it can be the case that a SAT solver is able to disprove a formula's Boolean abstraction so that no theory solver has to be called, which might not be able to disprove the corresponding formula in predicate logic.

If not stated otherwise, we always mean lazy SMT solving when referring to SMT solving in the remainder of this thesis.

[2] *e.g.*, see `https://smtlib.cs.uiowa.edu/benchmarks.shtml`

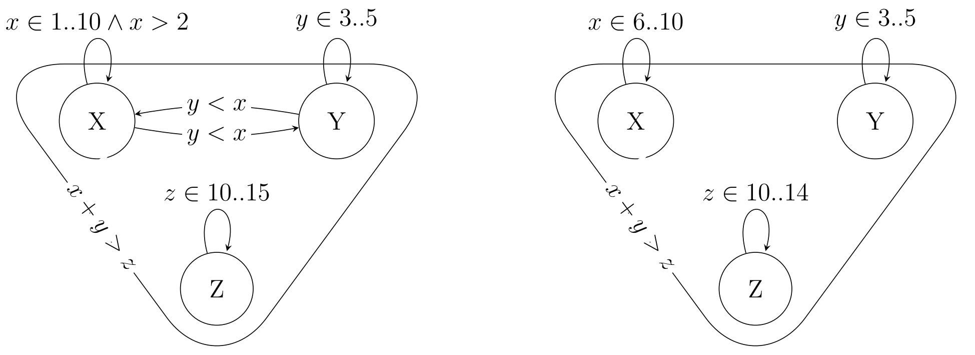


Figure 2.4.: The left-hand side shows a hypergraph representation of the constraint $x \in 1..10 \land y \in 3..5 \land z \in 10..15 \land x > 2 \land y < x \land x + y > z$. The right-hand side shows a hypergraph of the same constraint but with consistent domains, where consistent constraints have been removed.

2.4. Constraint Logic Programming

Constraint logic programming (CLP) [84, 102–105] generally aims at solving first-order logic formulas. The main concept is to maintain variable domains that are locally consistent with all available constraints by applying algorithms to reduce the domains of variables when new constraints are posted. A contradiction is identified if a domain becomes empty, which usually triggers chronological backtracking. After the phase of the domain consistency check, solutions can be found by enumerating the remaining domains. Constraint logic programming allows reasoning over unbounded domains. Yet, it is incomplete regarding the disproving of formulas over unbounded domains since unbounded domains cannot be enumerated exhaustively. For instance, a CLP solver is usually not able to disprove the constraint $x \in \mathbb{Z} \land x > y \land y > x$ since the domains of x and y cannot be narrowed down.

In common Prolog implementations, a CLP(X) scheme is defined where X stands for a specific theory. For instance, CLP(FD) allows reasoning over finite domain (FD) integers [30]. Other systems allow reasoning over reals (CLP(R)), rationals (CLP(Q)) or Boolean values (CLP(B)).

A constraint can be represented as a directed graph $G = (V, E)$ where a node is introduced for each variable occurring in the constraint [84]. Constraints are represented as directed edges depending on their arity. Let τ be a function assigning variables to nodes in V. Unary constraints, *i.e.*, constraints referring to a single variable, result in self loops labeled with the constraint. For a binary constraint referring to the variables x and y, two directed edges $(\tau(x), \tau(y))$ and $(\tau(y), \tau(x))$ are added to the graph [84]. A binary constraint results in two edges since the constraint is supposed to hold in both directions. For a k-nary constraint with $k > 2$, an edge between all k variables pointing in all directions is added to E making the graph G a hypergraph [84]. An exemplary constraint graph containing a unary, binary, and ternary constraint can be seen on the left-hand side of Figure 2.4.

2.4.1. Domain Consistency

Constraint logic programming applies algorithms for the reduction of domains in order to solve a constraint system. A classic notation of consistency is defined as follows [84, 102, 103]. Let $\text{dom}(v)$ be the domain of the variable $v \in V$, and C_v be the set of unary constraints referring to the variable v, $C_{v,w}$ be the set of binary constraints referring to the variables $v, w \in V$, and $C_{k_1,\dots,k_n}$ be the set of k-nary constraints with $k > 2, n \in \mathbb{N}, k_1, \dots, k_n \in V$. The node-consistency states that a node v is consistent if all of its domain values are consistent with all unary constraints, which can be formalized as $\forall x.(x \in \text{dom}(v) \wedge c \in C_v \Rightarrow c(x))$. Arc-consistency defines that a binary constraint is satisfiable for each assignment of variables occurring in this constraint. Let $v, w \in V$ be variables occurring in a binary constraint $c \in C_{v,w}$. The edge $(v, w) \in E$ is arc-consistent if $\forall x.(x \in \text{dom}(v) \Rightarrow \exists y.(y \in \text{dom}(w) \Rightarrow c(v, w))$ holds. The same has to apply for the edge (w, v). For k-nary edges, the definition is similar but using $k - 1$ existentially quantified variables called k-consistency or path-consistency. For instance, a ternary edge (u, v, w) is consistent if $\forall x.(x \in \text{dom}(u) \Rightarrow \exists(y, z).((y \in \text{dom}(v) \wedge z \in \text{dom}(w)) \Rightarrow c(u, v, w))$ holds. Again, this definition has to hold in each direction, *i.e.*, for the edges (v, u, w) and (w, u, v).

A relaxed definition of the domain consistency is the bounds consistency where only the minimum and maximum value of a domain are checked for consistency [84, 103]. Hereby, it is not guaranteed that the values in between the bounds of a domain are consistent.

There exist different algorithms to achieve consistent domains depending on the arity of a constraint. In short, each universally quantified value which does not satisfy one of the above definitions is removed from its domain [84, 102, 103]. Usually, a constraint is checked for consistency as soon as it is added to the constraint system, which is why constraint logic programming is said to be data-driven. A constraint is identified to be inconsistent if the domain of a variable becomes empty during the reduction of domains. At the right-hand side of Figure 2.4, a consistent constraint system represented as a hypergraph is shown, which originates from the constraint system shown on the left-hand side of Figure 2.4. Here, the domain of the variable x has been reduced to the interval 3..10 to achieve node-consistency regarding the constraint $x > 2$. Afterward, the binary constraint $y < x$ is already consistent but the ternary constraint $x + y > z$ leads to further reductions according to the definition of 3-consistency. In particular, the domain of the variable x is reduced to the interval 6..10 and the domain of the variable z to 10..14. It should be noted that a constraint solver based on CLP can apply arbitrary algorithms for the reduction of domains.

2.4.2. Labeling

There are two different outcomes after reaching domain consistency in a constraint system. On the one hand, it can be the case that all domains of variables contain a single value, *i.e.*, the constraint system only has a single solution. On the other hand, consistent domains can contain several elements. In this case, a so-called labeling (aka grounding) is used to find ground values by enumerating the domains of variables, *i.e.*, assigning variables to domain values [30, 84, 102, 103]. The order of variables to assign and elements of domains to enumerate can usually be guided by some heuristic [30, 84, 102, 103], *e.g.*, sorting variables lexicographically and enumerating domains in ascending natural order.

For instance, the consistent constraint system shown on the right-hand side of Figure 2.4 can be labeled to $x = 6 \wedge y = 5 \wedge z = 10$. This can be achieved by successively labeling the variables x, y, and z starting with propagating $x = 6$. Afterward, the ternary constraint $x + y > z$ becomes binary. The arc-consistency then induces the reduction of the domain of the variable y to the single value 5. Otherwise, the intermediate constraint $6 + y > z$ with $y \in 3..5$ and $z \in 10..14$ is not satisfiable. Subsequently, the node-consistency induces the reduction of the domain of the variable z to the value 10 due to the intermediate unary constraint $11 > z$ with $z \in 10..14$. Possible further solutions can be enumerated by backtracking. In particular, the enumeration of variable domains provides choice points. In the example from above, the last choice point is in the enumeration of the domain of the variable x, which provides the assignment $x = 7$ after backtracking.

Algorithms for labeling can allow optimizing specific variable values. For instance, one can define to minimize or maximize the value of a specific variable during labeling.

2.4.3. Constraint Reification

Different constraint systems and constraint solvers can be combined by using specific operators for constraint reification, *e.g.*, as is the case for SMT solving as described in Section 2.3.2. The constraint reification allows reflecting the truth value of a constraint into a Boolean value. Usually, the provided operators comprise all common logical operators, *i.e.*, the logical conjunction, disjunction, negation, implication, and equivalence. For instance, a constraint reification is given by $\mathrm{A} \equiv x > 0$, where A is a Boolean variable, and $\equiv$ is a reification constraint.

2.5. Formal Methods and Formalisms

Formal methods are an approach for the design and verification of software and hardware systems. For software, this usually also involves generating executable code from verified specifications. A key feature of the approach is the specification of system properties which can be mathematically proven to hold for a system's whole lifetime. In the following, we give a brief overview of the most important theoretical aspects of formal methods that are necessary for this thesis. Additionally, we introduce two prominent implementations of formal methods called the B-Method and Alloy. It should be noted that there exist many more implementations of formal methods such as TLA^{+}[41], Promela [42] or the Vienna Development Method (VDM) [106], which are not discussed in this thesis.

One fundamental concept of state based formal methods are transition systems. A transition system is a directed graph that consists of a set of states S, a set of actions, a

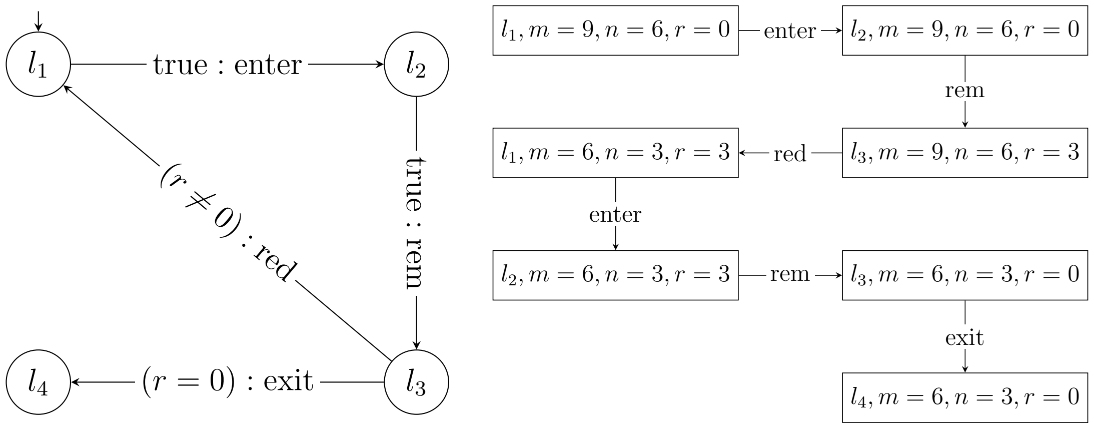

l_1
true : enter
l_2
true : rem
$(r \neq 0)$: red
l_4
$(r = 0)$: exit
l_3
$l_1, m = 9, n = 6, r = 0$
enter
$l_2, m = 9, n = 6, r = 0$
rem
$l_1, m = 6, n = 3, r = 3$
red
$l_3, m = 9, n = 6, r = 3$
enter
$l_2, m = 6, n = 3, r = 3$
rem
$l_3, m = 6, n = 3, r = 0$
exit
$l_4, m = 6, n = 3, r = 0$

Figure 2.5.: A program graph of the Euclidean algorithm using division [3] (left) as well as a transition system that has been unfolded from the program graph for the inital state $m = 9, n = 6, r = 0$ (right). The actions enter and exit do not have any effect on the evaluation of variables while rem is the action $r := m \bmod n$ and red is the action $m := n; n := r$. The semicolon stands for a sequential execution.

transition relation, a set of initial states, a set of atomic propositions AP, and a labeling function $L : S \to \mathbb{P}(\mathrm{AP})$ [107]. The power set of all atomic propositions in AP is defined by $\mathbb{P}(\mathrm{AP})$, which defines all possible combinations of labels. Atomic propositions are logical formulas corresponding to specified system properties. The labeling function relates each state s of a transition system to a set of atomic propositions $L(s)$. A state s is said to satisfy a formula Φ if $L(s)$ makes Φ true, *i.e.*, $s \models \Phi \equiv L(s) \models \Phi$ [107].

Program graphs are directed graphs that can be used to describe a system's general behavior abstracted from explicit state changes [107]. The nodes of a program graph represent locations in a system instead of explicit states as it is the case for transition systems. A program graph reasons over a set of variables and actions, and defines a so-called effect function, which is a successor state relation resorting to the provided actions [107]. Each action has a guard (also called precondition) that enables or disables the action for a specific evaluation of variables. The edges in a program graph correspond to an action with its corresponding guard, which are usually divided by a colon in visual and textual representations. One is able to generate a transition system of a program graph for one or more initial states. Here, a state in the transition system corresponds to an evaluation of variables for an explicit execution in the program graph. Successor states are computed by applying the effect function, *i.e.*, evaluating the effect of actions if their corresponding guard is satisfied in the current state. Edges of a transition system are thus labeled with actions implying that their guard is true for the starting node's state. The complete set of a transition system's states for one or more initial states is referred to as its state space [107].

A program graph of the Euclidean algorithm using division [3] can be seen on the left-hand side of Figure 2.5. We use the following definition of the Euclidean algorithm:

“Given two positive integers m and n, find their greatest common divisor, that is, the largest positive integer that evenly divides both m and n.

E1. [Find remainder.] Divide m by n and let r be the remainder. (We will have $0 \leq r < n$.)

E2. [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.

E3. [Reduce] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1.” [3, p.2, Algorithm E]

We realized the go-to statement in E3 using a while loop with a condition that is always true. The program graph defines four locations corresponding to the condition of the while loop (l_1), the computation of the remainder in the loop's body (l_2), the check for termination in the loop's body (l_3), and the return statement after the loop (l_4) of the Euclidean algorithm using division. An exemplary transition system that has been unfolded from the program graph for the initial state $m = 9$, $n = 6$ and $r = 0$ can be seen on the right-hand side of Figure 2.5. The transition system has been unfolded exhaustively using the program graph's actions until reaching location l_4, which results in $n = 3$. The actions enter and exit do not have any effect on the evaluation of variables, *i.e.*, they assign each variable to its current value (skip). The action rem corresponds to $r := m \bmod n$ and red corresponds to $m := n; n := r$, where the semicolon stands for a sequential execution.

For a more detailed description of the theoretical foundations of formal methods and model checking, we refer the reader to the work by Beier and Katoen [107] or Clarke et al. [108]. Of course, there is also other brilliant literature on this topic.

2.5.1. Linear Time Properties

One foundation in formal methods is the specification of linear time (LT) properties to specify and verify certain behavior. There are different kinds of LT properties each being defined over the set of atomic propositions AP [107]. A sequence of states in a transition system is called a path fragment [107]. If a path fragment starts in an initial state and either ends in a terminal state or is infinite, it is called a path of a transition system [107]. A system's behavior is represented by traces, which are sequences of labels from paths. By definition, a linear time property is an ω regular expression $\mathbb{P}(\mathrm{AP})^\omega$ [107], where $\mathbb{P}(\mathrm{AP})^\omega$ “denotes the set of words that arise from the infinite concatenation of words in” $\mathbb{P}(\mathrm{AP})$ [107, p.100, Definition 3.10].

The main types of linear time properties are safety, liveness, and fairness properties. Safety properties generally describe some bad behavior that should not happen [109], and are possibly violated by finite paths. An invariant is a safety property that has to hold in every reachable state of a transition system [107]. Every system that does not do anything complies with all its safety properties. Thus, liveness properties are used to describe good behavior of a system that should happen eventually. In contrast to safety properties, liveness properties are violated in infinite time [107]. Fairness properties are LT properties that can be used to rule out unrealistic behaviors of a system [107].

Fairness is usually required to prove liveness properties. For instance, a system where several processes demand access to a certain area might require a fairness constraint to prevent the starvation of certain processes. There exist different kinds of fairness constraints (weak, strong and unconditional) [107], which will not be discussed any further here.

In the remainder of this thesis and especially in the tools presented in this thesis, we are mainly concerned with safety properties and in particular invariants.

2.5.2. Linear Temporal Logic

Linear temporal logic (LTL) [107] is a logical formalism for specifying safety and liveness properties over a set of atomic propositions. Further, LTL is a subset of first-order logic. Besides the common Boolean operators, LTL provides the timed operators $X\Phi$ (next) and $\Phi\ \mathcal{U}\ \Psi$ (until), where Φ and Ψ are logical formulas. The next operator states that Φ has to hold in the next state, while $\Phi\ \mathcal{U}\ \Psi$ states that Φ has to be true until Ψ becomes true (not necessarily including this state).

One is able to derive several more LTL operators using these two definitions [107]. For instance, a formula can be checked to finally hold using $\Diamond$ which is defined as $\Diamond\,\Phi \equiv$ true $\mathcal{U}\ \Phi$, or to hold in every state (globally) using $\Box$ which is defined as $\Box\,\Phi \equiv \neg\Diamond\,(\neg\Phi)$. LTL operators can also be combined. For instance, $\Box\Diamond\ \Phi$ states that Φ has to hold infinitely often, and $\Diamond\Box\,\Phi$ states that Φ eventually holds forever. Other common LTL operators are $\Phi\ \ W\ \ \Psi \equiv (\Phi\ \ \mathcal{U}\ \ \Psi) \vee \Box\ \Phi$ (weak until) and $\Phi\ \ R\ \ \Psi \equiv \neg(\neg\Phi\ \ \mathcal{U}\ \ \neg\Psi)$ (release) [107]. Note that it is also possible to encode fairness constraints in LTL [107], which will not be discussed any further here.

2.5.3. Model Checking

In general, the goal of model checking is to verify that an LT property holds for a system. Explicit-state model checking is concerned with verifying safety properties (in particular, invariants) of finite systems by checking the complete state space exhaustively. If a state violating a property is found, a finite path and trace leading to this counterexample is returned [107]. A constraint solver is essential for model checking. It has to compute the effect of state changes during animation and verify properties in each state for invariant checking. The verification of properties in a single state can be a complex task, *e.g.*, resulting in minutes being spent for constraint solving. All in all, explicit-state model checking can be a sophisticated task taking days to terminate for large models depending on the size of its state space and complexity of constraints to be checked.

Symbolic model checking is a different approach that does not view states explicitly but symbolically [110–113]. Several states can be abstracted by predicates imposing constraints on the domains of variables. For instance, consider a single machine variable x. The explicit states $x = 0$, $x = 1$, and $x = 2$ can, *e.g.*, be represented by the predicate $x \in \{0, 1, 2\}$ or $0 \leq x \leq 2$. State changes (actions) can be described by so-called before-after predicates. Here, an action is described by a predicate $\mathrm{BA}_{\mathrm{act}}(v, v')$ applying the action act to the variables in v and assigning the results to the variables in v', which corresponds to the next state [114]. For instance, consider the action `act` defined as `x>0 : x:=x+1` with $\mathrm{BA}_{\mathrm{act}}(\{x\}, \{x'\}) := \exists x.(x > 0 \land x' = x + 1)$. Let Acts be the set of all actions. A state space can be represented symbolically by a monolithic transition predicate $\mathrm{T}(v, v') = \bigvee_{\mathrm{act} \in \mathrm{Acts}} \mathrm{BA}_{\mathrm{act}}(v, v')$ [114]. There are different approaches for symbolic model checking such as bounded model checking (BMC)[115], interpolation [116], k-induction [117] or the IC3 algorithm [118]. While BMC can only be used to check finite paths, the other approaches can also be used for checking infinite state spaces. In contrast to explicit-state model checking, symbolic model checking can speed up the verification process and allows checking infinite state spaces. Yet, the constraints can get complex and hard to solve. For instance, this depends on the sizes of the variables' domains and the amount of variables or actions. All in all, model checking requires a complete constraint solver for the underlying specification language providing the best possible performance.

There exist techniques for model checking of LTL properties based on infinite paths and so-called lasso-traces [107], which also require constraint solving.

2.5.4. The B-Method

The B-Method [25] is a state-based formal method for hardware and software development following the correct-by-construction approach. Here, the correctness refers to a system's specified LT properties. The development in B is incremental starting with an abstract specification that is successively refined and decomposed. Hence, a model in B consists of a collection of so-called machines. Machines are linked by proof obligations which have to be discharged in order to ensure correct behavior of the whole system. The concept of refinements generally increases the maintainability and eases the development in B, especially for complex models.

The B-Method's foundation is its rich specification language, which is based on first-order logic, typed Zermelo–Fraenkel set theory with the axiom of choice [119, 120], and integer arithmetic. The B language defines expressions, predicates, and substitutions. Expressions can be seen as values or formulas evaluating to a discrete value such as an integer, but do not have any side effect. Predicates, on the other hand, are logical formulas that evaluate to either true or false such as a conjunction. Here, it should be noted that B distinguishes between the truth values of predicates and Boolean. Substitutions are used to change the state of a machine, *i.e.*, the set of machine variables. For a complete definition of the B language, we refer the reader to the work by Jean-Raymond Abrial [25]. After the validation or a formal model in B, it is usually transferred to a low-level subset of B called B0 [121]. For instance, sets and quantifiers are replaced for translating to B0. There exist different code generators that are able to generate executable code from formal models using B0 [121–126].

The B language is nowadays referred to as classical B while Event-B [26] is its successor. Event-B extends and improves the classical B language, *e.g.*, by renaming ambiguous operators such as `-` or `*`, or by adding the operators finite and partition. Further, Event-B puts the focus on systems modeling by extending the concept of refinement. Instead of modeling single software or hardware components using the B-Method and classical

B, Event-B enables the design of complete systems. Note that in Event-B preconditions are called guards and substitutions are called actions while referring to nearly the same concepts. For a detailed comparison of B and Event-B, we refer the reader to the work by Michael Leuschel [127].

2.5.5. Alloy 5

Alloy is a formal specification language for modeling software systems [39, 40]. The Alloy Analyzer is used to verify behavior and prove a system's overall correctness regarding specified properties by applying SAT solving. Further, the Alloy Analyzer can be used for the visualization and animation of formal models. Note that recently a new version of Alloy, namely Alloy 6, has been released. In Section 5.3, we give a brief introduction to new features of Alloy 6. If not stated otherwise, we always refer to Alloy 5 when writing Alloy in this thesis.

The Alloy language is based on first-order logic and n-ary finite relations. For instance, sets are unary relations. The language resembles object-oriented programming languages making it more accessible for software developers that have no experience in formal modeling. Formulas are translated to propositional logic and SAT solvers are used for constraint solving. This implies that only finite systems can be modeled. In particular, one has to specify finite bounds for each domain.

Alloy has been used for many applications such as enterprise modeling, electronic commerce or access control and security policies [3].

2.6. ProB

ProB [28, 29] is a tool for designing, analyzing, and verifying formal models using the B-Method. Its main features are the animation of formal models for analysis and debugging, model checking for verification or model finding, and constraint solving, which is used for many tasks such as computing state changes during animation or verifying properties for model checking.

ProB is certified T2 SIL4 according to the EN 50128 [38] standard defined by the European Committee for Electrotechnical Standardization (CENELEC). This is an important aspect in order to use ProB for the design and verification of safety critical systems. For instance, ProB was used at Thales for the validation of the European Train Control System (ETCS) hybrid level 3 principles in the context of railways [16].

Other use cases of ProB are, *e.g.*, data validation [27, 128–130], constraint-based inductive invariant and deadlock checking or test-case generation, which will not be discussed any further here.

Besides the classical B language, ProB also supports other formal specification languages such as Event-B [26], Z [131], TLA$^+$[41, 132, 133] or Alloy 5 [39, 40, 43, 44].

[3] A list of applications using Alloy can be found here: `https://alloytools.org/citations/case-studies.html`

ProB supports explicit-state [29] and symbolic model checking [114]. Explicit-state model checking can be a sophisticated task depending on the size of a model's state space. To improve the performance, ProB provides different techniques such as partial order reduction [134, 135] or operation caching [136]. For symbolic model checking, ProB implements bounded model checking, k-induction as well as the IC3 algorithm [114]. Constraints from symbolic model checking can be complex. It is thus crucial to have a performant constraint solver.

ProB's constraint solver is implemented in SICStus Prolog [81] and uses constraint logic programming, in particular, for the enumeration of finite domain integers [30]. The constraint solver uses coroutines for deterministic propagation and constraint reification. In general, ProB's constraint solving routine can be divided into a deterministic propagation phase, where domains are reduced, and a grounding phase, where domains are enumerated to find exact values. One important feature of ProB's constraint solver is that it is able to provide counterexamples for unsatisfiable constraints. This is particularly desirable for debugging formal models to facilitate their repair.

Set cardinality constraints are frequently used in B and often pose complex constraints depending on the set. For instance, computing the cardinality of a set comprehension can be expensive if the actual set cannot be computed statically but depends on other variables occurring in a constraint. To improve the performance, sets occurring in a cardinality constraint are represented by bit vectors as soon as they are found to be finite. Here, a bit is reserved for each element which is set to 1 if a specific element is part of the set as is the case for eager SMT solving as described in Section 2.3.1. The actual cardinality of a set can then simply be computed using a constraint summing the elements of a bit vector as is provided by CLP(FD) [30]. Again, this is implemented using coroutines in Prolog for a delayed constraint propagation.

ProB's constraint solver implements many more sophisticated performance improvements. One such improvement is called common subexpression elimination (CSE), which is of interest in the context of SMT solving and thus also important for this thesis. CSE aims at collecting syntactically equivalent subformulas (expressions or predicates) in a formula to be replaced by a single new variable holding the subformula's value. This prevents multiple computations of the same subformula. For instance, the formula $f \in \mathbb{Z} \leftrightarrow (\mathbb{Z} \leftrightarrow \mathbb{Z}) \land x \in \mathrm{dom}(f(a)) \ \land \ r = f(a)(x)$ can be optimized to be $f \in \mathbb{Z} \leftrightarrow (\mathbb{Z} \leftrightarrow \mathbb{Z}) \land (\texttt{LET}\ t\ \texttt{BE}\ t = f(a)\ \texttt{IN}\ x \in \mathrm{dom}(t) \land r = t(x)\ \texttt{END})$. Here, the function application $f(a)$ has been lifted into a `LET` expression introducing a new variable t to prevent computing it twice.

A similar approach preventing multiple computations of the same subexpression is memoization, which is implemented in ProB's constraint solver for function applications as well as membership constraints. In contrast to CSE, this optimization is not computed statically but dynamically by managing an internal data structure to look up subexpressions that have already been evaluated.

Due to the complexity of constraint solving, there is probably no constraint solver that is best for all types of constraints. ProB thus also has other additional constraint solving backends, *e.g.*, resorting to external constraint solvers such as Z3 [47, 48] or Kodkod [45, 46]. Further, ProB provides an additional backend using constraint handling rules (CHR), which mainly aims to improve disproving of constraints, especially over unbounded domains. Constraint handling rules [137] provide a constraint store and allow defining rules that are triggered when adding constraints to manipulate the constraint store. This enables to implement constraint solvers, *e.g.*, in Prolog [138–140]. The CHR language is committed-choice, *i.e.*, once a rule is applied it cannot be undone by backtracking. All available constraints are propagated until the constraint store reaches a fix point, *i.e.*, no further CHR can be propagated. A contradiction is detected if the domain of a variable becomes empty. For ProB's CHR backend, propagations rules that are tailored towards finding contradictions were implemented, which are more aggressive than the propagation rules of ProB's native constraint solver. Yet, these additional rules might slow down the search for finding solutions, which is the reason why they are not used in the native constraint solver by default.

3. Thesis Structure

This is a cumulative dissertation. The main part of this thesis (Chapter 4 and Chapter 6) consists of two journal articles [43, 50] that originate from conference paper submissions [44, 49] as can be seen in the following:

[43] Sebastian Krings, Michael Leuschel, Joshua Schmidt, David Schneider, and Marc Frappier. Translating Alloy and Extensions to Classical B. *Science of Computer Programming*, 188, 2020. doi: 10.1016/j.scico.2019.102378

[44] Sebastian Krings, Joshua Schmidt, Carola Brings, Marc Frappier, and Michael Leuschel. A Translation from Alloy to B. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 10817 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2018. doi: 10.1007/978-3-319-91271-4_6

[49] Joshua Schmidt and Michael Leuschel. Improving SMT Solver Integrations for the Validation of B and Event-B Models. In Alberto Lluch Lafuente and Anastasia Mavridou, editors, *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 12863 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2021. doi: 10.1007/978-3-030-85248-1_7 (nominated for best paper award)

[50] Joshua Schmidt and Michael Leuschel. SMT Solving for the Validation of B and Event-B Models. *International Journal on Software Tools for Technology Transfer*, 24(6):1043–1077, 2022. doi: 10.1007/s10009-022-00682-y

Furthermore, we provide additional experiments and considerations for each journal article (Chapter 5 and Chapter 7). Last but not least, we present a final conclusion answering the initial research questions.

The algorithms presented in this thesis are implemented in the ProB tool which is the foundation of this work. The tool is available under the Eclipse Public License Version 1.0 and can be downloaded from `https://prob.hhu.de/`.

In the following, I state my contributions to each aforementioned publication, contributions that explicitly originate from one of my co-authors, as well as adaptions that were made for this thesis compared to the original articles.

A Translation from Alloy to B. The research idea of translating Alloy to B stems from Sebastian Krings, who was also the initiator of writing the paper. The introduction [44, Section 1], translation example [44, Section 2], and conclusion [44, Section 7] originate from Sebastian Krings. The description of improvements over existing Alloy tools [44, Section 6] originates from Michael Leuschel. Carola Brings was not involved in writing the paper but worked out an initial proof of concept in her bachelor thesis [141]. Further, the translation example [44, Section 2] originally stems from her thesis. Marc Frappier was involved in proofreading and provided Alloy models for testing and improving the software. I was involved in research as well as the formulation, description, and validation of the fundamental translation rules from Alloy to B. Both the translation of orderings [44, Section 4] and the empirical evaluation [44, Section 5] originate from me. Besides that, I was also involved in proofreading. Furthermore, I presented the work at the 6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ) in 2018 [142]. Besides the conference paper itself, I was a core developer of the translation from Alloy to B covering the fundamental semantics as well as the integration in ProB (the first version was implemented in Kotlin).

Translating Alloy and Extensions to Classical B. An extended version of the conference paper [44] was submitted to the Science of Computer Programming journal on invitation of the ABZ 2018 conference's editors. This journal article is used as is in Chapter 4 of this cumulative dissertation. Only the captions of figures and listings were adapted grammatically without changing the semantics to match the remaining thesis' style. I was involved in the conceptualization of the article, research, investigation, validation, and formalization of the fundamental translation rules from Alloy to B (Section 4.4). The following texts mainly originate from me:

- primer on B (Section 4.2.2) and comparison of Alloy and B (Section 4.2.3)
- translation of field quantifications and sequences (9th paragraph in Section 4.4.3 and following)
- translation of universe and identity (2nd paragraph in Section 4.4.4 and following)
- translation of Alloy commands (4th paragraph in Section 4.4.14 and following)
- post-processing optimization rules (Section 4.4.16)
- translation of Alloy extensions (Section 4.5)
- empirical evaluation (Section 4.7) except Listing 4.6

Figure 4.1 was created by Sebastian Krings, and Figure 4.2 was created by Michael Leuschel. The introduction in Section 4.1, translation example in Section 4.3, first paragraph in Section 4.4.4, and first three paragraphs in Section 4.4.14 as well as the translations of connectives and simple predicates in Section 4.4.5, quantifications, set comprehensions, and identifiers in Section 4.4.11, and fact, function, and predicate declaration in Section 4.4.13 originate from Sebastian Krings. The overview of the semantic functions in Section 4.4.1 and Section 4.4.2, discussion on representing tuples in Section 4.4.7, translations of Alloy's Cartesian product in Section 4.4.8, domain and range restriction in Section 4.4.9, and join operator in Section 4.4.10 as well as Section 4.6, Listing 4.6, and Section 4.8 originate from Michael Leuschel. Section 4.9 and Section 4.10 mainly originate from Sebastian Krings and Michael Leuschel. The primer on Alloy in Section 4.2.1 and the translation of multiplicity annotations in Section 4.4.15 originate from Marc Frappier. Further, Marc Frappier was involved in proofreading and provided Alloy models for testing and improving the software. David Schneider was involved in proofreading, improving the parser's implementation, and testing the software. Besides the aforementioned contributions, I was involved in proofreading, experimental evaluation, and performance improvements. Further, I am a core developer of the final version of the translation from Alloy and extensions to classical B in ProB using Prolog instead of Kotlin.

Improving SMT Solver Integrations for the Validation of B and Event-B Models. I am the main author of the conference paper and the core developer of the presented new integration of Z3 in ProB. Furthermore, I presented the work at the 26th International Conference on Formal Methods for Industrial Critical Systems (FMICS) in 2021 [143]. Michael Leuschel was involved in proofreading the conference paper and validating the software. The paper was nominated for the best paper award of the conference.

SMT Solving for the Validation of B and Event-B Models. An extended version of the conference paper [49] was submitted to the International Journal on Software Tools for Technology Transfer on invitation of the FMICS 2021 conference's editors. This journal article is mainly used as is in Chapter 6 of this cumulative dissertation. The Venn diagrams in Section 6.7 have been redrawn to match the remaining thesis' style. The contents of the diagrams were not adapted. The subsections “Basics of SAT & SMT Solving” [50, Section 2.1] and “Watched Literals” [50, first paragraph in Section 5.3] have been removed since a more detailed description is provided in Chapter 2. Further, the rules for rewriting set cardinality and power set constraints in Section 6.4.1 have been extended. I am the main author of the journal article and the core developer of the presented new integration of Z3 in ProB and the direct implementation of SMT solving in ProB including the additional theory solver for integer difference logic. Michael Leuschel was involved in proofreading the journal article as well as validating and improving the software.

Other peer-reviewed publications.

[144] Joshua Schmidt, Sebastian Krings, and Michael Leuschel. Interactive Model Repair by Synthesis. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 9675 of *Lecture Notes in Computer Science*. Springer, 2016. doi: 10.1007/978-3-319-33600-8_25

[145] Joshua Schmidt, Sebastian Krings, and Michael Leuschel. Repair and Generation of Formal Models Using Synthesis. In *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 11023 of *Lecture Notes in Computer Science*. Springer, 2018. doi: 10.1007/978-3-319-98938-9_20

[146] Alexandros Efremidis, Joshua Schmidt, Sebastian Krings, and Philipp Körner. Measuring Coverage of Prolog Programs Using Mutation Testing. In Josep Silva, editor, *Proceedings WFLP (International Workshop on Functional and (Constraint) Logic Programming)*, volume 11285 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2018. doi: 10.1007/978-3-030-16202-3_3

[147] Jannik Dunkelau, Sebastian Krings, and Joshua Schmidt. Automated Backend Selection for ProB Using Deep Learning. In Julia M. Badger and Kristin Yvonne Rozier, editors, *Proceedings NFM (International Symposium on NASA Formal Methods)*, pages 130–147. Springer, 2019. doi: 10.1007/978-3-030-20652-9_9

[148] Sebastian Krings, Philipp Körner, and Joshua Schmidt. Experience Report on an Inquiry-Based Course on Model Checking. In Veronika Thurner, Oliver Radfelder, and Karin Vosseberg, editors, *Tagungsband des 16. Workshops "Software Engineering im Unterricht der Hochschulen"*, volume 2358 of *Proceedings CEUR*, pages 87–98. CEUR-WS.org, 2019

[140] Sebastian Krings, Joshua Schmidt, Patrick Skowronek, Jannik Dunkelau, and Dierk Ehmke. Towards Constraint Logic Programming over Strings for Test Data Generation. In *Proceedings WLP (Workshop on (Constraint) Logic Programming)*, volume 12057 of *Lecture Notes in Computer Science*, pages 139–159. Springer, 2020. doi: 10.1007/978-3-030-46714-2_10

[149] Jannik Dunkelau, Joshua Schmidt, and Michael Leuschel. Analysing ProB's Constraint Solving Backends. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*. Springer, 2020. doi: 10.1007/978-3-030-48077-6_8

Part II.

New Applications for Constraint Programming in B

4. Translating Alloy and Extensions to Classical B

Sebastian Krings, Michael Leuschel, Joshua Schmidt, David Schneider, Marc Frappier

Abstract In this article, we introduce a denotational translation of the specification language Alloy to classical B. Our translation closely follows the Alloy grammar. Each construct is translated into a semantically equivalent component of the B language. In addition to basic Alloy constructs, our approach supports integers, sequences and orderings. The translation is fully automated and our implementation can be used in ProB. We evaluate the usefulness by applying AtelierB and ProB to translated models, showing benefits for proof and constraint solving with integers and higher-order quantification.

4.1. Introduction

Both B [25] and Alloy [40] are specification languages based on first-order logic. The languages share several features, such as native support for integers, sets and relations as well as user-defined types. However, there are also considerable differences. For instance, one of B's key concepts is to encode state changes by means of transitions, effectively computing successor states featuring all variables. In contrast, Alloy allows defining orderings over certain types.

Another difference between Alloy and B is tool support, especially when it comes to available backends for constraint solving. For Alloy, the Alloy Analyzer [40] is used to compute models by translating Alloy predicates to SAT using Kodkod [45]. The most prominent constraint solver for B, ProB [27–29], however mainly relies on constraint logic programming [104]. In particular, it uses the CLP(FD) library of SICStus Prolog [30] and extends it to support constraints over infinite domains [150]. Additionally, ProB allows using other backends, such as SMT solvers [48] or, again, Kodkod [46].

The different constraint solving techniques show different performance characteristics [151]. Certain predicates can be solved faster by using a particular backend or combination of backends; others cannot be handled by a particular solving technique at all. We thus suppose that a translation from Alloy models to B models serves different purposes:

– It provides Alloy users access to a set of new backends, and might enable constraint solving for Alloy models that cannot be handled efficiently by the Alloy Analyzer,

– it enables the application of the AtelierB provers [121] to Alloy models,

– it enables the usage of ProB as a second toolchain to validate the results of the Alloy Analyzer,

– it provides new test cases and benchmarks to the B community and should aid in improving ProB,

– it helps communication between the Alloy and B communities.

Our translation is integrated into ProB and is available at:

`https://www3.hhu.de/stups/prob`

Details about installing and using our translation can be found at:

`https://github.com/hhu-stups/alloy2b-doc`

This article is the extended version of our original ABZ submission [44]. For this article we extend our former work [44] in different aspects:

– The informal, more intuitive description of the translation from Alloy to B has been replaced by a formal description.

– We revised the translation of operations on orderings.

– The translation supports more Alloy constructs than the initial one. In particular, we added support for sequence operations, additional constraints on relations defined in `util/relation` and completed the translation of the join operator as well as all multiplicity annotations.

– We describe the tooling used in our translator in greater detail.

– Several special and edge cases are discussed more thoroughly.

– The empirical evaluation has been extended with more examples.

4.2. Introduction to Alloy and B

In the following, we give brief introductions to Alloy and B, discussing their approach to modeling and their specific features. Afterward, we point out the main differences between both languages.

4.2.1. Primer on Alloy

The Alloy notation is based on first-order logic with n-ary finite relations as the only type of terms. Sets are represented as unary relations. Basic sets and relations are defined using signatures, a construct similar to classes in object-oriented programming languages, which supports inheritance.

An Alloy specification consists of a set of signatures, noted `sig`, which basically define sets and relations, and a set of constraints, noted `fact`, that are first-order formulae which condition the values of sets and relations. A model can also contain assertions, which should hold when the facts are satisfied. The declaration `sig X {r : Y}` declares a signature (unary relation) X and a binary relation r which is a subset of the Cartesian product $X \times Y$. Alloy supports the usual operations on relations, such as union, intersection, difference, join, transitive closure, domain and range restriction. Fields (relations) of a signature are accessed using a convenient object-like notation (*e.g.*, $x.r = y$, with $x \in X$, $y \in Y$, and "." denotes the relational join operator). Alloy provides a universal type, noted `univ`, which is the union of all signatures. `Int` is the only predefined type; it is represented by the interval $-2^{n-1}..2^{n-1}-1$, where n is the number of bits used to store `Int` values. UML-like cardinality constraints can be defined on relations. Functions and predicates can be declared.

Alloy specifications can be decomposed into modules. Predefined modules provide support for Boolean, sequences, directed graphs, and total orderings on signatures.

The Alloy tool provides an editor, a model finder/enumerator and a model viewer based on the dot package. Alloy uses SAT solvers to build models to verify the satisfiability of axioms (facts) defined in an Alloy specification and to find counterexamples for assertions which should follow from these axioms. Only finite models are explored; their size is determined by a scope specified for each signature. Alloy facts and signatures are translated into Boolean formulas using Kodkod [45].

4.2.2. Primer on B

The formal specification language B [25] is based on first-order-logic and set theory and follows the correct-by-construction approach. B has initially been developed for the specification and design of software systems. Specific properties can be proven mathematically using theorem provers, *e.g.*, using AtelierB [121], or be checked using a model checker such as ProB [27–29].

A formal model in B consists of a collection of machines, containing a high-level abstract specification which is successively refined and decomposed. The development in B is thus incremental, which increases the maintainability and eases the specification of complex models.

A machine consists of variable and type definitions as well as initial values. A state is defined by the current values of the machine variables. By defining machine operations, one is able to specify transitions between states, effectively computing successor states featuring all variables. A machine operation has a unique name and consists of a B substitution (aka statement) defining the machine state after its execution. An operation can have a precondition, allowing or prohibiting execution based on the current state. For instance, a valid machine operation `o` is defined by `o = PRE x>0 THEN x:=x+1 END` using the single assignment substitution of B. Several variables can be assigned either in parallel or in sequence.

To ensure a certain behavior, the user can define machine invariants, *i.e.*, safety properties that have to hold in every reachable state. Hence, the correctness of a formal model refers to the specified invariants. ProB further supports linear temporal logic and fairness constraints, which enables the specification and verification of liveness properties.

In addition to the types explicitly provided by the B language such as `INTEGER` or `BOOL`, one can provide user-defined sets. These sets can be defined by a finite enumeration of distinct elements (the set is then referred to as an enumerated set) or left open (called deferred sets). For instance, by defining a set $S = \{s\}$ the element s is of type S and can be accessed by name within the machine. Deferred sets are assumed to be non-empty during proof and also finite for animation.

B is statically and strongly typed while ProB further executes runtime checks to ensure well-definedness. Type domains can be unbounded, possibly resulting in a model with an infinite state space. Further, B and ProB have support for higher-order quantification, in particular, sets and relations can be nested arbitrarily.

Code generators can be used to derive executable code from B models, targeting traditional programming languages such as C, C++ or Java.

4.2.3. Comparing Alloy and B

Although Alloy and B share common features, both languages have considerable differences. Most notably, Alloy and B have a different understanding of states. In B, state changes are encoded by means of operation executions, leading to successor states featuring all variables. At its heart, Alloy only has a single constant state, there is no concept of an operation. Alloy, however, allows defining orderings, allowing one to reason about sequences of states. In contrast to B, a predicate can then access any state in the sequence, not just the current state and its immediate successors.

Further, B is strongly typed, while Alloy only enforces the arities of an operation's arguments to match and provides the universal type, *i.e.*, the union of all signatures. On the one hand, strong typing is less error-prone than weak typing and enables a wide range of code analysis techniques to be applied. For instance, ProB throws a well-definedness error if a sequence operation is called on an improper sequence, whereas this is permitted in Alloy (but might not always be desired). On the other hand, strong typing possibly hinders a concise and idiomatic specification of software systems, especially in the context of object-oriented programming languages.

In B, tuples are encoded as nested pairs. Thus, several encodings of tuples exist and the modeler has to know which one is being used. For example, a triple can be represented as either $(x \mapsto (y \mapsto z))$ or $((x \mapsto y) \mapsto z)$. In Alloy, tuples are flat. This makes the join operator of Alloy powerful and enables expressing certain constructs more concisely than is possible in B.

Listing 4.1: Signatures of the Own Grandpa model in Alloy.

```
module SelfGrandpas
abstract sig Person {
    father : lone Man,
    mother : lone Woman
}
sig Man   extends Person {   wife : lone Woman    }
sig Woman extends Person {   husband : lone Man   }
```

Alloy follows the small scope hypothesis, which states that most bugs can be found by testing a program within a small scope [152], and bounds every domain. Hence, the Alloy Analyzer is not able to generally prove properties for a model but show the absence of counterexamples within a restricted scope. In contrast, domains can be unbounded in B. Besides using AtelierB's theorem provers, symbolic model checking techniques of ProB can be used to verify properties on infinite state spaces [153].

B and ProB have support for higher-order quantification, in particular, sets and relations can be nested arbitrarily. Higher-order specifications are also expressible in Alloy but cannot be handled by the Alloy Analyzer (an error is thrown). Alloy* [154] is an extension of Alloy which is able to do so.

4.3. Translation Example

In the following section, we introduce our translation on a simple Alloy model taken from Chapter 4 in "Software Abstractions: Logic, Language, and Analysis" [40]. The model is given in Listing 4.1 and Listing 4.3, the translation is given in Listing 4.2 and Listing 4.4. Our translation will only use the following concepts of a B machine:

1. Deferred sets, introducing new types for Alloy signatures in the `SETS` clause.
2. Constants, introduced in the `CONSTANTS` clause.
3. Predicates about the constants and deferred sets defined in the `PROPERTIES` clause. This includes typing predicates for the constants.
4. `DEFINITIONS`, aka B macros, to ease translating certain Alloy concepts.
5. B operations for Alloy assertion checks.

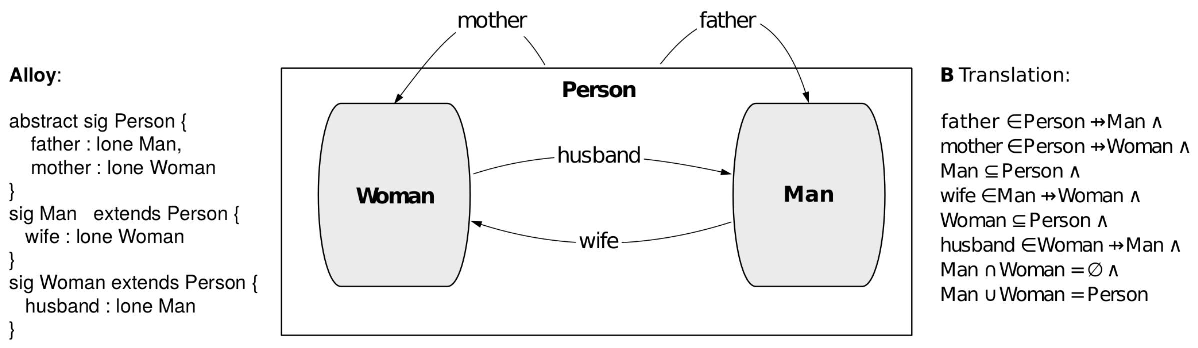
In particular, our translation does not use variables, invariants or assertions.

4.3.1. Translating Signatures

We first concentrate on the translation of Alloy's signatures and fields in Listing 4.1 to B types. An overview of the signatures and fields can be found in Figure 4.1.

Listing 4.2: Signatures of the Own Grandpa model translated to B.

```
 1  MACHINE SelfGrandpas
 2  SETS
 3      Person
 4  CONSTANTS
 5      Man, Woman, father, mother, wife, husband
 6  PROPERTIES
 7      father ∈ Person ⇸ Man ∧
 8      mother ∈ Person ⇸ Woman ∧
 9      Man ⊆ Person ∧
10      wife ∈ Man ⇸ Woman ∧
11      Woman ⊆ Person ∧
12      husband ∈ Woman ⇸ Man ∧
13      Man ∩ Woman = ∅ ∧
14      Man ∪ Woman = Person
15  END
```



Alloy:

```
abstract sig Person {
   father : lone Man,
   mother : lone Woman
}
sig Man   extends Person {
  wife : lone Woman
}
sig Woman extends Person {
  husband : lone Man
}
```

B Translation:

```
father ∈Person ⇸Man ∧
mother ∈Person ⇸Woman ∧
Man ⊆Person ∧
wife ∈Man ⇸Woman ∧
Woman ⊆Person ∧
husband ∈Woman ⇸Man ∧
Man ∩Woman =∅ ∧
Man ∪Woman =Person
```

Figure 4.1.: Signatures and fields of the Own Grandpa model in Alloy and B.

In order to translate the Alloy module `SelfGrandpas`, we create a B machine with the same name. Inside, the basic signature `Person`, defined in line 2 of the Alloy model, is represented as a user-given set in line 3 of the B machine in Listing 4.2. For the sake of readability, the example translation uses the same identifiers as the Alloy module. Of course, one has to ensure the translation is valid, *e.g.*, identifiers do not collide with B's keywords. Deferred sets in B can have any size, just like signatures in Alloy. In Section 4.4.14 we show how a limit on the size of the signature is translated.

The signature features two fields, `father` and `mother`, each representing a relation of members of `Person` to members of `Man` and `Woman`. The keyword `lone` states that the relation is in fact a partial function, *i.e.*, a 1-to-at-most-1 mapping. This can be encoded into B using a partial function, as created by the ⇸ operator in lines 7 and 8 of Listing 4.2.

The extending signatures `Man` and `Woman` are subsets of `Person`. As user-given sets in B are distinct, we introduce constants `Man` and `Woman` and assert the subset property in

Listing 4.3: Facts and predicates of the Own Grandpa model in Alloy.

```
...
fact Terminology {   wife = ~husband   }
fact SocialConvention {
    no wife & *(mother + father).mother
    no husband & *(mother + father).father
}
fact Biology {
    no p : Person | p in p.^(mother + father)
}
fun grandpas[p : Person] : set Person {
    let parent = mother + father + father.wife + mother.husband
    | p.parent.parent & Man
}
pred ownGrandpa[m : Man] {
    m in grandpas[m]
}
run ownGrandpa for 4 Person
```

lines 9 and 11 of Listing 4.2. As above, the fields `wife` and `husband` are translated into partial functions in lines 10 and 12.

Since `Person` was declared `abstract`, two additional properties have to hold for the sub-signatures: each element of `Person` has to be in one of the sub-signatures and the two sub-signatures have to be disjoint. This partitioning of `Person` is encoded in B's set theory in lines 13 and 14 of Listing 4.2.

4.3.2. Translating Facts and Predicates

Alloy facts are added to the B machine's `PROPERTIES` clause. For example, the Alloy fact `Terminology` of Listing 4.3, stating that `wife` is the inverse of `husband`, can be encoded in B using the relational inverse, see line 11 of Listing 4.4.

The first fact in `SocialConvention` states that your wife cannot be your mother or the mother of your ancestors. The second fact asserts the same property for husband and father. Both can be translated directly as far as set union, intersection, and closure computation are concerned. The dot join in this case is interpreted as the relational composition of the two relations, which is available in B by using the `;` operator. Other interpretations of the dot join operator are discussed later. The keyword `no` enforcing the emptiness of a set is translated to equalities to the empty set in lines 12 and 13.

The Alloy fact `Biology`, stating that nobody can be their own ancestor, introduces a quantified local variable `p`. Again, `no` enforces emptiness of the set. We translate the fact into a negated existential quantification, which is able to introduce the variable. Observe, that quantification in Alloy is over singleton sets only. More generally, we translate the quantification `no` $p\ :\ S\ \mid\ P$ into $\neg\exists p.(\{p\} \subseteq S \wedge P)$.

The function definition `grandpas` and the predicate definition `ownGrandpa`, both with a parameter, are encoded as B definitions, permitting their reuse throughout the model.

Listing 4.4: Facts and predicates of the Own Grandpa model translated to B.

```
MACHINE SelfGrandpas
...
DEFINITIONS
    parent == mother ∪ father ∪
              (father ; wife) ∪ (mother ; husband);
    ownGrandpa(m) == {m} ⊆ Man ∧ ({m} ⊆ grandpas(m));
    grandpas(p) == {tmp | {p} ⊆ Person ∧
                          tmp ∈ (parent[parent[{p}]] ∩ Man)}
PROPERTIES
    ...
    wife = husband⁻¹ ∧
    wife ∩ (closure((mother ∪ father)) ; mother) = ∅ ∧
    husband ∩ (closure((mother ∪ father)) ; father) = ∅ ∧
    not(∃p.({p} ⊆ Person ∧
         {p} ⊆ closure1((mother ∪ father))[{p}])) ∧
    card(Person) ≤ 4
OPERATIONS
    run_ownGrandpa = PRE ∃m.(ownGrandpa(m)) THEN skip END
END
```

The predicate definition `ownGrandpa` only includes the application of `grandpas` as well as a membership check and can thus be translated directly.

Translating `grandpas` however is not straightforward, as it includes a let expression, which is not available in B.[1] As an alternative to inlining, we again create a definition named `parent` in order to hold the value of the newly introduced variable. Note that this changes the scope in which the variable resides and might make renaming necessary in order to avoid conflicts. Furthermore, observe that there are no free variables in the definition of `parent`. Otherwise, those would be passed to the B definition as parameters. As `grandpas` returns a set of `Person`, the definition again uses a set comprehension.

4.4. Formal Description of the Translation

The original paper by Daniel Jackson [44] (notably Figure 2) as well as Appendix C in „Software Abstractions: Logic, Language and Analysis" [40] provide a semantics of the kernel of Alloy in terms of logical and set-theoretic operators. It introduces a function M to give the meaning of formulas (aka predicates) and a function E that gives the meaning of expressions. One of the rules defined by Jackson [40] gives the meaning of the + operator as the set-theoretic union of the meaning of its arguments:

$$E[\![p + q]\!]i \;\widehat{=}\; E[\![p]\!]i \cup E[\![q]\!]i$$

The argument i is an environment where some identifiers can be given a value. The environment is used to deal with quantifiers and identifiers: quantifiers update the en-

[1] Let expressions are available in an extended version of B understood by ProB.

vironment, while the function is applied when computing the meaning of an identifier r as follows: $E[\![r]\!]i = i(r)$.

Our translation rules are an alternate specification of this semantics, using the B operators and also using B quantification. Our translation rules are more comprehensive and sometimes more involved due to the following reasons:

– The B language has a more restrictive syntax concerning set comprehensions and always requires explicit quantification of all introduced identifiers, in contrast to the "flexible" mathematical notation employed by Jackson [39, 40].

– In Alloy tuples are flat and Cartesian product is associative; in B Cartesian product is not associative and tuples are represented as nested pairs.

– We have to provide the full translation for all operators. (Jackson [39, 40] presents the kernel semantics and only a few translation rules for the full language to the kernel language.)

– The translation by Jackson [39, 40] does not specify all aspects of encoding signatures, which we have to deal with in an automated translation.

4.4.1. Overview of the Semantic Functions

We provide four semantic functions, one for expressions, one for predicates, and two for declarations that introduce new quantified variables.

Each semantic function has an argument i which is an environment storing different information to be used during translation. In particular, the information is used for optimization as it allows using more specialized encodings in certain situation. For instance, the environment stores identifiers that are singleton sets. Thus, if $x \in i$, then x is translated as $\{x\}$, whereas identifiers not in i are translated as x. This information is relevant to obtain a more effective encoding into B, using scalar values instead of set values whenever possible. Note that, as we translate Alloy quantification to B quantification, there is no need to store values for the quantified variables as in the functions defined by Jackson [39, 40]. However, we also store information about identifiers which are known to be total functions in i. We are then able to translate specific Alloy constructs in a more idiomatic and, in terms of solving constraints, more efficient way. For instance, when using total functions, we can translate element access using B's function application instead of using the relational image operator.

For the sake of readability and brevity, our translation rules only detail how identifiers representing singleton sets are tracked. Identifiers for total functions are tracked similarly.

In particular, we provide the following semantic functions:

1. $E[\![A]\!]i$ is the B encoding of the Alloy expression A given the environment i.

2. $M[\![A]\!]i$ is the B encoding of the Alloy predicate A given the environment i.

3. $D[\![A]\!]i$ is the B encoding of the Alloy quantification declaration A given the environment i. $I[\![A]\!]i$ is the updated set of identifiers after processing the declaration A.

4. $F[\![A]\!]i$ is the B encoding of the Alloy signature field declaration A given the environment i.

4.4.2. Example

Before presenting the rules in detail, we process a small sub-predicate from Section 4.3.2:

`no` wife & *(mother + father).mother

To translate this Alloy predicate to B we need the following semantic rules:

$M[\![\texttt{no}\ p]\!]i \ \widehat{=}\ E[\![p]\!]i = \varnothing$

$E[\![p\ \texttt{\&}\ q]\!]i \ \widehat{=}\ E[\![p]\!]i \cap E[\![q]\!]i$

$E[\![p\ \texttt{+}\ q]\!]i \ \widehat{=}\ E[\![p]\!]i \cup E[\![q]\!]i$

$E[\![x]\!]i \ \widehat{=}\ x$ for identifiers not occurring in i (*e.g.*, signature names and fields)

$E[\![\texttt{*}p]\!]i \ \widehat{=}\ \text{closure}(E[\![p]\!]i)$

$E[\![p.q]\!]i \ \widehat{=}\ (E[\![p]\!]i; E[\![q]\!]i)$ if both p and q are binary relations

Note that closure is B's transitive and reflexive closure operator on relations, while ";" is the relational composition operator.

Here is a step-by-step application of these rules to obtain the B translation, where $i = \varnothing$.

1. $M[\![\texttt{no wife \& *(mother + father).mother}]\!]i$

2. $E[\![\texttt{wife \& *(mother + father).mother}]\!]i = \varnothing$

3. $E[\![\texttt{wife}]\!]i \cap E[\![\texttt{*(mother + father).mother}]\!]i = \varnothing$

4. wife $\cap$ $(E[\![\texttt{*(mother + father)}]\!]i\ ;\ E[\![\texttt{mother}]\!]i) = \varnothing$

5. wife $\cap$ (closure($E[\![\texttt{(mother + father)}]\!]i$) ; mother) $= \varnothing$

6. wife $\cap$ (closure($E[\![\texttt{mother}]\!]i \cup E[\![\texttt{father}]\!]i$) ; mother) $= \varnothing$

7. wife $\cap$ (closure(mother $\cup$ father) ; mother) $= \varnothing$

4.4.3. Signature and Field Declarations

Since a signature declaration can be quite complex, let us start with the simplest one, omitting everything optional, *i.e.*, we only add a named signature to the model. A signature has the properties of a set, containing atoms of the signature's type. For the translation to B, we create a new deferred set for each signature. In B, a deferred set introduces a user-defined type as a finite and non-empty set.

Additionally, a signature can extend another signature by making use of either the `in` or the `extends` keyword. In this case, we set up a subset of an already existing set, *i.e.*, for each sub-signature S extending base signature S_b we define a constant S and add $S \subseteq S_b$ to the `PROPERTIES` clause.

For the `extends` keyword, we ensure that extending signatures are pairwise disjoint by adding $S_1 \cap S_2 = \varnothing$ for each combination of extending signatures S_1, S_2, with $S_1 \neq S_2$, to the B machine's `PROPERTIES` clause.

Next, base signatures can be declared as `abstract`: Abstract signatures are used for the sole purpose of being extended by other signatures. They do not contain elements which are not also elements of other sets [40]. In B, this property can be modeled by adding the following constraint to the `PROPERTIES` section:

$$S_b = \bigcup_{S \text{ extends } S_b} S.$$

Alloy allows stating the cardinality of signatures using multiplicities. The quantifiers `lone` (at most one) and `some` (at least one) are translated straightforwardly using cardinality constraints in the B machine's `PROPERTIES` section. In case of `one` (exactly one), we define a signature as a singleton enumerated set in B instead of a deferred set.

An Alloy signature may contain a list of fields, *i.e.*, relations defined over the signature's elements. Since B natively supports relations, the translation is straightforward: for a signature S with fields $f_{S,j}$, each mapping an element of S to S_j, we add a constant $f_{S,j}$ and state that $f_{S,j}$ is a relation between S and S_j by the B constraint $f_{S,j} \in S \leftrightarrow S_j$. Note that in B relations and functions are sets of tuples, which is why we define a membership constraint for $f_{S,j}$.

We have to ensure that the names of the constants $f_{S,j}$ are unique in terms of the current model since fields of different signatures can have the same name in Alloy. Otherwise, the resulting B machine might not be well-defined because of clashes between constants with the same name but different types.

It is also possible to make use of quantifiers when declaring field variables: In this way we can decide on the number of elements that are mapped to. The default quantification for relations in Alloy is a mapping (Alloy quantifier `one`) while in B it is an 1-to-n mapping (Alloy quantifier `set`). Therefore, if no quantifier is given in the Alloy model, the translation to B has to be adapted, *i.e.*, we add the constraint $f_{S,j} \in S \rightarrow S_j$, stating that $f_{S,j}$ is a total function. The translation of the quantifier `lone` results in a partial function. In case of `set`, no additional property is needed, since it is the default of B. For the multiplicity `some`, we add the additional constraint $\text{dom}(f_{S,j}) = S$, *i.e.*, the field $f_{S,j}$ is a total relation.

Furthermore, a signature's field can be defined as a sequence of atoms. We translate a sequence q as a partial function with a finite and coherent domain $0..\operatorname{card}(q)-1$ and ensure that the maximum allowed length of sequences $m \in \mathbb{N}$ is not exceeded. As a signature can have several instances, the complete domain of a signature field that is a sequence is defined as the Cartesian product of the signature and the partial function. We explain the translation of Alloy sequences to B more precisely in Section 4.5.5.

Taken together, the formal translation of field quantifications for an exemplary signature `S` to B machine properties is as follows:

$$F[\![\mathit{field} : \texttt{set}\ \mathit{Set}]\!]i \ \widehat{=}\ \mathit{field} \in E[\![S]\!]i \ \leftrightarrow\ E[\![\mathit{Set}]\!]i$$

$$F[\![\mathit{field} : \texttt{one}\ \mathit{Set}]\!]i \ \widehat{=}\ \mathit{field} \in E[\![S]\!]i \ \rightarrow\ E[\![\mathit{Set}]\!]i$$

$$F[\![\mathit{field} : \texttt{lone}\ \mathit{Set}]\!]i \ \widehat{=}\ \mathit{field} \in E[\![S]\!]i \ \mapsto\!\!\!\!\!\!\!+\ E[\![\mathit{Set}]\!]i$$

$$F[\![\mathit{field} : \texttt{some}\ \mathit{Set}]\!]i \ \widehat{=}\ \mathit{field} \in E[\![S]\!]i \ \leftrightarrow\ E[\![\mathit{Set}]\!]i \wedge \operatorname{dom}(\mathit{field}) = E[\![S]\!]i$$

$$F[\![\mathit{field} : \texttt{seq}\ \mathit{Set}]\!]i \ \widehat{=}\ \mathit{field} \in (E[\![S]\!]i \ \times\ 0..(m-1)) \ \mapsto\!\!\!\!\!\!\!+\ E[\![\mathit{Set}]\!]i \ \wedge$$
$$\forall s.(s \in E[\![S]\!]i \Rightarrow \operatorname{dom}(\mathit{field})[\{s\}] = 0..\operatorname{card}(\operatorname{dom}(\mathit{field})[\{s\}])-1)$$

$$F[\![\mathit{field} :\ \mathit{Set}]\!]i \ \widehat{=}\ F[\![\mathit{field} : \texttt{one}\ \mathit{Set}]\!]i$$

Besides constraining the quantification of signature fields, one can use the keyword `disj` in combination with any multiplicity in order to define that distinct members of a signature yield distinct field values. For instance, the signature `sig` S $\{f$: `disj` $e\}$ defines the signature field f to be disjoint for distinct members of S. This results in the additional constraint `all` a, b : S | a ! = b `implies no` $a.f$ & $b.f$ [40]. We straightforwardly translate the universal quantification and add it to the translated B machine's `PROPERTIES` section.

Alloy allows providing additional constraints on signature elements together with the signature definition. However, aside of syntactical sugar, they do not differ from regular constraints stated via `fact` declarations and are thus not considered further in this section.

4.4.4. Universe and Identity

Alloy features two special constants: `univ`, referring to the set of all instances of all signatures and `iden`, the identity relation over the universe. Neither is available in B. To translate `univ`, we could create a top-level set `UNIVERSE` and ensure that all base signatures implicitly extend it. This negatively impacts ProB's solving capabilities: without distinct sets for different signatures, techniques such as symmetry reduction cannot be applied as efficiently. ProB's kernel becomes unable to reason on types and thus has to perform more involved case distinctions. Further, in case integers are used in an Alloy model, we would need to create a singleton set for each integer extending the set `UNIVERSE` although B and ProB have native support for real integers. In consequence, we only create parent types for specific signatures if necessary.

For instance, if two signatures S_1 and S_2 have no parent type except for `univ` and interact with each other using a union $S_1 + S_2$, we introduce an additional deferred set in the translated B machine and declare membership for both signatures. The two signatures are then defined as machine constants rather than deferred sets in B. For above example, this results to $S_1 \cap S_2 = \varnothing \wedge S_1 \cup S_2 = P$ being added to the B machine properties where P is the introduced parent type, *i.e.*, a deferred set in B.

To do so, we analyze an Alloy model prior to the translation and collect pairs of signatures that interact with each other and have no parent type except for `univ`. All collected pairs are merged to distinct sets of signature types where in each set at least two signatures interact with each other. When translating an Alloy model, we define a parent type for each set of signatures. In case all signatures of a model interact with each other, our translation introduces one parent type for all signatures. Despite containing integers, this type is equal to Alloy's universal type given a specific model.

As we use the integer type of B to translate Alloy integers, which cannot be a subset of a deferred set, we cannot translate interactions between signatures and integers straightforwardly. To do so, we would need to create a singleton set for each integer extending the set `UNIVERSE` in B. To our knowledge, a binary interaction between a signature type and integers is not needed in any reasonable Alloy model. However, it is allowed by the Alloy Analyzer's typechecker.

For binary operations, the translation of `univ` can be avoided in several typical use cases, *e.g.*, left and right joins with the universe can be translated into domain and range computation.

Using `UNIVERSE`, we could translate `iden` to id(`UNIVERSE`). However, as we want to avoid the universal type as much as possible, we again chose a more specialized translation. That is, instead of translating into the identity over the universe, we rely on the Alloy Analyzer's typechecking information and translate into a more restricted identity relation if possible without changing the semantics. For instance, the Alloy expression `iden` & r, where T is a signature and $r \in T \leftrightarrow T$, can be translated as $\mathrm{id}(T) \cap r$.

However, we do not allow the keywords `univ` or `iden` for specific operators which we cannot translate to an equivalent B expression without introducing the universal type in B. We instead throw an error and do not translate the Alloy model to B. In particular, these operators are equality, inequality and the subset relation `in`. For instance, consider an Alloy model which defines a signature `sig` T { r : `set` T } . We translate the signature as described in Section 4.4.3, *i.e.*, we introduce a deferred set in B, define the field as a machine constant, and add $E[\![r]\!]i \in E[\![T]\!]i \leftrightarrow E[\![T]\!]i$ to the machine properties. The Alloy model further defines a basic signature R without any field as well as a global fact $T.r = T$, *i.e.*, the signature field r contains all possible elements. When checking the assertion `iden` *in* $(\tilde{}r).r$, the Alloy Analyzer finds a counterexample as the identity relation of the signature R is not part of the dot join's result. If following our restricted translation of the keyword `iden` to B, we would translate the assertion into a more restricted expression $\mathrm{id}(E[\![T]\!]i) \subseteq ((E[\![r]\!]i)^{-1}; E[\![r]\!]i)$ which does not provide a counterexample in B, *i.e.*, the translation would be semantically non-equivalent. An equivalent behavior in B could be achieved by introducing the universal type and using id(`UNIVERSE`) rather than $\mathrm{id}(E[\![T]\!]i)$. Yet, we decided to not support the translation of said expressions containing one of the keywords `univ` and `iden`.

4.4.5. Connectives and Simple Predicates

Let us first look at how to translate Alloy's logical connectives. This part is very straightforward, as they have matching counterparts in B.

$$M[\![p \texttt{ and } q]\!]i \;\widehat{=}\; M[\![p]\!]i \wedge M[\![q]\!]i$$

$$M[\![p \texttt{ or } q]\!]i \;\widehat{=}\; M[\![p]\!]i \vee M[\![q]\!]i$$

$$M[\![p \texttt{ implies } q]\!]i \;\widehat{=}\; M[\![p]\!]i \Rightarrow M[\![q]\!]i$$

$$M[\![p \texttt{ equiv } q]\!]i \;\widehat{=}\; M[\![p]\!]i \Leftrightarrow M[\![q]\!]i$$

$$M[\![\texttt{not } p]\!]i \;\widehat{=}\; \neg M[\![p]\!]i$$

Similarly, equality and inequality in Alloy and B are identical:

$$M[\![p \texttt{ = } q]\!]i \;\widehat{=}\; E[\![p]\!]i = E[\![q]\!]i$$

$$M[\![p \texttt{ != } q]\!]i \;\widehat{=}\; E[\![p]\!]i \neq E[\![q]\!]i$$

The following unary expressions can be used to constrain a set's cardinality:

$$M[\![\texttt{no } p]\!]i \;\widehat{=}\; E[\![p]\!]i = \varnothing$$

$$M[\![\texttt{one } p]\!]i \;\widehat{=}\; \mathrm{card}(E[\![p]\!]i) = 1$$

$$M[\![\texttt{some } p]\!]i \;\widehat{=}\; \mathrm{card}(E[\![p]\!]i) > 0$$

$$M[\![\texttt{lone } p]\!]i \;\widehat{=}\; \mathrm{card}(E[\![p]\!]i) \leq 1$$

Now, to translate the `in` predicate, it is important to understand that Alloy only operates on set values. This means, that it is translated using the B $\subseteq$ predicate, and not the $\in$ predicate.

$$M[\![p \texttt{ in } q]\!]i \;\widehat{=}\; E[\![p]\!]i \subseteq E[\![q]\!]i$$

4.4.6. Simple Expressions

First, we need to translate simple identifiers not occurring in the environment i, *e.g.*, signature names and fields. In general, identifiers are simply kept as they are. In the presence of modules and namespaces, identifiers have the module names prefixed to avoid ambiguity. This is already handled by the Alloy Analyzer's parser and typechecker we use as a frontend as is outlined in Section 4.7 where we give technical details on the implementation.

$$E[\![x]\!]i \;\widehat{=}\; x \text{ for identifiers } x \text{ not occurring in } i$$

In the following, we present simple translation rules, where one Alloy operator gets translated to a B operator or constant:

$$E[\![\texttt{none}]\!]i \;\widehat{=}\; \varnothing$$

$$E[\![p \texttt{ + } q]\!]i \;\widehat{=}\; E[\![p]\!]i \cup E[\![q]\!]i$$

$$E[\![p \texttt{ \& } q]\!]i \;\widehat{=}\; E[\![p]\!]i \cap E[\![q]\!]i$$

$$E[\![p \texttt{ - } q]\!]i \;\widehat{=}\; E[\![p]\!]i \setminus E[\![q]\!]i$$

$$E[\![\tilde{}p]\!]i \;\widehat{=}\; (E[\![p]\!]i)^{-1}$$

$$E[\![\hat{}p]\!]i \;\widehat{=}\; \text{closure1}(E[\![p]\!]i)$$

$$E[\![{*}p]\!]i \;\widehat{=}\; \text{closure}(E[\![p]\!]i)$$

$$E[\![p \mathbin{++} q]\!]i \;\widehat{=}\; E[\![p]\!]i \mathbin{<\!\!+} E[\![q]\!]i$$

Note that the Alloy operators ~, ^ and * are only allowed for binary relations. Hence, we can translate them to the B counterparts. However, other Alloy operators also work for relations of higher arity. As such we can encounter not just ordered pairs but tuples, whose translation we discuss in the next subsection.

As discussed in the introductory example in Section 4.3, classical B does not feature a let expression. This can either be resolved by using a definition as done in the example, inlining, or by using the extended version of B understood by ProB. In our current translation we use the let expression provided by ProB. Of course, a model can then not be processed by other tools such as AtelierB anymore. To do so, ProB provides a pretty-printer which rewrites B abstract syntax trees to native B as, for instance, understood by AtelierB.

4.4.7. Representing Tuples

In Alloy tuples are flat and Cartesian product is associative. In B, Cartesian product is not associative and tuples are represented as nested pairs. Hence, in B $(e_1 \mapsto e_2) \mapsto e_3$ is a different value and has a different type than $e_1 \mapsto (e_2 \mapsto e_3)$. Which encoding should we use for an Alloy triple (e_1, e_2, e_3) within a ternary relation r? Both have their advantages and drawbacks, concerning the use of the B operators such as domain, range, relational image or function application. The B language also provides a comma notation for pairs, whose associativity corresponds to the first alternative:

$$(e_1, e_2, e_3) = (e_1 \mapsto e_2) \mapsto e_3$$

We have finally chosen to use the first alternative, as it allows us to write set comprehensions of the form $\{x_1, \dots, x_n \mid P\}$ to generate n-ary relations of the right type.

Another alternative would have been to allow all variations in our translation, keeping track in the type system which associativity has been generated. It is not clear whether this is worthwhile and it definitely makes the translation rules much more complex.

4.4.8. Cartesian Product

Due to the difference in the treatment of tuples, the Cartesian product in Alloy can also behave differently than in B. When the second argument is a unary relation, we can reuse the B Cartesian product, otherwise we need to compute it using a set comprehension.

For example, the following B Cartesian product between a ternary and a unary relation works correctly:

$$\{(1 \mapsto 2) \mapsto 3\} \times \{4\} = \{((1 \mapsto 2) \mapsto 3) \mapsto 4\}$$

However, for a ternary and binary relation it does not work, as the pairs in the result are incorrectly nested:

$$\{(1 \mapsto 2) \mapsto 3\} \times \{(4 \mapsto 5)\} = \{(((1 \mapsto 2) \mapsto 3) \mapsto (4 \mapsto 5))\}$$

Using a set comprehension we can compute the correct result:

$\{t, q1, q2 \mid t \in \{(1 \mapsto 2) \mapsto 3\} \wedge (q1, q2) \in \{(4 \mapsto 5)\}\}$ gives us a correctly encoded tuple: $\{(((1 \mapsto 2) \mapsto 3) \mapsto 4) \mapsto 5\}$

This leads to the two following rules:

$E[\![p\ \texttt{->}\ q]\!]i \ \widehat{=}\ E[\![p]\!]i \times E[\![q]\!]i$ if q is a unary relation

$E[\![p\ \texttt{->}\ q]\!]i \ \widehat{=}\ \{t, q_1, \dots, q_n \mid t \in E[\![p]\!]i \wedge (q_1, \dots, q_n) \in E[\![q]\!]i\}$ if q is an n-ary relation with $n > 1$.

4.4.9. Domain and Range Restriction

The domain restriction can be translated as follows. For binary relations q we can reuse the corresponding B operator $\lhd$, otherwise we need to compute the result using a set comprehension:

$E[\![p\ \texttt{<:}\ q]\!]i \ \widehat{=}\ E[\![p]\!]i \lhd E[\![q]\!]i$ if q is a binary relation

$E[\![p$ `<:` $q]\!]i \;\widehat{=}\; \{q_1, \ldots, q_n \mid q_1 \in E[\![p]\!]i \wedge (q_1, \ldots, q_n) \in E[\![q]\!]i\}$ if q is an n-ary relation with $n > 2$.

For range restriction the corresponding B operator $\rhd$ works for all arities, since q must be a unary relation in Alloy.

$E[\![p$ `:>` $q]\!]i \;\widehat{=}\; E[\![p]\!]i \rhd E[\![q]\!]i$

Yet, there are some special cases for domain and range restriction with `iden` or `univ`. We thus encode the restriction of the identity relation to the domain p in B as the binary relation that relates every element of p to itself, while a domain restriction on `univ` returns the domain itself. For the identity relation, the assertion `all` p : `univ` | p <: `iden` = `iden` :> p holds (analogous for `univ`). This shows that both expressions are equivalent, which we represent using the symbol $\equiv$, resulting in the following translation to B:

$E[\![p <: \texttt{iden}]\!]i \equiv E[\![\texttt{iden} :> p]\!]i \;\widehat{=}\; \lambda x.(x \in E[\![p]\!]i \mid x)$

$E[\![p <: \texttt{univ}]\!]i \equiv E[\![\texttt{univ} :> p]\!]i \;\widehat{=}\; E[\![p]\!]i$

4.4.10. Join

The dot join $p.q$, one of the most important operators in Alloy, is also the most difficult to translate. We have quite a lot of special cases below, trying to use existing B operators if possible. First are three special cases, where one of the arguments is the Alloy universe. These operations correspond to computing the domain and range of the Alloy relations:

$E[\![p.\texttt{univ}]\!]i \;\widehat{=}\; \mathrm{dom}(E[\![p]\!]i)$ where p is an n-ary relation, $n \geq 2$

$E[\![\texttt{univ}.q]\!]i \;\widehat{=}\; \mathrm{ran}(E[\![q]\!]i)$ if q is a binary relation

$E[\![\texttt{univ}.q]\!]i \;\widehat{=}\; \{q_2, \ldots, q_k \mid \exists j.((j, q_2, \ldots, q_k) \in E[\![q]\!]i)\}$ if q is an n-ary relation, $n > 2$

Another typical pattern in Alloy is to use the join operator for relational image or function application. The next three translation rules capture these patterns:

$E[\![p.q]\!]i \;\widehat{=}\; \{E[\![q]\!]i\ (pv)\}$ if $E[\![p]\!]i = \{pv\}$, p is a unary relation, q is an n-ary relation with n ≥ 2, and q is known to be a total function in pv.

$E[\![p.q]\!]i \;\widehat{=}\; E[\![q]\!]i\ [\ E[\![p]\!]i\ ]$ if p is a unary relation, q is a binary relation

$E[\![p.q]\!]i \;\widehat{=}\; (E[\![p]\!]i)^{-1}[E[\![q]\!]i]$ if q is a unary relation, p is a binary relation

The translation rules above are optional, since the next three translation rules of the join operator also cover the same cases, but lead to a less idiomatic B translation. The first one uses B's relational composition operator:

$E[\![p.q]\!]i \;\widehat{=}\; (E[\![p]\!]i\ ;\ E[\![q]\!]i)$ if both p and q are binary relations

$E[\![p.q]\!]i \;\widehat{=}\; \{q_2, \ldots, q_k \mid \exists j.(j \in E[\![p]\!]i \wedge (j, q_2, \ldots, q_k) \in E[\![q]\!]i\}$ if p is unary relation

$E[\![p.q]\!]i \;\widehat{=}\; \{t, q_2, \ldots, q_k \mid \exists j.((t, j) \in E[\![p]\!]i \wedge (j, q_2, \ldots, q_k) \in E[\![q]\!]i\}$ in all other cases (*i.e.*, p is an n-ary relation with $n > 1$ and q is a k-ary relation with $k > 1$).

One can observe that the join operator of Alloy is very elegant and flexible; together with flat tuples, it provides an expressive construct. One could think about extending the B relational composition operator to work more flexibly (*i.e.*, also with sets and n-ary relations). This would also make the translation to B easier.

Alloy further provides the box join operator $y[x]$ which is just syntactic sugar for the dot join operation $x.y$ though.

4.4.11. Quantifications, Set Comprehensions and Identifiers

Quantifications in Alloy can introduce a finite set of identifiers using the : operator for unary sets S:

$D[\![x : \texttt{one}\ S]\!]i \;\widehat{=}\; \{x\} \subseteq E[\![S]\!]i$

$D[\![x : \texttt{set}\ S]\!]i \;\widehat{=}\; x \subseteq E[\![S]\!]i$

$D[\![x : \texttt{some}\ S]\!]i \;\widehat{=}\; x \subseteq E[\![S]\!]i \wedge x \neq \varnothing$

$D[\![x : \texttt{lone}\ S]\!]i \;\widehat{=}\; x \subseteq E[\![S]\!]i \wedge \mathrm{card}(x) \leq 1$

$D[\![x : S]\!]i \equiv D[\![x : \texttt{one}\ S]\!]i$ if the arity of S is 1

$D[\![x : S]\!]i \equiv M[\![x\ \texttt{in}\ S]\!]i$ if the arity of S is greater than 1

$D[\![x_1, \ldots, x_k : S]\!]i \equiv D[\![x_1 : S]\!]i \wedge \ldots \wedge D[\![x_k : S]\!]i$ for $k > 1$

$D[\![\texttt{disj}\ x_1, \ldots, x_k : S]\!]i \equiv D[\![x_1 : S]\!]i \wedge \ldots \wedge D[\![x_k : S]\!]i \wedge$
$\mathrm{card}(\{x_1, \ldots, x_k\}) = k$ for $k > 1$

Both Alloy and B feature set comprehensions, consisting of local identifiers and a constraining predicate. Translation is straightforward, as only the predicate has to be translated according to the rules given above. However, we have to ensure that unique names are used for the translation of local identifiers to avoid clashes between identifier names for nested scopes.

We apply a separate function to compute updates to the environment by identifier declaration, which is defined as follows:

$I[\![x : \texttt{one}\ S]\!]i \;\widehat{=}\; i \cup \{x\}$

$I[\![x : m\ S]\!]i \;\widehat{=}\; i - \{x\}$ for $m \neq \texttt{one}$.

$I[\![x : S]\!]i \;\widehat{=}\; i \cup \{x\}$ if arity of S is 1, $i - \{x\}$ otherwise

Given an environment i, we translate identifiers to B either as a singleton set or as a raw identifier:

$E[\![x]\!]i \;\widehat{=}\; \{x\}$ if $x \in i$ (*i.e.*, the environment i states that x is a singleton set identifier (*e.g.*, quantified variables with multiplicity one)

$E[\![x]\!]i \;\widehat{=}\; x$ for $x \notin i$ (*i.e.*, x is a signature name or field)

In the following, x are the left-hand side arguments of the declaration *Decl* and $i' = I[\![Decl]\!]i$ holds:

$M[\![\texttt{some}\ Decl \mid P]\!]i \;\widehat{=}\; \exists x.(D[\![Decl]\!]i \wedge M[\![P]\!]i')$

$M[\![\texttt{all}\ Decl \mid P]\!]i \;\widehat{=}\; \forall x.(D[\![Decl]\!]i \Rightarrow M[\![P]\!]i')$

$M[\![\texttt{no}\ Decl \mid P]\!]i \;\widehat{=}\; \neg\exists x.(D[\![Decl]\!]i \wedge M[\![P]\!]i')$

$M[\![\texttt{one}\ Decl \mid P]\!]i \;\widehat{=}\; \mathrm{card}(\{x \mid D[\![Decl]\!]i \wedge M[\![P]\!]i'\}) = 1$

$M[\![\texttt{lone}\ Decl \mid P]\!]i \;\widehat{=}\; \mathrm{card}(\{x \mid D[\![Decl]\!]i \wedge M[\![P]\!]i'\}) \leq 1$

4.4.12. Conditionals

Alloy provides a conditional statement, which can either be treated as a predicate or as an expression. We therefore provide the following two translation rules, one of which uses the if-then-else expression of ProB:

$M[\![p \Rightarrow q \text{ ELSE } r]\!]i \;\widehat{=}\; (M[\![p]\!]i \Rightarrow M[\![q]\!]i) \wedge (\neg M[\![p]\!]i \Rightarrow M[\![r]\!]i)$

$E[\![p \Rightarrow q \text{ ELSE } r]\!]i \;\widehat{=}\; \text{IF } M[\![p]\!]i \text{ THEN } E[\![q]\!]i \text{ ELSE } E[\![r]\!]i \text{ END}$

Again, ProB is able to rewrite if-then-else expressions to be compatible to native B as, *e.g.*, understood by AtelierB.

4.4.13. Fact, Function & Predicate Declaration

Alloy's `fact` declaration has an optional name and contains any number of predicates, which pose additional constraints to be added to the model. We translate the expressions as described above. The results are conjoined and added to the `PROPERTIES` section of the B machine.

Alloy allows declaring functions and predicates for later reuse. As usual, a function declaration takes a name, a (possibly empty) list of parameters and a body containing the actual computation. Parameters are scoped and can only be referred to by the function itself. Furthermore, they are typed as subsets of an Alloy signature and can again be quantified to constrain the set sizes.[2]

Functions are listed in the `DEFINITIONS` section of the B machine if the machine contains at least one invocation. Each function is translated into a single B definition with matching parameters, consisting of a set comprehension wrapping the actual expression in the body to account for the expected return type. For instance, the function declaration `fun` $f\ [s : S] : S\ \{\ \mathit{body}\ \}$ is translated into the B definition $f(s) == \{x \mid D[\![s : S]\!]i \wedge x \in E[\![\mathit{body}]\!]i\}$, where $i' = I[\![s : S]\!]i$ as in Section 4.4.11. Syntax and functionality of the predicate definition is slightly different. For the predicate to evaluate to true or false instead of computing a value, we omit the set comprehension resulting in a B predicate. For instance, the predicate declaration `pred` $p\ [s : S]\ \{\ \mathit{body}\ \}$ is translated into the B definition $p(s) == D[\![s : S]\!]i \wedge M[\![\mathit{body}]\!]i'$.

Again, we have to ensure that unique names are used for the translation of local identifiers in order to avoid clashes between identifier names in B for nested scopes.

Note that ProB inlines definitions to the positions where they are used, which basically is a text replacement. When loading a B machine in ProB, the machine's `DEFINITIONS` section is thus not present anymore.

4.4.14. Assertion Declaration and Run & Check Commands

In Alloy, assertions can be stated using the `assert` declaration. An assertion does not immediately enforce further constraints. Rather, it can later be verified or falsified in a given variable scope, using the `run` and `check` commands. To do so, assertions are named and contain any number of predicates to be checked. For instance, `assert` $a\ \{\ \}$ is a named but empty assertion that is always true.

The `run` command instructs the Alloy Analyzer to search for variable states that satisfy the model's constraints. It can either refer to a named predicate introduced by one of the declarations above or include an explicit Alloy predicate. The `check` command is used to check an assertion by searching for a counterexample.

We introduce an `operation` to the B machine for each `run` command having the translated instructions of the command as its precondition. The operation's substitution is a skip, *i.e.*, we only test if the operation can be executed without any effect on the model. If the translated model satisfies the predicate to be checked, its specific operation is enabled. In case of a `check` command, we proceed analogously but negate the command's instructions within the precondition in order to search for a counterexample. That is, the operation is enabled if a counterexample exists.

Together with the predicate to be checked, both `run` and `check` include a scope used to control the search space. By default, the scope defines an upper bound for the cardinality of a signature. The size can be set to a fixed value by using the keyword `exactly`. We define the translated scope in the precondition of the corresponding operation. For instance, the command `run` p `for` $3\ S$, for a predicate p and an unordered signature S, translates to an operation $\text{run} = \text{PRE card}(E[\![S]\!]i) \leq 3 \wedge M[\![p]\!]i \text{ THEN skip END}$ in

[2]Quantifiers are used for typing but do not enforce restrictions on possible models.

B. In case of a `check` command, the only difference in the translation is to use $\neg M\llbracket p\rrbracket i$.

The Alloy keywords `Int` and `seq` can be used to specify the bit width used to represent integers and the maximum allowed length of sequences.

To execute an Alloy command with ProB one can, *e.g.*, use constraint-based checking as is explained in Section 4.6. In short, constraint-based checking searches for a variable state that satisfies the precondition of an operation considering the machine's properties.

Given that the proof assistants for classical B do not require scopes we also support translating commands without scopes, which can be set via a user preference in ProB. While the resulting B machine does not mimic the Alloy model's behavior exactly, it allows for a more general proof, following the approach we show in Section 4.8.3.

4.4.15. Multiplicity Annotations

Alloy supports the multiplicity annotations `some`, `one`, `lone` and `set`. If no multiplicity is given, the default multiplicity `set` is used.

When translating multiplicity annotations, the semantics are no longer denotational. The predicate $M\llbracket x\ in\ A\ \texttt{->}\ \texttt{one}\ B\rrbracket i$ cannot be encoded as $E\llbracket x\rrbracket i \subseteq E\llbracket A\ \texttt{->}\ \texttt{one}\ B\rrbracket i$, because the property of being a total function is not a closed subset. Hence, we translate the multiplicity annotations as follows:

$$M\llbracket x\ in\ A\ \texttt{->}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{->}\ \texttt{some}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i \wedge \mathrm{dom}(E\llbracket x\rrbracket i) = E\llbracket A\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{->}\ \texttt{one}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \rightarrow E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{->}\ \texttt{lone}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \mathrel{+\!\!\!\!\rightarrow} E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{some}\ \texttt{->}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i \wedge \mathrm{ran}(E\llbracket x\rrbracket i) = E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{some}\ \texttt{->}\ \texttt{some}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \leftrightarrow E\llbracket B\rrbracket i \wedge \mathrm{dom}(E\llbracket x\rrbracket i) = E\llbracket A\rrbracket i \wedge \mathrm{ran}(E\llbracket x\rrbracket i) = E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{some}\ \texttt{->}\ \texttt{lone}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \mathrel{+\!\!\!\!\twoheadrightarrow} E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{some}\ \texttt{->}\ \texttt{one}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \twoheadrightarrow E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{lone}\ \texttt{->}\ B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i \mathrel{+\!\!\!\!\rightarrow} E\llbracket A\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{lone}\ \texttt{->}\ \texttt{some}\ B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i \mathrel{+\!\!\!\!\twoheadrightarrow} E\llbracket A\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{lone}\ \texttt{->}\ \texttt{lone}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \mathrel{\rightarrowtail\!\!\!\!\!\!+\ } E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{lone}\ \texttt{->}\ \texttt{one}\ B\rrbracket i \mathrel{\widehat{=}} E\llbracket x\rrbracket i \in E\llbracket A\rrbracket i \rightarrowtail E\llbracket B\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{one}\ \texttt{->}\ B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i \rightarrow E\llbracket A\rrbracket i$$

$$M\llbracket x\ in\ A\ \texttt{one}\ \texttt{->}\ \texttt{some}\ B\rrbracket i \mathrel{\widehat{=}} (E\llbracket x\rrbracket i)^{-1} \in E\llbracket B\rrbracket i \twoheadrightarrow E\llbracket A\rrbracket i$$

$$M[\![x\ in\ A\ \texttt{one -> lone}\ B]\!]i \ \widehat{=}\ E[\![x]\!]i \in E[\![A]\!]i \twoheadrightarrow\!\!\!\!\!\!\!\rightarrowtail E[\![B]\!]i$$

$$M[\![x\ in\ A\ \texttt{one -> one}\ B]\!]i \ \widehat{=}\ E[\![x]\!]i \in E[\![A]\!]i \rightarrowtail\!\!\!\!\twoheadrightarrow E[\![B]\!]i$$

Note that B does not provide operators for each multiplicity annotation that are equivalent to Alloy's definition, *e.g.*, there is no operator to directly define a total relation in B. We thus translate the corresponding multiplicity annotations to B as relations and add additional constraints.

While ProB supports the translated predicates as typing predicates, AtelierB does not support directly typing the inverse as done, for instance, in $M[\![x\ in\ A\ \texttt{lone ->}\ B]\!]i \ \widehat{=}\ (E[\![x]\!]i)^{-1} \in E[\![B]\!]i \twoheadrightarrow E[\![A]\!]i$. Instead, one has to type the relation itself and add any restrictions on the inverse as additional constraints, *e.g.*, $M[\![x\ in\ A\ \texttt{lone ->}\ B]\!]i \ \widehat{=}\ E[\![x]\!]i \in E[\![A]\!]i \leftrightarrow E[\![B]\!]i \wedge (E[\![x]\!]i)^{-1} \in E[\![B]\!]i \twoheadrightarrow E[\![A]\!]i$.

4.4.16. Post-Processing Optimization Rules

As our translation has to be generalized and applicable to all possible Alloy constructs, some translations might not be ideal for the ProB constraint solver, especially when using singleton sets. For instance, our translation might define an identifier to be an element of a specific set such as $x \in S$. Yet, if this set is a singleton set $S = \{y\}$, the membership relation can be replaced by a simple equality $x = y$.

In order to improve performance, ProB provides many rules to improve the representation of abstract syntax trees prior to solving constraints, which is referred to as an abstract syntax tree cleanup. In the following, we present the additional rewriting rules that arose during the implementation of the translation from Alloy to B mostly caused by the use of singleton sets:

$$\{x\} = \{y\} \rightsquigarrow x = y$$

$$\{x\} \neq \{y\} \rightsquigarrow x \neq y$$

$$x \in \{y\} \rightsquigarrow x = y$$

$$x \notin \{y\} \rightsquigarrow x \neq y$$

$$\{x\} \subseteq \{y\} \rightsquigarrow x = y$$

$$\{x\} \cap \{y\} = \varnothing \rightsquigarrow x \neq y$$

$$\{x\} * \{y\} \rightsquigarrow \{x \mapsto y\}$$

4.5. Translation of Alloy Extensions

Alloy provides several language extensions, for instance, supporting integers or additional constraints for certain types. As the modules are specified in Alloy, we could directly translate them as well. Yet, we aim to provide an idiomatic and more efficient translation to B for each module. The translation of Boolean operations such as `Nand` is trivial and implemented using B's logical operators. In the following, we present the translation of the extensions we currently support besides Boolean operations.

4.5.1. Integers and Natural Numbers

An integer expression in Alloy is a set, just like any other expression. In order to deal with operators that expect a scalar value, Alloy first evaluates the sum of the elements of a set of integers before applying an operator. For instance, given `s={x : Int | x=1 or x=2}`, the first operand in the alloy expression `minus[s,4]` evaluates to 3, so the result is -1. Similarly, the predicate `3 > s` returns false. Empty sets are evaluated to 0.

Since Alloy encodes constraints to SAT, each Alloy command defines a bit width n used to store integers, *i.e.*, the range $-2^{n-1}..2^{n-1}-1$ with one bit being used to represent the sign. Consequently, integer overflows might occur and the Alloy Analyzer may return a model which is invalid outside the given scope. For example, a model might satisfy `plus[5,3] = -8` with a bit width of 4. It is also possible to divide by zero. An option of the Alloy analyzer can be set to exclude models that entail an overflow or division by 0. However, this slows down the analysis process. Thus, when a modeler presumes that a model does not overflow, this option is usually set to off for efficiency reasons, but there is the risk that an overflow goes undetected. Since B has native support for full integers, overflows do not occur.

Although well-defined, the semantics of integers in Alloy is somewhat unnatural. For instance, accepting non-singleton sets as arguments of integer operations is error-prone and might not always be desired. As overflows are a stumbling block in the use of integers in Alloy, we do not want to replicate this behavior by introducing a bit width in B. However, we provide a ProB preference to translate into bounded integers without overflows using ProB's settings for `MININT` and `MAXINT`. Further, we do not want to allow dividing by zero but throw a well-definedness error instead.

For the translation of integer operations, we use an additional semantic function $E_{int}[\![.]\!]i$ that transforms a set of integers into a scalar expression. It uses the Σ operator of B which returns the sum of the elements of a set of integers or 0 if the set is empty. For instance, $\Sigma(z).(z \in 1..4 \mid z)$ returns 10.

As mentioned above, we believe that accepting non-singleton sets as arguments of integer operations is error-prone. We thus decided to provide a ProB preference which enables a strict translation of integers, *i.e.*, only accept singleton sets where integers are expected. This preference is set by default. To do so, we use the definite description operator, noted MU, which is defined as follows:

$$MU(x) = (\{TRUE\} \times x)(TRUE)$$

For instance, $MU(\{1\})$ returns 1. Since the operator uses B's function application, the MU operator has the well-definedness condition that its argument x is a singleton set. Otherwise, the function application would be undefined for empty sets or ambiguous for non-singleton sets. MU does not exist in the B notation, but is supported by ProB.

Let Nr be an integer constant. We have the following four rules for the semantic function $E_{int}[\![.]\!]i$:

$E_{int}[\![Nr]\!]i \;\widehat{=}\; E[\![Nr]\!]i$ for integer constants Nr

$E_{int}[\![p]\!]i \;\widehat{=}\; Nr$ if $E[\![p]\!]i = \{Nr\}$

$E_{int}[\![p]\!]i \;\widehat{=}\; MU(E[\![p]\!]i)$ if ProB preference for a strict integer translation is set

$E_{int}[\![p]\!]i \;\widehat{=}\; \Sigma(z).(z \in E[\![p]\!]i \mid z)$ otherwise

Note that the first two rules of $E_{int}[\![.]\!]i$ are in principle redundant, they "only" improve the performance of the translation (avoiding unnecessary Σ or MU constructs).

Using this definition the integer operations are translated as follows:

$E[\![Nr]\!]i = \{Nr\}$ for integer constants Nr

$E[\![\#p]\!]i \;\widehat{=}\; \{\mathrm{card}(E[\![p]\!]i)\}$

$E[\![min[p]]\!]i \;\widehat{=}\; \{\min(E[\![p]\!]i)\}$

$E[\![max[p]]\!]i \;\widehat{=}\; \{\max(E[\![p]\!]i)\}$

$E[\![plus[p,q]]\!]i \;\widehat{=}\; \{E_{int}[\![p]\!]i + E_{int}[\![q]\!]i\}$

$E[\![mul[p,q]]\!]i \;\widehat{=}\; \{E_{int}[\![p]\!]i * E_{int}[\![q]\!]i\}$

$E[\![minus[p,q]]\!]i \;\widehat{=}\; \{E_{int}[\![p]\!]i - E_{int}[\![q]\!]i\}$

$E[\![div[p,q]]\!]i \;\widehat{=}\; \{E_{int}[\![p]\!]i / E_{int}[\![q]\!]i\}$

$E[\![rem[p,q]]\!]i \;\widehat{=}\; \{E_{int}[\![p]\!]i \; mod \; E_{int}[\![q]\!]i\}$
(currently only works for positive numbers)

$E[\![negate[p]]\!]i \;\widehat{=}\; \{-E_{int}[\![p]\!]i\}$

$M[\![eq[p,q]]\!]i \;\widehat{=}\; E_{int}[\![p]\!]i = E_{int}[\![q]\!]i$

$M[\![gt[p,q]]\!]i \;\widehat{=}\; E_{int}[\![p]\!]i > E_{int}[\![q]\!]i$

$M[\![lt[p,q]]\!]i \;\widehat{=}\; E_{int}[\![p]\!]i < E_{int}[\![q]\!]i$

$M[\![gte[p,q]]\!]i \;\widehat{=}\; E_{int}[\![p]\!]i \geq E_{int}[\![q]\!]i$

$M[\![lte[p,q]]\!]i \;\widehat{=}\; E_{int}[\![p]\!]i \leq E_{int}[\![q]\!]i$

$M[\![zero[p]]\!]i \;\widehat{=}\; E_{int}[\![p]\!]i = 0$

$M[\![pos[p]]\!]i \;\widehat{=}\; E_{int}[\![p]\!]i > 0$

$M[\![neg[p]]\!]i \;\widehat{=}\; E_{int}[\![p]\!]i < 0$

$M[\![\mathit{nonpos}[p]]\!]i \;\widehat{=}\; E_{\mathit{int}}[\![p]\!]i \leq 0$

$M[\![\mathit{nonneg}[p]]\!]i \;\widehat{=}\; E_{\mathit{int}}[\![p]\!]i \geq 0$

$E[\![\mathit{signum}[p]]\!]i \;\widehat{=}\;$ IF $E_{\mathit{int}}[\![p]\!]i < 0$ THEN -1
ELSE IF $E_{\mathit{int}}[\![p]\!]i > 0$ THEN 1 ELSE 0 END END

$E[\![\mathit{next}]\!]i \;\widehat{=}\; \mathrm{succ}$

$E[\![\mathit{prev}]\!]i \;\widehat{=}\; \mathrm{pred}$

$E[\![\mathit{next}[p]]\!]i \;\widehat{=}\; E[\![p.\mathit{next}]\!]i$

$E[\![\mathit{prev}[p]]\!]i \;\widehat{=}\; E[\![p.\mathit{prev}]\!]i$

$E[\![\mathit{nexts}[p]]\!]i \;\widehat{=}\; \{x \mid x \in \mathbb{Z} \wedge x > E_{\mathit{int}}[\![p]\!]i\}$

$E[\![\mathit{prevs}[p]]\!]i \;\widehat{=}\; \{x \mid x \in \mathbb{Z} \wedge x < E_{\mathit{int}}[\![p]\!]i\}$

$E[\![\mathit{larger}[p,q]]\!]i \;\widehat{=}\; \{\max(\{E_{\mathit{int}}[\![p]\!]i, E_{\mathit{int}}[\![q]\!]i\})\}$

$E[\![\mathit{smaller}[p,q]]\!]i \;\widehat{=}\; \{\min(\{E_{\mathit{int}}[\![p]\!]i, E_{\mathit{int}}[\![q]\!]i\})\}$

$E[\![\mathit{min}]\!]i \;\widehat{=}\;$ `MININT`

$E[\![\mathit{max}]\!]i \;\widehat{=}\;$ `MAXINT`

Note that `MININT` and `MAXINT` are user preferences of ProB. Further, we do not consider Alloy's bit width for the translation of `nexts` and `prevs` but return unbounded sets of integers.

The definitions of division and modulo differ slightly between Alloy and B. B uses a floored division [3]. More precisely, the definition of division in B [25] is $n/m = \min(\{x|x \in \mathbb{Z} \wedge n < m * \mathrm{succ}(x)\})$. Furthermore, in B, $x \mod y$ is only defined if x is non-negative and y is positive.

In contrast, the definition used by Alloy permits both cases. Alloy's division rounds towards zero in general, but permits a number of special cases. According to comments in the Alloy utility module `util/integer`, there are three exceptions to the „round to zero" definition of a/b. First, if $a = 0$, the division returns zero. Second, if $a < 0 \wedge b = 0$, the division returns 1; if $a > 0 \wedge b = 0$ it returns -1. Last, if a is the smallest negative integer and $b = -1$, the division returns a. The different definitions of division and modulo [155] can easily be expressed in B by rewriting them to B's floored division [114]. However, as mentioned above, we do not want to completely reproduce the behavior of Alloy regarding integers.

The translation of operations on natural numbers as defined in the Alloy utility module `util/natural`[3] is analogous to integers but considering the condition to be a positive integer. ProB does not implement any special operations for natural numbers.

[3]See `http://alloytools.org/quickguide/util.html`

4.5.2. Relations

Alloy provides a module for common operations and constraints on binary relations. We translate the relational operations as follows where r is a binary relation with domain d and codomain c:

$$E[\![dom[r]]\!]i \;\widehat{=}\; \mathrm{dom}(E[\![r]\!]i)$$

$$E[\![ran[r]]\!]i \;\widehat{=}\; \mathrm{ran}(E[\![r]\!]i)$$

$$M[\![total[r,d]]\!]i \;\widehat{=}\; \forall x.(x \in E[\![d]\!]i \Rightarrow E[\![r]\!]i[\{x\}] \neq \varnothing)$$

$$M[\![functional[r,d]]\!]i \;\widehat{=}\; \forall x.(x \in E[\![d]\!]i \Rightarrow \mathrm{card}(E[\![r]\!]i[\{x\}]) \leq 1)$$

$$M[\![function[r,d]]\!]i \;\widehat{=}\; \forall x.(x \in E[\![d]\!]i \Rightarrow \mathrm{card}(E[\![r]\!]i[\{x\}]) = 1)$$

$$M[\![injective[r,c]]\!]i \;\widehat{=}\; \forall y.(y \in E[\![c]\!]i \Rightarrow \mathrm{card}((E[\![r]\!]i)^{-1}[\{y\}]) \leq 1)$$

$$M[\![surjective[r,c]]\!]i \;\widehat{=}\; \forall y.(y \in E[\![c]\!]i \Rightarrow (E[\![r]\!]i)^{-1}[\{y\}] \neq \varnothing)$$

$$M[\![bijective[r,c]]\!]i \;\widehat{=}\; \forall y.(y \in E[\![c]\!]i \Rightarrow \mathrm{card}((E[\![r]\!]i)^{-1}[\{y\}]) = 1)$$

$$M[\![bijection[r,d,c]]\!]i \;\widehat{=}\; M[\![function[r,d]]\!]i \wedge M[\![bijective[r,c]]\!]i$$

$$M[\![reflexive[r,s]]\!]i \;\widehat{=}\; \mathrm{id}(s) \subseteq E[\![r]\!]i$$

$$M[\![irreflexive[r]]\!]i \;\widehat{=}\; \mathrm{id}(\mathrm{dom}(E[\![r]\!]i)) \cap E[\![r]\!]i = \varnothing$$

$$M[\![symmetric[r]]\!]i \;\widehat{=}\; (E[\![r]\!]i)^{-1} \subseteq E[\![r]\!]i$$

$$M[\![antisymmetric[r]]\!]i \;\widehat{=}\; ((E[\![r]\!]i)^{-1} \cap E[\![r]\!]i) \subseteq \mathrm{id}(\mathrm{dom}(E[\![r]\!]i))$$

$$M[\![transitive[r]]\!]i \;\widehat{=}\; (E[\![r]\!]i \;;\; E[\![r]\!]i) \subseteq E[\![r]\!]i$$

$$M[\![acyclic[r,s]]\!]i \;\widehat{=}\; \forall x.(x \in E[\![s]\!]i \Rightarrow x \mapsto x \notin \mathrm{closure1}(E[\![r]\!]i))$$

$$M[\![complete[r,s]]\!]i \;\widehat{=}\; \forall (x,y).(x \in E[\![s]\!]i \wedge y \in E[\![s]\!]i \wedge \\ x \neq y \Rightarrow x \mapsto y \in (E[\![r]\!]i \cup (E[\![r]\!]i)^{-1}))$$

$$M[\![preorder[r,s]]\!]i \;\widehat{=}\; M[\![reflexive[r,s]]\!]i \wedge M[\![transitive[r]]\!]i$$

$$M[\![equivalence[r,s]]\!]i \;\widehat{=}\; M[\![preorder[r,s]]\!]i \wedge M[\![symmetric[r]]\!]i$$

$$M[\![partialOrder[r,s]]\!]i \;\widehat{=}\; M[\![preorder[r,s]]\!]i \wedge M[\![antisymmetric[r]]\!]i$$

$$M[\![totalOrder[r,s]]\!]i \;\widehat{=}\; M[\![partialOrder[r,s]]\!]i \wedge M[\![complete[r,s]]\!]i$$

4.5.3. Orderings

Alloy data types are universally based on relations. For instance, sets are unary relations while scalars are singleton sets. Signatures are not ordered by default. Yet, Alloy allows declaring a total order on signature elements by defining a signature to be ordered, and offers several operations for element access on ordered signatures. For instance, for an ordered Signature S_o, $S_o/nexts(s)$ returns the set of all successors of $s \in S_o$.

Initially, we translated ordered signatures to B sequences. Sequences are ordered sets of pairs whose domains are finite and coherent sets $1..n$, where $n \in \mathbb{N}$ is the number of elements. Usually, we translate an Alloy signature to a deferred set in B having the same name as described in Section 4.4.3. An ordered signature S_o can then be represented by a sequence of type S_o, *i.e.*, a set of pairs of integers and S_o. B directly offers most of the operations on ordered signatures while others can be implemented using set comprehensions.

However, ProB's performance on predicates involving sequences can be lacking when compared to (sets of) integers. In consequence, we switched to a different translation: The scope of a signature is defined within the run or check statement of an Alloy model. Assuming the ordered signature S_o has size $k \in \mathbb{N}$, we translate it to an interval $s..(s+k-1)$, $s \in \mathbb{N}$, in B. The offset s is used to take into account that ordered signatures can interact, *e.g.*, when computing the union. We thus ensure that ordered signatures are distinct by translating them into disjoint intervals.

Besides that, ordered signatures might interact with unordered ones in Alloy. We then have to define the unordered signature as a set of integers as well to avoid type errors in B. To do so, we check an Alloy model for interactions between ordered and unordered signatures prior to the translation.

We expect the input values of operations on orderings to be singleton integer sets or empty sets. When using integer intervals, the operations *first* and *last* can be translated using `min` and `max` wrapped in a singleton set. For an ordered signature S_o, we define $S_o/next$ and $S_o/prev$ using the successor and predecessor relations of B. The operations $S_o/nexts[e]$ and $S_o/prevs[e]$ are translated using set comprehensions.

We noticed that the relational operations on orderings such as $S_o/lt[e1, e2]$ always return true if the left-hand side is an empty set. If the left-hand side is non-empty and the right-hand side is an empty set on the other hand, the relational operators always return false. In the remaining cases, the relational operators behave as expected from real integers.

In particular, we translate the operations on an ordered signature S_o in Alloy as follows:

$E[\![S_o]\!]i \ \widehat{=}\ m..n$, where m is the lower and n the upper bound of the corresponding integer domain

$E[\![S_o/first]\!]i \ \widehat{=}\ \text{IF } E[\![S_o]\!]i \neq \varnothing\ THEN\ \{\min(E[\![S_o]\!]i)\} \text{ ELSE } \varnothing \text{ END}$

$E[\![S_o/last]\!]i \ \widehat{=}\ \text{IF } E[\![S_o]\!]i \neq \varnothing \text{ THEN } \{\max(E[\![S_o]\!]i)\} \text{ ELSE } \varnothing \text{ END}$

$E[\![S_o/min[es]]\!]i \ \widehat{=}\ \text{IF } E[\![S_o]\!]i \neq \varnothing \text{ THEN } \{\min(E[\![es]\!]i)\} \text{ ELSE } \varnothing \text{ END}$

$E[\![S_o/max[es]]\!]i \ \widehat{=}\ $ IF $E[\![S_o]\!]i \neq \varnothing$ THEN $\{\max(E[\![es]\!]i)\}$ ELSE $\varnothing$ END

$E[\![S_o/next[e]]\!]i \ \widehat{=}\ $ IF $\text{succ}[E[\![e]\!]i] \subseteq E[\![S_o]\!]i$ THEN $\text{succ}[E[\![e]\!]i]$ ELSE $\varnothing$ END

$E[\![S_o/nexts[e]]\!]i \ \widehat{=}\ \{x \mid x \in E[\![S_o]\!]i \wedge \min(\{x\} \cup E[\![e]\!]i) \neq x\}$

$E[\![S_o/prev[e]]\!]i \ \widehat{=}\ $ IF $\text{pred}[E[\![e]\!]i] \subseteq E[\![S_o]\!]i$ THEN $\text{pred}[E[\![e]\!]i]$ ELSE $\varnothing$ END

$E[\![S_o/prevs[e]]\!]i \ \widehat{=}\ \{x \mid x \in E[\![S_o]\!]i \wedge \max(\{x\} \cup E[\![e]\!]i) \neq x\}$

$E[\![S_o/larger[e1, e2]]\!]i \ \widehat{=}\ $ IF $E[\![e1]\!]i \cup E[\![e2]\!]i \neq \varnothing$ THEN $\{\max(E[\![e1]\!]i \cup E[\![e2]\!]i)\}$ ELSE $\varnothing$ END

$E[\![S_o/smaller[e1, e2]]\!]i \ \widehat{=}\ $ IF $E[\![e1]\!]i \cup E[\![e2]\!]i \neq \varnothing$ THEN $\{\min(E[\![e1]\!]i \cup E[\![e2]\!]i)\}$ ELSE $\varnothing$ END

$M[\![S_o/lt[e1, e2]]\!]i \ \widehat{=}\ (E[\![e1]\!]i = \varnothing) \vee (E[\![e1]\!]i \neq \varnothing \wedge E[\![e2]\!]i \neq \varnothing \wedge E[\![e1]\!]i \neq E[\![e2]\!]i \wedge \{\min(E[\![e1]\!]i \cup E[\![e2]\!]i)\} = E[\![e1]\!]i))$

$M[\![S_o/lte[e1, e2]]\!]i \ \widehat{=}\ (E[\![e1]\!]i = \varnothing) \vee (E[\![e1]\!]i \neq \varnothing \wedge E[\![e2]\!]i \neq \varnothing \wedge \{\min(E[\![e1]\!]i \cup E[\![e2]\!]i)\} = E[\![e1]\!]i))$

$M[\![S_o/gt[e1, e2]]\!]i \ \widehat{=}\ (E[\![e1]\!]i = \varnothing) \vee (E[\![e1]\!]i \neq \varnothing \wedge E[\![e2]\!]i \neq \varnothing \wedge E[\![e1]\!]i \neq E[\![e2]\!]i \wedge \{\max(E[\![e1]\!]i \cup E[\![e2]\!]i)\} = E[\![e1]\!]i))$

$M[\![S_o/gte[e1, e2]]\!]i \ \widehat{=}\ (E[\![e1]\!]i = \varnothing) \vee (E[\![e1]\!]i \neq \varnothing \wedge E[\![e2]\!]i \neq \varnothing \wedge \{\max(E[\![e1]\!]i \cup E[\![e2]\!]i)\} = E[\![e1]\!]i))$

4.5.4. Enumerations

In Alloy, an enumeration can be used to define a number of distinct singleton signatures with a common ordered base signature. Enumerations are syntactical sugar, not providing new functionalities but enabling a less verbose specification. For instance, consider the following enumeration S:

```
enum S { S1,S2 }
```

The same behavior can be achieved by defining an abstract ordered signature S and two singleton signatures S_1 and S_2 which extend S:

```
open util/ordering[S]
abstract sig S {}
one sig S1, S2 extends S {}
```

In our current translation to B, we do not consider that enumerations are ordered. We rather translate enumerations based on the declared signatures, *i.e.*, we introduce a deferred set S and two constants S_1 and S_2 which are singleton subsets of S. Furthermore, we set the extending signatures to be distinct. For instance, for the signature S_1, this is done using the additional constraint $S_1 \subseteq S \wedge \text{card}(S_1) = 1 \wedge S_1 \cap S_2 = \varnothing$.

Not translating enumerations as ordered signatures allows ProB to use advanced optimization techniques such as symmetry reduction. In case a model relies on enumerations being ordered, we could of course treat enumerations as ordered signatures and translate as described in Section 4.5.3. In the future, our translator should check automatically if an Alloy models makes use of the ordering of enumerations and translate accordingly.

4.5.5. Sequences

In B, sequences are defined as partial functions with finite and coherent domains $1..n$, where $n \in \mathbb{N}$ is the size of the sequence. Sequences are therefore defined as sets of pairs and might be nested arbitrarily.

In Alloy, the field of a signature, the parameters of a quantification and the arguments of a function can be defined as a sequence of atoms using the `seq` keyword. In contrast to B, the elements of a sequence are enumerated from 0 to $n - 1$.

In consequence, we cannot straightforwardly translate sequences from Alloy to B. If using B's internal representation of sequences, we would need to increase each integer value from Alloy accessing a sequence by one, and decrease each integer value by one which is computed by a sequence operation or used within one. Moreover, there is no counterpart to most of Alloy's sequence operations in B, so we would have to manually implement most of the operations anyway. We thus decided to retain the domain of a sequence defined by Alloy. For the translation of a sequence, we define a partial function with domain $0..n - 1$ as described in Section 4.4.3, where $n \in \mathbb{N}$ is the sequence's cardinality, and use manually implemented operations on sequences without resorting to B's sequence operations.

Alloy allows modifying a set of sequences using a set of elements rather than a single element at a time. For instance, consider the signature `sig T {s:  seq Int}` which defines a field s as a sequence of integers. Assuming that T has a scope of exactly two, *i.e.*, `T = {T$0, T$1}` in the Alloy Evaluator, a call to `T.s.insert[0,{1}+{2}]` is satisfiable. For instance, this results in inserting 1 at position 0 in `T$0.s` and inserting 2 at position 0 in `T$1.s`. As this behavior can also be achieved by accessing each field s of T's instances independently and cannot be described efficiently by a single expression in B, we decided to only allow operations on sequences using single elements.

Therefore, we use an additional semantic function $E_{one}[\![.]\!]i$ that transforms a singleton set into a scalar expression. If the input is not a singleton set, a well-definedness error is thrown by ProB. To do so, we use the definite description operator, noted MU, described in Section 4.5.1. We then define the semantic function as follows:

$$E_{one}[\![x]\!]i \ \widehat{=}\ y \text{ if } E[\![x]\!]i = \{y\}$$

$$E_{one}[\![x]\!]i \;\widehat{=}\; MU(E[\![x]\!]i)$$

It is very similar to the semantic function $E_{int}[\![.]\!]i$, just replacing Σ by the MU operator. Again, the first rule is in principle redundant, but improves the performance of the translation (avoiding the introduction of MU if possible).

By default, an Alloy model defines a maximum allowed length of sequences due to the SAT encoding. The maximum size can be changed within the scope of a `run` or `check` command by using the `seq` keyword.

Let m be the maximum allowed length of sequences of a specific `run` or `check` command, and c_s be the cardinality of the set s in B, *i.e.*, $c_s = \mathrm{card}(E[\![s]\!]i)$. The sequence operations provided by Alloy are translated as follows:

$$E[\![s.\mathit{first}]\!]i \;\widehat{=}\; E[\![s]\!]i[\{0\}]$$

$$E[\![s.\mathit{last}]\!]i \;\widehat{=}\; E[\![s]\!]i[\{c_s - 1\}]$$

$$E[\![s.\mathit{rest}]\!]i \;\widehat{=}\; \text{IF } E[\![s]\!]i = \varnothing \text{ THEN } \varnothing \\ \text{ELSE } \lambda z.(z \in 0..(c_s - 2) \mid E[\![s]\!]i(z+1)) \text{ END}$$

$$E[\![s.\mathit{elems}]\!]i \;\widehat{=}\; \mathrm{ran}(E[\![s]\!]i)$$

$$E[\![s.\mathit{butlast}]\!]i \;\widehat{=}\; E[\![s]\!]i[\{c_s - 2\}]$$

$$M[\![s.\mathit{isEmpty}]\!]i \;\widehat{=}\; E[\![s]\!]i = \varnothing$$

$$M[\![s.\mathit{hasDups}]\!]i \;\widehat{=}\; c_s \neq \mathrm{card}(\mathrm{ran}(E[\![s]\!]i))$$

$$E[\![s.\mathit{inds}]\!]i \;\widehat{=}\; 0..(c_s - 1)$$

$$E[\![s.\mathit{lastIdx}]\!]i \;\widehat{=}\; \{c_s - 1\} \cap 0..(c_s - 1)$$

$$E[\![s.\mathit{afterLastIdx}]\!]i \;\widehat{=}\; \{c_s\} \cap 0..(m - 1)$$

$$E[\![s.\mathit{idxOf}[x]]\!]i \;\widehat{=}\; \text{IF } (E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}] \neq \varnothing \text{ THEN} \\ \{\min((E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}])\} \\ \text{ELSE } \varnothing \text{ END}$$

$$E[\![s.\mathit{lastIdxOf}[x]]\!]i \;\widehat{=}\; \text{IF } (E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}] \neq \varnothing \text{ THEN} \\ \{\max((E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}])\} \\ \text{ELSE } \varnothing \text{ END}$$

$$E[\![s.\mathit{indsOf}[x]]\!]i \;\widehat{=}\; (E[\![s]\!]i)^{-1}[\{E_{one}[\![x]\!]i\}]$$

$$E[\![s.\mathit{append}[t]]\!]i \;\widehat{=}\; (0..(m-1)) \lhd (E[\![s]\!]i \cup \\ \lambda z.(z \in c_s..(c_s + c_t - 1) \mid E[\![t]\!]i(z - c_s)))$$

$$E[\![s.\mathit{add}[x]]\!]i \;\widehat{=}\; \text{IF } c_s < m \text{ THEN } E[\![s]\!]i \cup \{c_s \mapsto E_{one}[\![x]\!]i\} \\ \text{ELSE } E[\![s]\!]i \text{ END}$$

$E[\![s.delete[j]]\!]i \ \widehat{=}\ $ IF $(0 \leq E_{int}[\![j]\!]i)$ THEN
$(0..(E_{int}[\![j]\!]i - 1) \lhd E[\![s]\!]i) \cup \lambda z.(z \in E_{int}[\![j]\!]i..(c_s - 2) \mid E[\![s]\!]i(z + 1))$
ELSE $E[\![s]\!]i$ END

$E[\![s.setAt[j, x]]\!]i \ \widehat{=}\ $ IF $(E_{int}[\![j]\!]i \geq 0 \wedge E_{int}[\![j]\!]i \leq c_s)$ THEN
$E[\![s]\!]i \lhd\!\!\!- \{E_{int}[\![j]\!]i \mapsto E_{one}[\![x]\!]i\}$ ELSE $E[\![s]\!]i$ END

$E[\![s.insert[j, x]]\!]i \ \widehat{=}\ 0..(m - 1) \lhd ((0..(E_{int}[\![j]\!]i - 1) \lhd E[\![s]\!]i) \cup \{E_{int}[\![j]\!]i \mapsto E_{one}[\![x]\!]i\} \cup \lambda z.(z \in (E_{int}[\![j]\!]i + 1)..c_s \wedge (z - 1) \in \mathrm{dom}(E[\![s]\!]i) \mid E[\![s]\!]i(z - 1)))$

$E[\![s.subseq[from, to]]\!]i \ \widehat{=}$
IF $E_{int}[\![from]\!]i \geq 0 \ \wedge\ E_{int}[\![from]\!]i \leq E_{int}[\![to]\!]i \ \wedge\ E_{int}[\![to]\!]i < c_s$ THEN
$\lambda z.(z \in 0..(E_{int}[\![to]\!]i - E_{int}[\![from]\!]i) \mid E[\![s]\!]i(z + E_{int}[\![from]\!]i))$
ELSE $\varnothing$ END

Note that in several translations of sequence operations, *e.g.*, in the translation of `s.append[t]`, we could use a set comprehension instead of a lambda expression. However, since lambda expressions constitute total functions, they improve performance when solving constraints.

As can be seen in the translations, the maximum allowed length of sequences influences the behavior of several operations. For instance, the result of appending two sequences is truncated if it exceeds the scope of sequences. As described in Section 4.4.14, we usually translate all commands into the same B machine. In case at least two commands define a different maximum allowed length of sequences, our translation would possibly behave differently than the Alloy model does as we can only consider a single scope at a time when translating operations on sequences.

We thus analyze an Alloy model prior to the translation to determine if it uses sequences within differently scoped commands, *i.e.*, commands that do not define a common maximum allowed length of sequences. If so, we only translate a single command at a time which can be selected by the user within ProB's graphical user interface. Otherwise, all commands are translated into the same B machine.

Note that the Alloy Analyzer does not enforce that sequence operations are called with well-defined sequences[4]. In contrast, our translation to B provides static type safety which improves error detection.

Further, we noticed that the Alloy Analyzer behaves inconsistently regarding the evaluation of preconditions. The operations `delete`, `setAt` and `insert` are unsatisfiable if their precondition is false while the other operations always succeed, *e.g.*, returning the input sequence if an operation's precondition is false. To achieve a consistent behavior regarding the use of preconditions, we decided to not fail for the mentioned operations if their precondition is false but return the input sequence.

However, we could also translate operations to be unsatisfiable if their precondition is false. As a B expression cannot fail without throwing an error, *e.g.*, a well-definedness error, we would need to pass a flag in the environment i to inform the preceding predicate to fail.

[4]See `http://alloytools.org/quickguide/seq.html`

4.6. Tooling

In the following section, we first give an overview over the tooling used to automate the translation from Alloy to B. Additionally, we give insight into the Prolog implementation in Section 4.6.2 and further implementation details.

4.6.1. Overview

As shown in Figure 4.2, our automatic translation relies on two software tools. The first component is a small application (around 500 lines of code, not counting tests) written in Kotlin and running on the JVM. Its purpose is to use the original parser and typechecker of the Alloy Analyzer to parse Alloy files and pretty print the resulting abstract syntax tree into a Prolog representation that can be loaded by ProB's core. During this first translation, some changes in representation are done in order to make all information available to the Alloy Analyzer available to ProB as well, *i.e.*, we extend the abstract syntax tree with additional information.

Moreover, we generalize the types provided by the Alloy parser to their top-level signatures. For instance, let S be a top-level signature, and S_1, S_2 are both signatures that extend S. The type of the expression $S_1 + S_2$ provided by the Alloy Analyzer's parser is a set (a relation with arity 1) of the two types S_1 and S_2. For our translation to B, it is more intuitive and necessary to use the most general type except for `univ` if present. In the given example, the most general type is a set of type S. We are then able to easily set up typing constraints for each Alloy construct during the translation to B. Otherwise, we would need to generalize types on demand in Prolog. However, two signatures might not have a parent type except for `univ`. Since we want to avoid the universe type in B as described in Section 4.4.4, we define a parent type for each of such signature collections as a deferred set in B.

Afterward, the Alloy abstract syntax tree is read by ProB and translated into ProB's internal representation of a B machine, following the translation rules discussed in Section 4.4. The result is an untyped B abstract syntax tree, that is fed into the regular B typechecker. Once typed, it can be used inside the model checker or animator as well as in the constraint solver. Furthermore, all backends available to ProB consume the same internal representation, *i.e.*, the resulting typed B abstract syntax tree can be fed to them as well. For instance, a constraint solver which uses the Alloy Analyzer's Kodkod API [45] to translate B to SAT is available [46]. Furthermore, an integration with the SMT solver Z3 [47] can be used to solve constraints [48], or a combination of the CLP(FD) and SMT backends where both solvers share constraints [48].

As described in Section 4.4.14, `run` and `check` commands are translated to B machine operations. To execute an Alloy command with ProB one can either use model checking, *i.e.*, try all possible ways to instantiate the constants of the B translation and examine whether the operation is covered, or use constraint-based checking, *e.g.*, using the `cbc_sequence` command of ProB, which sends the operation's precondition and the machine's properties to ProB's constraint solver. In the latter case the machine's properties are considered as we translate several constraints in this machine section, for

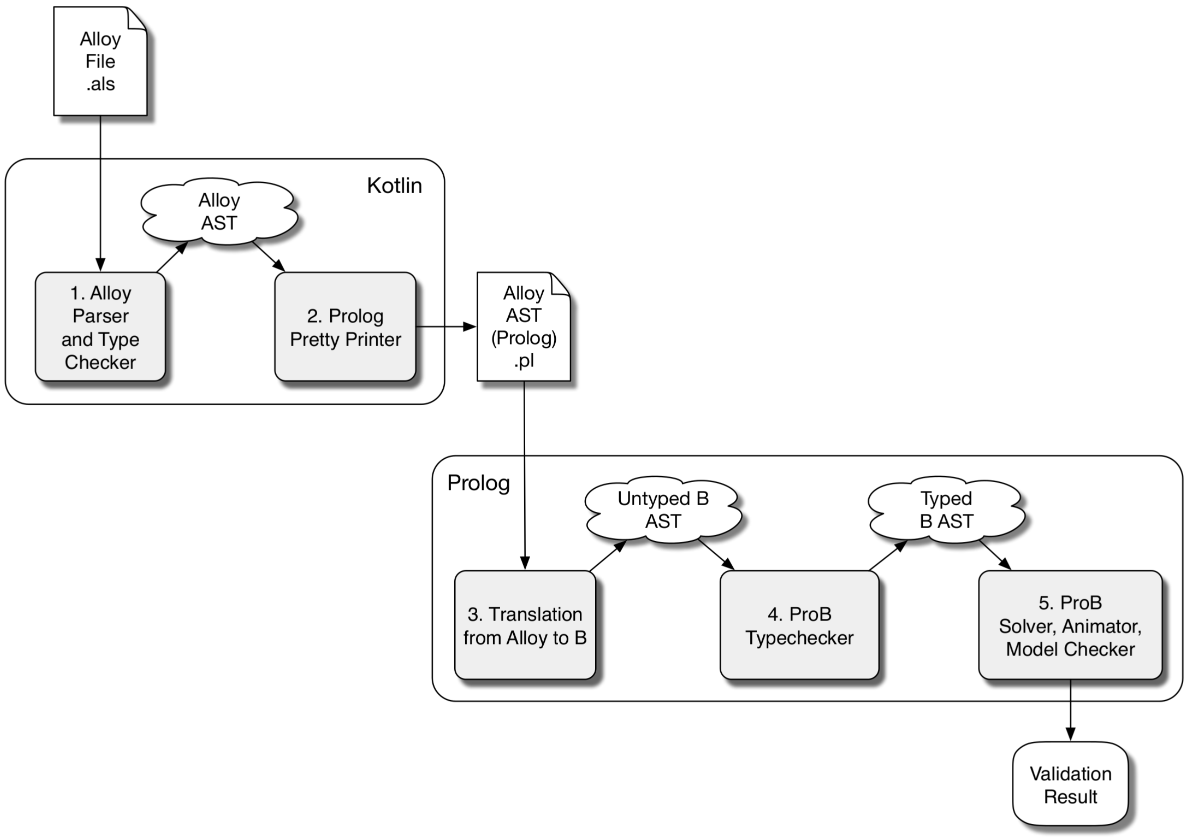


Figure 4.2.: A visualization of the different phases of the Translation from Alloy to B.

instance, constraints on the fields of signatures.

In order to use the generated B machine inside other B tools such as AtelierB, ProB can export the internal representation to a regular B machine file. Further, we provide a ProB preference to translate a single command into the B machine's `ASSERTIONS` section rather than creating a machine operation as described in Section 4.4.14. As the assertion of a `check` command is negated to search for a counterexample, we remove the negation when adding the constraint to the B machine assertions. This enables the generation of proof obligations in AtelierB. The command to be translated can be selected by the user within ProB's graphical user interface.

4.6.2. Prolog Encoding of Translation Rules

As stated above, we use a small Kotlin library to extract the AST generated by the Alloy Analyzer's parser and typechecker. The resulting file is then read by ProB's Prolog core.

The mathematical rules featured in our translation can quite often be translated to Prolog clauses straightforwardly. In particular, the implementation usually consists of single translation rules being implemented by a single corresponding Prolog clause. This leads to an implementation that is close to the formal specification. In consequence, the

Listing 4.5: Parts of the Prolog code that translates Alloy's set union to B.

```
translate_binary_e_p(Binary,TBinary) :-
  Binary =.. [Op,Arg1,Arg2,_Type,POS],
  alloy_to_b_binary_operator(Op,BOp),
  translate_e_p(Arg1,TArg1),
  translate_e_p(Arg2,TArg2),
  translate_pos(POS,BPOS),
  TBinary =.. [BOp,BPOS,TArg1,TArg2].

alloy_to_b_binary_operator(plus,union).
  ...
```

implementation is comprehensible and can easily be reviewed, extended and adapted. Take for example the rule for the Alloy plus operator:

$$E[\![p + q]\!]i \ \widehat{=}\ E[\![p]\!]i \cup E[\![q]\!]i$$

Listing 4.5 shows parts of the Prolog code translating the set union from Alloy to B, where `translate_e_p` is the Prolog name of the function $E[\![.]\!]i$.

The code is somewhat more generic and factors several rules into one, namely all binary operators that can be translated directly to B operators. Additional operators that can be directly translated are given by Prolog facts further defining `alloy_to_b_binary_operator`. You can also see that the code translates Alloy's position information to B (for error messages). The keen observer will note that the environment i is not present; it is currently encoded using assert/retract (*i.e.*, as Prolog global variables).

4.7. Empirical Evaluation

To validate the correctness of our translation we have applied it to a variety of mathematical laws (see Fig. 4.6) and have checked that ProB does not find counterexamples to those laws on the translated B machines.

Furthermore, we have translated several Alloy models to B. In the following, we give a brief empirical evaluation of selected models, comparing the performance of the Alloy Analyzer and ProB.

The benchmarks were run on an Intel Core i7-8750H CPU (2.2GHz) and 16 GB of RAM. We use the median time of five independent checks. The runtime of the Alloy Analyzer includes generating the conjunctive normal form and uses the SAT4J backend. For the ProB constraint solver we purely use the CLP(FD) backend with a linear enumeration order and without extensions such as Kodkod [46] or Z3 [48]. Of course, despite constraint solving itself, processing an Alloy model using the Alloy Analyzer's parser, pretty printing the model to Prolog, transforming types as described in Section 4.6, and translating the model to B needs some time. However, this is not a bottleneck for performance. In the following, we thus assume each model to be loaded

Listing 4.6: An exemplary Alloy model for checking of mathematical laws.

```
abstract sig setX { }
one sig V {
  SS: set setX,
  TT: set setX,
  VV: set setX,
  Empty: set setX
}
fact EmptySet {  no V.Empty }
assert SetLaws {
  V.SS + V.SS = V.SS
  no V.SS - V.SS
  V.SS = V.SS & V.SS
  V.SS - V.Empty = V.SS
  V.Empty =  V.SS -  V.SS
  V.Empty -  V.SS =  V.Empty
  V.SS +  V.TT =  V.TT +  V.SS
  ...
}
check SetLaws for 5 setX, 7 int
```

in the Alloy Analyzer and ProB, *i.e.*, we only measure the impact of our translation on finding solutions for a model's constraints.

Since the Alloy Analyzer translates models to SAT, we assume it to be efficient for mostly relational models. However, SAT encoding is often inefficient for integers, *e.g.*, one has to encode arithmetic using binary adders. ProB on the other hand has native support for integers, hopefully leading to better performance for arithmetic calculations. In contrast, relations often cause a combinatorial explosion, which results in weaker performance compared to the Alloy Analyzer. To explore both extremes, we chose different models for performance comparison.

An exact opposite to our translation has been presented by Plagge and Leuschel [46], which uses the Alloy Analyzer's Kodkod API [45] to translate B to SAT. We further solve the translated models using ProB with its Kodkod backend in order to investigate if our translation from Alloy to B is needlessly complicated. If this is not the case, we expect the runtime to be only slightly larger than the one of the Alloy Analyzer. Note that in recent work [156] we have shown that an integration of the Alloy and ProB backends can be very useful for complex constraint satisfaction problems.

We start with translating an Alloy model of the river crossing puzzle, a type of transport puzzle with the goal to carry several objects from one river bank to another. There are constraints defining which objects are safe to be left alone, *e.g.*, a fox cannot be left alone with a chicken. The model is interesting for our performance evaluation as it uses an ordered signature for states. The Alloy Analyzer finds a solution in 21 ms (6 ms only SAT solving). Although the translated model is valid, ProB fails to find a solution in less than 5 min.

The B machine defines three relations, two of which have an ordered signature for a

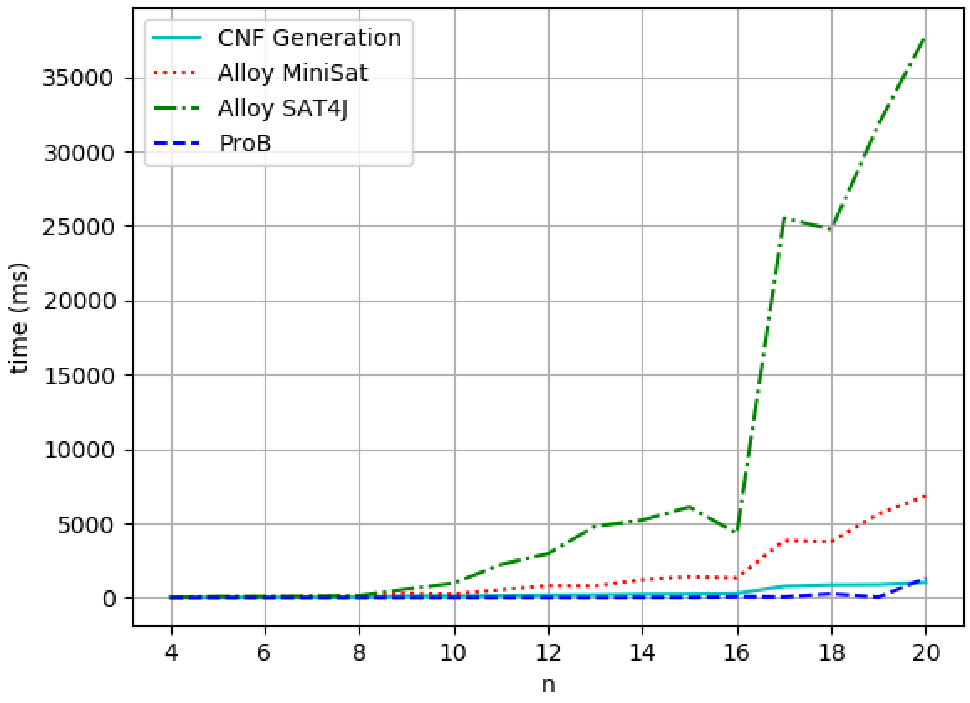


Figure 4.3.: Visualization of the Alloy Analyzer's and ProB's constraint solver's runtimes for finding a single solution for the n queens puzzle with varying n.

domain. Using a total function instead of a relation improves performance: ProB now finds a solution in about 7 s. After rewriting the model in idiomatic B style by hand, ProB can solve it in about 80 ms. However, this translation is a manual optimization using background knowledge and cannot simply be generalized. Using the Kodkod backend of ProB does not improve performance significantly. This indicates that for this specific model our translation is not performant which is most likely caused by the translation of ordered signatures.

Besides the river crossing puzzle, we translated a model of the n queens problem as it makes use of integer arithmetic. Here, the goal is to place n queens on a $n * n$ chess board without two queens threatening each other. The chess board is represented as tuples of row and column, encoded as integers.

We evaluated the n queens model for $n \in 4..20$ using ProB and the Alloy Analyzer with the MiniSat and SAT4J backend. As a comparison, we also measured the time that the Alloy Analyzer needs to generate the conjunctive normal form. The evaluation in Figure 4.3 shows that ProB is the fastest solver for the chosen model. ProB's runtime for solving the constraints ranges from 5 to 1328 ms. The time needed for generating the conjunctive normal form is similar to the time ProB needs for solving the constraints and ranges from 18 to 1028 ms. The solving time of the Alloy Analyzer gets worse when increasing the bit-width for $n > 8$ and $n > 16$. In Figure 4.3 we can see that the runtime of each solver is increasing non-linearly, especially when using the Alloy Analyzer with the SAT4J backend. On the one hand, this might be caused by inaccuracies in our measurements. On the other hand, the constraints might be easier to solve for specific configurations. ProB's runtime for $n = 20$ is an outlier and the cause of this performance drop needs to be investigated more thoroughly. As a comparison, ProB can solve the translated model for $n = 21$ in about 100 ms. Further, when using a randomized enumeration order, ProB can solve the translated model for $n = 20$ in about 80 ms. Note that an idiomatic B version of the n queens puzzle for $n = 20$ can be solved in around 20 ms by ProB. Altogether, it can be seen that integers are a bottleneck for performance when encoding constraints to pure SAT problems.

As a rather simple benchmark, we translated a model of the knights and knaves puzzle. The puzzle defines two types of humans, which either always tell the truth (knights) or always lie (knaves). The goal is to determine the type of several persons from a set of statements each made by one person. The model of the puzzle that we used contains three individual settings with statements made by two or three persons. The model just uses joins, set unions as well as one existential quantification. The Alloy Analyzer finds a solution for the model in 10 ms (6 ms only SAT solving) while ProB needs 5 ms. Solving the translated model with ProB and its Kodkod backend needs about 150 ms, which is most likely caused by the additional overhead of translating B to Kodkod.

Furthermore, we translated a model of the so-called jobs puzzle [157], which defines eight distinct jobs and four persons whose names imply their gender. The goal is to allocate two different jobs to each person and establish the relationships between male and female persons considering a set of constraints. For instance, a constraint states that the husband of the chef is the telephone operator. Besides the common join operations, the model uses a predicate from the extension `util/relation`, defines a field to be quantified by `some`, and uses three quantifications as well as two cardinality constraints. The Alloy Analyzer finds a solution in 23 ms (9 ms only SAT solving). The ProB constraint solver is not able to find a solution within several minutes. As our translation from Alloy to B has to be generalized, some translations considering certain arities are currently not ideal for the ProB constraint solver. To counter this, we intend to provide additional rules to rewrite B abstract syntax trees prior to solving constraints as described in Section 4.4.16 and improve the constraint solver in general. When using the Kodkod backend of ProB, the translated model can be solved in about 50 ms. This shows that the translated B model is not needlessly complicated but contains specific constructs that cannot be handled efficiently by ProB's CLP(FD) backend. As a comparison, an idiomatic B version of the Jobs puzzle [158] can be solved by ProB's CLP(FD) backend in about 150 ms.

To obtain further benchmarks, we translated a model of the Zebra puzzle (also called Einstein's puzzle). The goal is to find a person owning a specific pet for given constraints describing the preferences and houses of a group of five persons. There is only one solution. The model defines one ordered signature, five unordered signatures, and uses fifteen existential quantifications. The Alloy Analyzer finds the solution in 12 ms (5 ms only SAT solving). ProB on the other hand currently needs 748 ms to find the

Table 4.1.: Performance evaluation of the Alloy Analyzer and ProB's constraint solver.

Model	Runtime in ms	
	Alloy	ProB
River Crossing Puzzle	21	$> 300\,000$
4 Queens Puzzle	26	5
8 Queens Puzzle	91	8
12 Queens Puzzle	820	16
16 Queens Puzzle	1334	78
20 Queens Puzzle	6850	1328
Knights and Knaves Puzzle	10	5
Jobs Puzzle	23	$> 300\,000$
Zebra Puzzle	12	748
Towers of Hanoi Puzzle	5201	$> 300\,000$

solution. Again, some constraints are not ideal for ProB and require improvements to the post-processing described in Section 4.4.16 or the constraint solver itself. In this case, using the Kodkod backend of ProB neither improves nor worsens performance. Note that ProB can solve the original Z version of the Zebra puzzle in about 100 ms.

Lastly, we translated a model of the towers of Hanoi puzzle, with three stakes and several discs with different sizes. The model we use defines three ordered signatures, several joins and nested quantifiers. The Alloy Analyzer finds a solution in about 5.2 s while ProB is currently not able to find a solution within several minutes. Using the Kodkod backend of ProB does not improve performance significantly. In this case, our translation of orderings as presented in Section 4.5.3 is inefficient for constraint solving. The Alloy model defines a signature field as a relation between three ordered signatures: `sig` *State* { *on* : *Disc* `->` `one` *Stake* }. In our current translation to B, this results in a possibly large set leading to a bad performance. To counter this, we want to investigate the causes of performance loss more thoroughly and improve our translation wherever possible. Moreover, we want to investigate a translation into a (symbolic or explicit) model checking rather than a constraint satisfaction problem. That is, in case access on ordered elements is linear, we can encode orderings as B machine states using machine variables and operations on orderings as state transitions using machine operations. Doing so, the ProB model checker can be used to find solutions for a model which uses ordered signatures. Note that the ProB model checker can solve manually specified B versions of the river crossing puzzle in about 100 ms and the towers of Hanoi puzzle in about 250 ms.

In summary, we have translated several Alloy models of well-known logic puzzles to classical B. As pointed out, our translation is not optimal for models using relational operators or ordered signatures regarding ProB's performance in solving constraints. Yet, ProB outperforms the Alloy Analyzer for models using integers by several orders of magnitude. Table 4.1 summarizes the comparison of the Alloy Analyzer's and ProB's

Listing 4.7: An Alloy model demonstrating the unsoundness of integers.

```
open util/integer
abstract sig setX { }
one sig V {
  SS:   setX -> setX
}
assert Bug {
  #(V.SS)>1 implies #(V.SS->V.SS)>3
  #(V.SS->V.SS)=0 iff no V.SS
}
// for 8 int Translation capacity exceeded
check Bug for 3 setX, 7 int
```

runtimes in solving the presented models. We used a maximum solver timeout of 5 min. Further, we present the results using the translation as is without any manual optimization of the generated B code and without using the Kodkod or Z3 backend of ProB. The presented times of the Alloy Analyzer for solving the n queens puzzle are the ones using the Minisat backend.

4.8. Improvements Over Existing Alloy Tools

Even though our translation cannot always compete with the Alloy Analyzer as we have demonstrated in Section 4.7, it provides several interesting improvements and applications.

4.8.1. Integers

Mathematically speaking, the integers in Alloy are unsound when overflow detection is turned off. In contrast, ProB has multi-precision integers without overflows[5]. According to Milicevic and Jackson [159] the Alloy Analyzer can detect models with overflows, but to our knowledge cannot detect where an overflow has prevented a model being found. For this purpose, an alternative to translating a model to B would be to use an SMT-based backend for Alloy [160–162].

For example, for the model shown in Listing 4.7 Alloy 4.2 finds a counterexample, while ProB correctly determines that no counterexample exists. If overflows are permitted (the default), the Alloy Analyzer finds a counterexample for the first formula. If overflows are forbidden, no counterexample is detected by the Alloy Analyzer for the first formula, but then a counterexample is found for the second one. With higher integer ranges the translation fails.

[5]CLP(FD) overflows are caught and handled by custom implementation.

Listing 4.8: An exemplary Alloy model using higher-order quantification.

```
open util/integer
abstract sig setX { }
one sig V {
  SS:   setX -> setX,
  TT:   setX -> setX
}
assert HO {
  V.SS + V.SS = V.SS
  all xx : V.SS | (xx in V.TT implies xx in V.SS & V.TT)
}
check HO for 3 setX
```

4.8.2. Higher-Order Quantification

The universal quantification shown in Listing 4.8, using the same signatures as in Listing 4.7, causes an error. The Alloy Analyzer states that analysis cannot be performed since it requires higher-order quantification that could not be skolemized. ProB, on the other hand, can check the validity of this assertion. An extension of Alloy called Alloy* [154] might be able to handle this example. In the future, we would like to investigate translating Alloy* models to B.

4.8.3. Proof

Finally, our translation to B also makes it possible to apply existing provers for the language, such as AtelierB [121], to translated Alloy models. One could thus try to develop a proof assistant for Alloy, similar to the work pursued by Ulbrich et al. [163] via a translation to the first-order logic supported by Key.

In the example shown in Listing 4.9, we can prove the assertion using AtelierB's prover for any scope, by applying it to the translated B machine. We check that the move predicate, removing one element from `src` and adding it to `dst`, preserves the invariant `src+dst=Object`, *i.e.*, that the union of `src` and `dst` covers exactly `Object`.

Note that our translation does not (yet) generate an idiomatic B encoding, with `move` as a B operation and `src+dst=Object` as an invariant: it generates a check operation encoding the predicate `add_preserves_inv` with universal quantification. Listing 4.10 shows the B machine we have input into AtelierB. It was obtained by pretty-printing from ProB. For the translation from Alloy to B, we enabled the preference to translate a model without scopes described in the end of section Section 4.4.14 as well as the preference to translate a single command into the B machine assertions described in the end of Section 4.6.1 (so that AtelierB generates the desired proof obligation).

Listing 4.9: An exemplary Alloy model to prove an assertion in AtelierB.

```
sig Object {}
sig Vars {
  src,dst : Object
}
pred move (v, v': Vars, n: Object) {
  v.src+v.dst = Object
  n in v.src
  v'.src = v.src - n
  v'.dst = v.dst + n
}
assert add_preserves_inv {
  all v, v': Vars, n: Object |
      move [v,v',n] implies  v'.src+v'.dst = Object
}
check add_preserves_inv for 3
```

4.9. Related and Future Work

Translations to Alloy have been pursued from B [164, 165] and also Z [166]. Rather than translation directly to Alloy, a translation from B to Kodkod has been introduced and implemented inside ProB[46].

Other formal languages have previously been translated to B as well, *e.g.*, Z [167] and TLA$^+$[132]. A comparison between TLA$^+$and Alloy has been presented by Macedo and Cunha [168].

The original paper by Jackson [39] (notably Figure 2) provides a semantics of the kernel of Alloy in terms of logical and set-theoretic operators. Our translation rules can be seen as an alternate specification of this semantics, using the B operators and also using B quantification. Future work could be a formal proof of the equality of the different semantics given for Alloy.

Another, albeit less thorough approach, would be to implement a combined solver that runs the Alloy Analyzer and ProB in conjunction and thus verifies the results using a double chain.

While we strive for full support of the Alloy language, we currently do not provide custom implementations for all available utility modules. In particular, we are missing implementations for the translation of common operations on graphs and ternary relations. We currently just translate these modules using our tool as they are defined in Alloy. Of course, the resulting translation might not be as efficient as providing custom implementations.

Furthermore, we intend to translate Alloy* [154] and Electrum [169] (which is a temporal extension of the Alloy modeling language) to B. As B and ProB have support for higher-order quantification and linear temporal logic, translation should be straightforward.

While our translation of orderings, as presented in Section 4.5.3, allows translating

Listing 4.10: A translated Alloy model to prove an assertion in AtelierB.

```
MACHINE alloytranslation
SETS /* deferred */
  Object; Vars
CONCRETE_CONSTANTS
  src_Vars, dst_Vars
PROPERTIES
    src_Vars : Vars --> Object
  & dst_Vars : Vars --> Object
ASSERTIONS
  !(v,v_,n).(v : Vars & v_ : Vars & n : Object
   =>
   (src_Vars[{v}] \/ dst_Vars[{v}] = Object &
    v |-> n : src_Vars &
    src_Vars[{v_}] = src_Vars[{v}] - {n} &
    dst_Vars[{v_}] = dst_Vars[{v}] \/ {n}
    =>
    src_Vars[{v_}] \/ dst_Vars[{v_}] = Object)
   )
END
```

arbitrary Alloy models, the resulting B machine is often suboptimal for ProB's solving kernel as shown in Section 4.7. To improve performance, we want to investigate alternative translations of orderings. For instance, we could impose an order on the elements of a signature S by defining a bijective function $S \rightarrowtail\!\!\!\!\twoheadrightarrow 0..(\text{card}(S) - 1)$ allocating unique indices to the elements. Further, we want to investigate a translation into a (symbolic or explicit) model checking rather than a constraint satisfaction problem. In particular, we intend to translate predicates over states and their successors into B operations. While this is not possible in general, *e.g.*, in the presence of predicates relating more than two states, it would allow us to use symbolic model checking algorithms [153] to find solutions.

Near and Jackson [170] presented an imperative extension of Alloy, *i.e.*, making a step towards B and its operations. Similarly, Frias et al. [171, 172] extended Alloy with actions. Cunha [173] presented an approach using bounded model checking for temporal properties in Alloy. It would be interesting to extend our translation and produce idiomatic B machines with B operations from such Alloy models.

As soon as our translation relies more on operations, we want to investigate translating into a set of models linked by refinement rather than translating an Alloy model into a single B machine. However, since we currently do not impose any restrictions on the Alloy model to be translated, it remains to be seen to what extent automatic refinement techniques such as the one used in BART [174] or the one introduced by Iliasov et al. [175] can be used efficiently.

4.10. Conclusion

In summary, we have presented an automatic translation of Alloy to B, which provides an alternative semantics definition of Alloy, and enables proof and constraint solving tools of B to be applied to Alloy specifications. We have shown empirically that for certain constraints, the B language tools in general and ProB in particular are superior to the Alloy Analyzer and its SAT backend. For other constraints however, the Alloy Analyzer outperforms ProB. As expected, different backends exhibit different strengths and weaknesses. Using our translation, we make ProB's backends available to Alloy users, enabling them to experiment with technologies other than the ones employed by the Alloy Analyzer.

The formal definition of the translation revealed both shortcomings and elegant features of Alloy and B. One aspect where B is awkward is the treatment of tuples: many encodings exist and the modeler has to know which one is being used. Associative tuples with flexible join and projection operations (similar to database operations) would be a very useful addition to B.

The object-oriented notation of Alloy makes specifications more modular and easier to read than classical B and is closer to a UML-like model that most conventional designers are familiar with. In B, one can use records or use B's machine decomposition statements such as `INCLUDE`, but the syntax is not as handy as Alloy's.

Alloy allows expressing certain constructs in a much more concise fashion, showing that B sometimes is not as expressive as desired. However, the same applies for Alloy as well. Multiplicity annotations in Alloy are inspired from conceptual modelling notations, but their mathematical representation relies on well-known classes of functions that the B notation natively supports concisely. We have also shown that B can be much more concise and expressive especially when dealing with integers.

Alloy is not tailored for transition system analysis since system behavior is analyzed using bounded traces. ProB offers sophisticated tools for analyzing the transition graph of a system by supporting invariant and deadlock checks, LTL[e] and CTL, fairness constraints, reachability analysis, and model-based testing.

In general, a comparison and translation such as the one presented in this article should inspire the evolution of both languages. We hope that our translation can serve as a vehicle of communication between the Alloy and B communities.

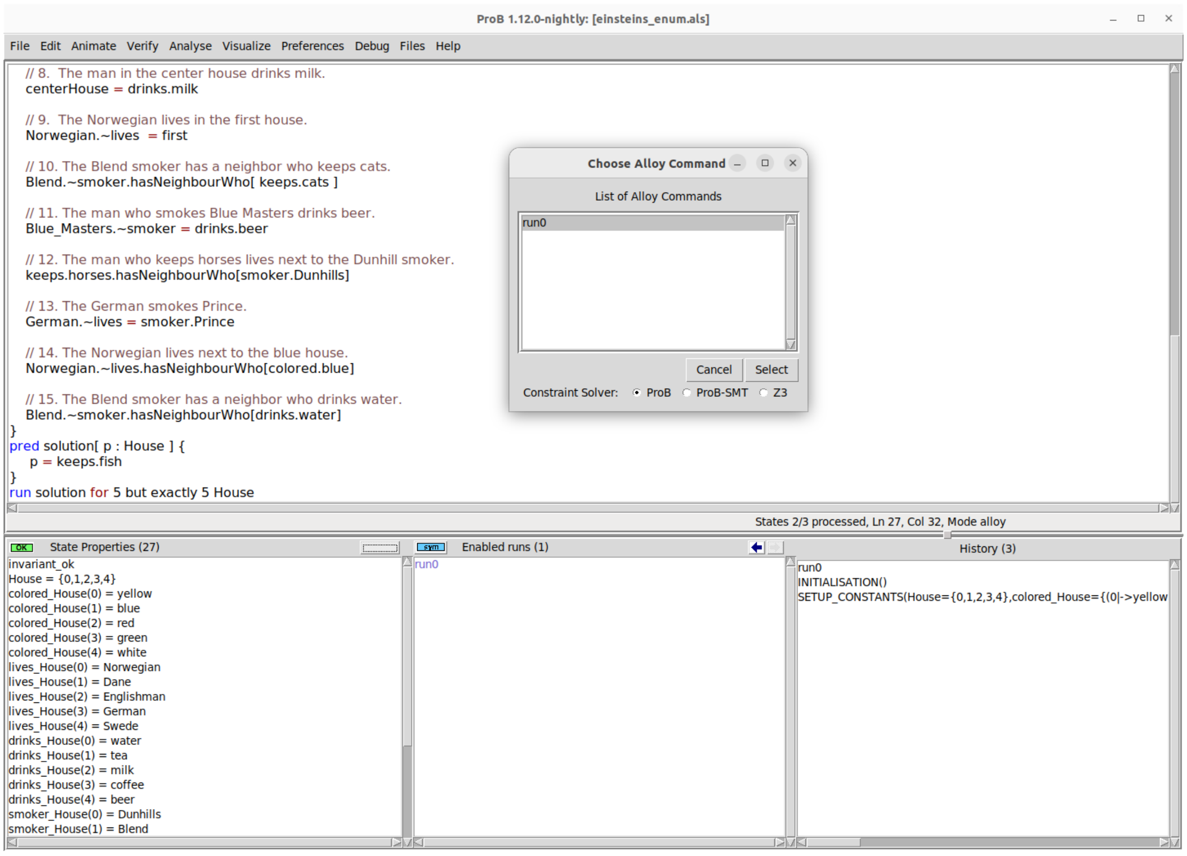
5. Additional Experiments and Considerations

In the following, we present an extended integration of Alloy in ProB's graphical Tcl/Tk user interface as well as additional experiments for the evaluation of our automated translation from Alloy to classical B using the Alloy Analyzer as well as different constraint solving backends of ProB. Further, we point out limitations of our automated translation, describe the new major release of Alloy including mutable state and linear temporal logic, and propose a possible translation from the new language constructs introduced in Alloy 6 to B.

5.1. Extended Verification of Commands

Alloy commands are translated to B machine operations defining a command's scope in its precondition as described in Section 4.4.14. The scope of an Alloy command constrains the sizes of signatures. Our translation introduces a deferred set in B for each signature. One peculiarity of ProB is that it determines the cardinality of deferred sets when loading a machine. ProB provides an option to define a global cardinality for deferred sets while the user is also able to manually set the cardinality of a specific deferred set using a B definition. Yet, these cardinalities cannot be set dynamically after loading a machine in ProB. For the translation from Alloy to B, this means that we can only translate commands which define the same scope in a single translated machine. For instance, consider two Alloy commands constraining the same signature but setting its scope to be one in the first and two in the second command. For the first translated operation, ProB would assume a cardinality of one for the deferred set introduced for the Alloy signature. The second command can then not be translated to B in the same machine since the scope of one could lead to a spurious counterexample being found. Conversely, translating the second command would apply a scope of two in ProB which could result in different behavior for the first command.

We thus decided to translate an Alloy model for a specific command only, which we refer to as the main command. By default, we use the first Alloy command that is defined in a model. We first search for all commands that define the same scope as the main command. Afterward, all these commands are translated into a single B machine. This guarantees that a translated B machine defines scopes that are consistent with the corresponding Alloy model. If a command to be checked is currently not translated due to a different scope, we assign this command to be the new main command and translate the Alloy model from scratch. Of course, this adds the additional overhead of


ProB 1.12.0-nightly: [einsteins_enum.als]
File Edit Animate Verify Analyse Visualize Preferences Debug Files Help
Choose Alloy Command
List of Alloy Commands
Cancel
Select
OK
State Properties (27)
sym
Enabled runs (1)
History (3)

```
  // 8.  The man in the center house drinks milk.
  centerHouse = drinks.milk

  // 9.  The Norwegian lives in the first house.
  Norwegian.~lives  = first

  // 10. The Blend smoker has a neighbor who keeps cats.
  Blend.~smoker.hasNeighbourWho[ keeps.cats ]

  // 11. The man who smokes Blue Masters drinks beer.
  Blue_Masters.~smoker = drinks.beer

  // 12. The man who keeps horses lives next to the Dunhill smoker.
  keeps.horses.hasNeighbourWho[smoker.Dunhills]

  // 13. The German smokes Prince.
  German.~lives = smoker.Prince

  // 14. The Norwegian lives next to the blue house.
  Norwegian.~lives.hasNeighbourWho[colored.blue]

  // 15. The Blend smoker has a neighbor who drinks water.
  Blend.~smoker.hasNeighbourWho[drinks.water]
}
pred solution[ p : House ] {
   p = keeps.fish
}
run solution for 5 but exactly 5 House
```

run0

Constraint Solver: (● ProB) ○ ProB-SMT ○ Z3

States 2/3 processed, Ln 27, Col 32, Mode alloy

```
invariant_ok
House = {0,1,2,3,4}
colored_House(0) = yellow
colored_House(1) = blue
colored_House(2) = red
colored_House(3) = green
colored_House(4) = white
lives_House(0) = Norwegian
lives_House(1) = Dane
lives_House(2) = Englishman
lives_House(3) = German
lives_House(4) = Swede
drinks_House(0) = water
drinks_House(1) = tea
drinks_House(2) = milk
drinks_House(3) = coffee
drinks_House(4) = beer
smoker_House(0) = Dunhills
smoker_House(1) = Blend
```

run0

```
run0
INITIALISATION()
SETUP_CONSTANTS(House={0,1,2,3,4},colored_House={(0|->yellow
```

Figure 5.1.: Checking an Alloy command in ProB Tcl/Tk via the main menu `Verify` → `Alloy Command`. One can either use ProB's constraint solver, its SMT solver or its integration of Z3 for the verification.

translating an Alloy model.

Besides that, we provide a purely constraint-based approach for checking Alloy commands in ProB. By now, Alloy commands were verified by constraint-based checking. Here, a state satisfying all invariants but allowing to transition into a state that violates an invariant with a single B machine operation is searched as explained in Section 4.6. In ProB, constraint-based checking uses constraint solving and explicit-state model checking. This is not necessary in the case of translated Alloy models since our translation does not define state changes in B. Constraint-based checking for Alloy commands thus always resulted in initializing a B machine (checking its properties) and checking if a single machine operation is enabled. We thus provide a dedicated constraint solving routine for checking Alloy commands. For this, we create a conjunction of a B machine's properties and the precondition of a B machine operation corresponding to an Alloy command to be checked. We allow using different constraint solving backends of ProB for the verification. If a run command is satisfiable or a counterexample has been found for a check command, we load the corresponding state in ProB as can be seen in Figure 5.1 (state properties section on the bottom left), which enables the translated B machine operation.

5.2. Additional Empirical Evaluation

The Alloy community provides a public Github repository [176] containing Alloy models. For instance, many models are taken from Daniel Jackson's book on Alloy [40]. To extend the performance evaluation of our translation from Alloy to B presented in Section 4.7, we selected a subset of 25 Alloy models. We deem these models to be suited for a performance evaluation since they use different features of Alloy. We compare the runtimes of the Alloy Analyzer and ProB's constraint solver for running or checking an Alloy model's first command as can be seen in Table 5.1. For ProB, we additionally use its interface to Kodkod (ProB-Kodkod) [46], its parallel integration of Z3 (ProB-Z3), and ProB's SMT solver with (ProB-SMT) and without (ProB-Raw-SMT) an additional static syntax analysis as well as with (ProB-Sym-SMT) and without (ProB-Sym-Raw-SMT) static symmetry breaking as is presented in Chapter 6. The use of ProB's interface to Kodkod seems odd since constraint are translated back to Alloy, but we want to investigate the impact and overhead of our translation from Alloy to B. In the best case, ProB's interface to Kodkod should provide a similar performance as the Alloy Analyzer. A maximum constraint solver timeout of 5 min was used. We use the median time out of three independent runs.

It can be the case that a constraint solver could not decide for the satisfiability (unknown) but did not exceed the predefined timeout. Further, a constraint can be found to be contradictory in the current scope of ProB without being refuted in general due to the use of one or more deferred sets without a fixed size (indicated by unf.). ProB assumes a cardinality for unfixed deferred sets when loading a B machine. For the benchmarks, we use a maximum set size of 3, which can be set by preference in ProB. If finding a contradiction for a constraint using an unfixed deferred set, it cannot decide for the satisfiability since the constraint could be satisfiable when considering a larger cardinality. In this case, we also state the time needed for solving a constraint. Besides the time needed for constraint solving, we state the time needed for loading an Alloy model excluding parsing in the Alloy Analyzer (CNF) and ProB (Alloy2B). The benchmarks were run on a system with an Intel Core I7-8750H CPU (2.2GHz) and 16 GB of RAM using ProB version `1.12.2` built from commit `05f1e64c`, SICStus Prolog version `4.8.0`, Z3 version `4.12.3` built from commit `cc4ac0e6`, and the Alloy Analyzer version 6.1.0. The reason for using a pre-release version of Z3 is that we found and reported a bug in Z3 version `4.12.2` [177], where an erroneous rewriting rule implementing destructive equality resolution lead to finding solutions for unsatisfiable formulas. The bug has been fixed but a stable release of version `4.12.3` was not present at the time of writing this thesis.

First and foremost, it can be seen that the automated translation from Alloy to B is not a bottleneck for performance. Yet, loading an Alloy model in ProB can take longer than checking it, *e.g.*, as is the case for the 13th or 19th benchmark in Table 5.1. This can also be the case for the Alloy Analyzer, *e.g.*, for the 25th benchmark in Table 5.1 or when

Table 5.1.: Performance comparison of the Alloy Analyzer, ProB's constraint solver (ProB), its backend to Kodkod (ProB-Kodkod), its parallel backend to Z3 (ProB-Z3), and its different SMT solver configurations using Alloy models [176] including the time needed for loading a model in the Alloy Analyzer (CNF) and ProB (Alloy2B).

		Runtime in ms									
						ProB					
No.	Model	CNF	Alloy2B	Alloy Analyzer	ProB	Kodkod	Z3	SMT	Raw-SMT	Sym-Raw-SMT	Sym-SMT
1	com	16	193	5	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000
2	paragraph_numbering	16	29	72	> 300 000	unf. (280)	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000
3	einstein_puzzle	16	182	7	14 642	2158	> 300 000	10 977	11 058	1499	1731
4	peano	2	142	2	unf. (79)	unf. (45)	unknown	unf. (44)	unf. (44)	unf. (89)	unf. (50)
5	color_australia	2	148	3	2	73	unknown	4	4	15	25
6	address_book	3	169	4	15	84	unknown	> 300 000	> 300 000	> 300 000	> 300 000
7	address_book_3a	5	213	3	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000
8	crewalloc	4	155	3	812	867	unknown	76	80	101	105
9	birthday	2	230	2	> 300 000	unf. (41)	unf. (244)	> 300 000	> 300 000	> 300 000	> 300 000
10	abstract_memory	3	21	1	> 300 000	unf. (29)	unknown	> 300 000	> 300 000	> 300 000	> 300 000
11	cache_memory	4	170	2	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000
12	origin_tracking	5	157	6	9786	9813	unknown	> 300 000	> 300 000	> 300 000	> 300 000
13	java_types	2	236	2	120	104	unknown	191	105	120	106
14	railway	4	181	8	> 300 000	unf. (83)	unknown	> 300 000	> 300 000	> 300 000	> 300 000
15	syllogism	1	61	1	7	30	417	8	8	14	15
16	chord	18	297	13	> 300 000	> 300 000	unknown	> 300 000	> 300 000	> 300 000	> 300 000
17	chord2	34	210	250	> 300 000	> 300 000	unknown	> 300 000	> 300 000	> 300 000	> 300 000
18	chord_bug_model	3	192	2	57	128	unknown	83	93	179	164
19	dijkstra_2_process	2	177	2	9	24	unknown	19	12	16	17
20	peterson	11	244	1	unf. (37)	unf. (81)	unf. (3917)	unf. (143)	unf. (142)	unf. (2895)	unf. (2755)
21	genealogy	5	157	4	> 300 000	188	unknown	> 300 000	> 300 000	> 300 000	> 300 000
22	handshake	12	143	89	> 300 000	387	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000
23	farmer	4	174	3	> 300 000	> 300 000	unknown	> 300 000	> 300 000	> 300 000	> 300 000
24	ring_election1	7	185	6	> 300 000	> 300 000	unknown	> 300 000	> 300 000	> 300 000	> 300 000
25	overlapping_ranges	126	160	1	27	43	440	287	253	555	474
Total Solved Constraints				25	10	12	2	8	8	8	8

using integers as shown in Section 4.7. Since the time needed for solving a constraint is dominant, we deem the limitation of translating Alloy models from scratch for loading Alloy commands with different scopes as described in Section 5.1 to be neglectable.

The Alloy Analyzer is able to solve each model faster than ProB's constraint solving backends. In particular, the Alloy Analyzer is the only constraint solver that is able to solve each constraint. Unfortunately, ProB's constraint solving backends fail to solve many constraints within the predefined timeout. The models define many relations, quantifiers, and function applications. Here, the suggested translation from Alloy to B is often suboptimal and non-idiomatic for B, which corresponds to the results presented in Section 4.7. This is often not caused by non-idiomatic translation rules but the style of modeling in Alloy, which is different compared to B. For instance, B does not define a type describing the complete universe but uses concise functions and relations leading to a better performance in ProB's constraint solver. Besides that, several benchmarks use the relational closure operator for which the Alloy Analyzer has already shown benefits compared to ProB [46]. The models do not use integers for which the translation to B showed benefits in Section 4.7.

Several benchmarks cannot be decided due to the use of unfixed deferred sets (unf.). Alloy signatures are translated as deferred sets in B. If the scope of an Alloy signature is not explicitly set, the defined scope (or default scope of 3) is set to be an upper bound. ProB then assumes a fixed deferred set size when loading a B machine, *e.g.*, the upper bound. One solution would be to solve a constraint for each possible configuration of deferred set cardinalities. Yet, the amount of combinations can grow exponentially depending on the amount of deferred sets and their upper bounds. Thus, this is not a proper solution in practice.

ProB's interface to Kodkod is able to solve the most amount of constraints aside from the Alloy Analyzer. Yet, 13 constraints cannot be decided, 6 of which contain unfixed deferred sets. This shows that our translation from Alloy to B adds additional complexity that aggravates constraint solving for Kodkod. The translation from B to Kodkod does not add much overhead, which can be seen when comparing the runtimes of ProB's constraint solver and its interface to Kodkod.

ProB's integration of Z3 is only able to solve 2 constraints. In the most cases, Z3 is not able to decide for the satisfiability (unknown) or exceeds the predefined timeout. We deem quantifiers to be the reason for Z3's bad performance. The Alloy models use many quantifiers and the translation to B potentially adds more.

ProB's SMT solver is superior to the integration of Z3 but does not allow solving any constraint that cannot be decided by ProB's constraint solver. For several benchmarks, the SMT solver seems to be guided in a wrong direction leading to exceeding the predefined timeout when grounding domains in the theory solver, *i.e.*, ProB. Only for the third benchmark, the SMT solver is able to solve the constraint faster than ProB's constraint solver, and faster than ProB's backend to Kodkod when using static symmetry breaking. Yet, computing symmetry breaking predicates can also add a considerable overhead, *e.g.*, as is the case for the 20th benchmark.

Our automated translation from Alloy to B has some limitations. To the best of our knowledge, the translation rules presented in Section 4.4 cover the complete semantics of Alloy's core language version 5. However, Alloy's syntax is more flexible than the one of B and our automated translation might not cover specific cases, *i.e.*, combinations of operators. Further, we currently do not provide custom translations for the Alloy extensions providing graphs, ternary multirelations, and time macros. The above benchmark results lead to the conclusion that it is probably not worth it to translate these modules to B. The Alloy Analyzer translates constraints to SAT which can be superior to saturation-based solving as performed by ProB for constraints involving finite relations [46]. For the verification of Alloy models, the Alloy Analyzer thus remains the best choice. Yet, when it comes to solving constraints involving integers, the use of our automated translation in ProB is beneficial as shown in Section 4.7. Furthermore, one does not have to cope with integer overflows in ProB.

5.3. Translating Alloy 6 to B

One criticism of Alloy 5 is that it is difficult to model concurrent systems since there is no concept of mutable states and temporal logic by default. For instance, this was one of the main reasons why a developer team at Amazon decided to use TLA$^+$ instead of Alloy 5 [178]. To model state changes, Alloy 5 allows defining traces via a language extension. However, using this extension is a tedious task and can be error-prone. Electrum [169] is an extension of Alloy 5 that introduces operators for LTL as well as a tool, the Electrum Analyzer [179], to verify such properties. Alloy 5 was the latest release at the time of the publication of our automated translation from Alloy to classical B presented in Chapter 4.

Alloy 6 is a major new release that natively supports mutable states and LTL. For the support of LTL, the Electrum language extension was slightly adapted and integrated in the Alloy 6 core language. Further, the Alloy Analyzer was extended to include the Electrum Analyzer for verifying LTL properties. A major new keyword in Alloy 6 is `var`, which allows defining signatures and/or signature fields, *i.e.*, relations, as mutable state.

The theoretical concept of mutable states in Alloy 6 follows the one of TLA$^+$[41]: Traces are infinite executions of state changes which can be referred to as lasso traces. State changes are defined by predicates describing the next state of all mutable entities. It should be noted that the next state relation in LTL is a total function, which is the reason why all state variables have to be assigned in a state transition. In particular, this possibly requires the definition of so-called frame conditions, which assign mutable states that should not change when applying a specific predicate to their previous value. A lasso trace either terminates in a state that loops to itself or any previous state. To ensure the existence of lasso traces, *i.e.*, at least one predicate for state change can be applied in each state, a predicate that assigns all mutable states to their previous values (skip) is necessary. Hereby, all traces in Alloy 6 are guaranteed to be lasso traces since all scopes are finite in Alloy, *i.e.*, a trace has to terminate after a finite amount of state changes. For the verification of properties, the Alloy Analyzer provides bounded model checking based on SAT solving as well as backends to the model checkers NuSMV [180]

Listing 5.1: An idiomatic formal specification of the „Chameleon Puzzle“ [7] in classical B.

```
MACHINE Chameleon
SETS
 Colors = {blue,green,yellow}
VARIABLES cham
INVARIANT
  cham : Colors --> NATURAL &
  not(cham(blue) = 0 & cham(green)=0)
INITIALISATION cham := {blue |-> 13, green |-> 15,  yellow |-> 17 }
OPERATIONS
  meet(c1,c2,c3) = PRE [c1,c2,c3]:perm(Colors) &
                     c1:Colors & cham(c1)>0 & cham(c2)>0 THEN
    cham := {c1|-> cham(c1)-1, c2|-> cham(c2)-1, c3 |-> cham(c3)+2}
  END
END
```

and nuXmv [181], which also support unbounded model checking. Unfortunately, the integrations of external model checkers currently do not support integers.

In the following, we propose a translation of Alloy 6's new features to classical B while using the translation presented in Chapter 4 as a basis. It should be noted that the existing semantics of Alloy 5 have not changed but were extended by mutable states and LTL. Our presented translation from Alloy 5 to classical B is thus still valid for Alloy 6. We use the so-called „Chameleon Puzzle“ as a translation example. The puzzle first appeared in the Kvant magazine in 1985 [7], and is defined as follows:

45 chameleons live on an island, 13 of which are blue, 15 green and 17 yellow. If two chameleons of different colors meet, they change their color to the third color. Show that this doesn't make it possible for all chameleons to be yellow at some point.

We modeled the puzzle in B as can be seen in Listing 5.1, and use a specification of Peter Kriens presented in a Discourse forum for Alloy [8] as can be seen in Listing 5.2 for comparison. We adapted the Alloy model syntactically in order to emphasize the new concepts of Alloy 6. In particular, we added a dedicated predicate for skipping a state change, extracted a fact for the initialization of the mutable state, and added a fact for the actual transition system (signature step). The presented Alloy model is not optimized for performance but serves the sole purpose of an example for a translation from Alloy 6 to classical B.

The Alloy specification defines a signature for chameleons providing a relation that defines their color. This color is defined as a mutable state using the new keyword `var` since the color of a chameleon can change over time. A manual translation of the Alloy 6 specification presented in Listing 5.2 to classical B can be seen in Listing 5.3. The Alloy constructs that do not use the keywords `var`, `always` or `eventually` are translated according to our rules for translating Alloy 5 to classical B presented in Chapter 4. The relation `color_chameleon` is defined as a machine variable instead of a constant due to the new Alloy keyword `var`. The Alloy model's initialization of variables requires model

Listing 5.2: A formal specification of the „Chameleon Puzzle“ [7] in Alloy 6 using mutable state and linear temporal logic presented by Peter Kriens in a Discourse forum for Alloy [8]. We adapted the model syntactically to emphasize the new concepts of Alloy 6.

```
 1  enum Color { blue, green, yellow }
 2  sig Chameleon { var color: Color }
 3  fact init {
 4    #Chameleon = 45
 5    #color.blue = 13
 6    #color.green = 15
 7    #color.yellow = 17
 8  }
 9  pred meet[a, b : Chameleon] {
10    a.color != b.color and
11    let thirdColor = Color - a.color - b.color |
12      color' = color ++ (a->thirdColor + b->thirdColor)
13  }
14  pred skip { color' = color }
15  fact step {
16    always ((some a, b : Chameleon | meet[a,b]) or skip)
17  }
18  run { eventually Chameleon.color = yellow }
19    for 45 but 7 int, 361 steps
```

finding which can be achieved in B using the `ANY` substitution as can be seen in line 15 to 18 of Listing 5.3.

Alloy predicates that change a mutable state can be translated as B machine operations. For instance, the Alloy predicate `meet` defined in line 9 to 13 of Listing 5.2 can be translated as can be seen in line 20 to 25 of Listing 5.3. For this, the Alloy predicate's guard has to be extracted to create the B machine operation's precondition. Alloy's next state assignment can be translated using a single assignment substitution in B. Several assignments can be translated using B's sequential or parallel substitution assignment. It should be noted that the assignment of a mutable state might be nested in Alloy as can be seen in line 11 and 12 of Listing 5.2. In contrast to this, B machine variables are explicitly assigned a value using a B expression as can be seen in line 12 of Listing 5.1. An automated translation thus possibly requires a preprocessing of Alloy predicates extracting each assignment, *e.g.*, by rewriting to a normal form. Alternatively, B's `ANY` substitution can be used to translate predicates without any preprocessing. For instance, the predicate `skip` defined in line 14 of Listing 5.2 can be translated as the B machine operation op = ANY color2 WHERE color2 = color THEN color := color2 END. Therefor, only the next state assignment color$'$ has to be renamed to a new variable, *e.g.*, color2, while the Alloy predicate can be translated as is without any preprocessing. Yet, this translation is less idiomatic, and a single assignment substitution in B would be more readable and more performant.

A difference between state assignments in Alloy 6 and B is that Alloy allows assigning

Listing 5.3: A manual translation of the Alloy 6 specification presented in Listing 5.2 to classical B mainly following our rules for translating Alloy 5 to classical B presented in Chapter 4.

```
 1  MACHINE chameleon_alloy_to_b
 2  SETS Color = {yellow,green, blue}; Chameleon
 3  PROPERTIES
 4    card(Chameleon) = 45 & card(Color) = 3 &
 5    {yellow}/\{green} = {} & {yellow}/\{blue} = {} &
 6    {green}/\{blue} = {}
 7  VARIABLES
 8    color_Chameleon
 9  DEFINITIONS
10    ASSERT_LTL  == "F{color_Chameleon[Chameleon] = {yellow}}"
11  INVARIANT
12    color_Chameleon : Chameleon --> Color &
13    not (color_Chameleon[Chameleon] = {yellow})
14  INITIALISATION
15    ANY c
16        WHERE c : Chameleon --> Color
17            & card((c~)[{blue}]) = 13 & card((c~)[{green}]) = 15 &
                  card((c~)[{yellow}]) = 17
18        THEN color_Chameleon := c END
19  OPERATIONS
20    meet(a,b) =
21      PRE a : Chameleon & b : Chameleon & color_Chameleon[{a}] /=
            color_Chameleon[{b}] THEN
22          color_Chameleon := color_Chameleon <+
23            (LET thirdColor BE  thirdColor = (Color -
                  color_Chameleon[{a}] - color_Chameleon[{b}])
24              IN {a} * thirdColor \/ {b} * thirdColor END)
25      END
26  END
```

any state while B only allows assigning the next state in a single machine operation. Therefore, Alloy predicates assigning any previous or future state other than the next state cannot be translated to B.

A main difference of a new translation from Alloy 6 to B compared to Alloy 5 is the idiomatic use of state changes using B machine operations. This enables the use of explicit-state, symbolic, and LTL model checking in ProB.

B and ProB provide full support for LTL as is supported by Alloy 6. The Alloy model's `run` command defines an invariant using the keyword `eventually`. This property can be translated as an LTL formula in ProB using a definition as can be seen in line 10 of Listing 5.3. However, it can be more efficient to negate the property and translate it as a B machine invariant as can be seen in line 13 of Listing 5.3. This allows for explicit-state model checking instead of LTL model checking. Frame conditions do not have to be translated to B since a B operation skips all variables which are not explicitly assigned by default.

It should be noted that the proposed translation is not idiomatic in B. For instance, it is more efficient to translate the chameleon population's relation using a total function relating an integer to each color as can be seen in line 6 of Listing 5.1. However, such a model specific performance improvement can probably not be deduced automatically from the Alloy model. The implementation of a general preprocessing extracting guards and assignments from all kinds of Alloy predicates is a tedious and error-prone task, but should be possible in practice. The alternative translation using B's `ANY` substitution as described above is straightforward, but requires some effort to be fully integrated in our automated translation. An implementation of the proposed translation from Alloy 6 to classical B in ProB is up to future work.

For the different chameleon puzzles, the runtimes of the Alloy Analyzer and ProB differ a lot. We used the same system settings as in Section 4.7. ProB is able to solve the idiomatic B encoding presented in Listing 5.1 using explicit-state invariant checking in 236 ms. The B model contains 361 states. The Alloy Analyzer fails to check the Alloy model within 10 min (23 steps of BMC using Minisat, 21 steps using NuSMV, and 15 steps using nuXmv were checked). Yet, the manually translated model presented in Listing 5.3 can also not be solved within 10 min by ProB using explicit-state, symbolic or LTL model checking. We noted that the Alloy model defines a much larger state space compared to the idiomatic B model containing many symmetries. For instance, the Alloy model defines many different initializations for the signature of chameleons since each chameleon (total: 45) is mapped to a color. Here, two chameleons can interchange their colors leading to a different instantiation but still satisfying the initialization, *i.e.*, the fact `init` in Listing 5.2. The idiomatic B model, on the other hand, defines a single instantiation since it maps each color (total: 3) to a natural number. This again shows that the approaches to solve a problem are often different in Alloy and B leading to drastic performance differences in model finding and checking. For the translated model, the performance of ProB's explicit-state model checker can be improved by using hash-based symmetry breaking. Nevertheless, the model can still not be checked within 10 min.

For comparison, we created an alternative Alloy model of the chameleon puzzle defining a total function similar to the one defined in Listing 5.1, which can be seen in Listing A.1. Although the model only defines a single instantiation, the Alloy Analyzer was only able to check the model for 24 steps within 10 min using the Minisat backend. NuSMV and nuXmv cannot be used since integers are currently not supported by the Alloy Analyzer's integration. In Listing A.2, we present a manual translation of the Alloy model to B mainly following our translation rules for Alloy 5. Note that we have to use B's *MU* operator to receive an integer from a singleton set of integers. For this, the external predicate file `CHOOSE.def` has to be loaded. The model defines 362 states and can be checked by ProB's explicit-state model checker in around 216 ms and in around 205 ms using its LTL model checker. This shows that a translation from Alloy 6 to B for model checking with ProB can be beneficial compared to using the Alloy Analyzer's backends.

Part III.

New Techniques for Constraint Programming in B

6. SMT Solving for the Validation of B and Event-B Models

Joshua Schmidt and Michael Leuschel

Abstract ProB provides a constraint solver for the B-Method written in Prolog and can make use of different backends based on SAT and SMT solving. One such backend translates B and Event-B operators to SMT-LIB using the Z3 solver. This translation uses quantifiers to axiomatize some operators, which are not well-handled by Z3. Several relational constraints such as the transitive closure are not supported by this translation.

In this article, we substantially improve the translation to SMT-LIB by employing a more constructive rather than axiomatized style using Z3's lambda function. Thereby, we are able both to translate more B and Event-B operators to SMT-LIB and improve the overall performance. We further extend ProB's interface to Z3 to run different solver configurations in parallel.

In addition, we present a direct implementation of SMT solving in Prolog using ProB's constraint solver as a theory solver. We hereby aim to combine the strengths of conflict-driven clause learning for identifying contradictions with ProB's constraint solver for finding solutions. We deem this implementation to be worthwhile since ProB's constraint solver is tailored toward solving B and Event-B constraints, and we herewith avoid the dependency on an external SMT solver.

Empirical results show that the new integration of Z3 has improved performance of constraint solving and enables to solve several constraints which cannot be solved by ProB's constraint solver. Furthermore, the direct implementation of SMT solving in ProB shows benefits compared to ProB's constraint solver and the integration of Z3.

6.1. Introduction

The B-Method [25] is a correct-by-construction approach for software development based on formal refinement. Its foundation is an expressive formal language rooted in set-theory, integer arithmetic, and first-order logic. The B language supports higher order data types such as functions or arbitrarily nested relations, and is nowadays referred to as classical B. Event-B [26] is its successor which, *e.g.*, puts the focus on systems modeling by extending refinement. In this article, we only refer to the B language which covers predicates and expressions that are present in classical B and Event-B. In particular, there is no need to differentiate between classical B and Event-B in the context of constraint solving.

ProB [28, 29] is an animator, model checker, and constraint solver for the B-Method. The constraint solver is used for many tasks and is the foundation of the ProB tool. For instance, the constraint solver has to compute the effect of state changes during animation, find counterexamples to proof obligations during disproving or solve constraints for symbolic model checking or program synthesis. One key feature of ProB is that it computes all solutions of a constraint. For instance, this is important for a complete state-space exploration during model checking or when computing set comprehensions. This search is performed using chronological backtracking. A set comprehension in B is a quantified formula describing the elements of a set using a constraint that has to be satisfied by each element. The core of ProB is implemented in SICStus Prolog [81] using its library for constraint solving over the finite domain integers (CLP(FD)) [30] and other features such as coroutines for deterministic propagation and constraint reification. Coroutines in Prolog can be used to suspend computations until a certain condition is met. Constraint logic programming (CLP) generally uses algorithms to reduce the domain of variables when new constraints are posted and identifies a contradiction if a domain becomes empty. After the phase of domain reduction, solutions can be found by enumerating the remaining domains (aka grounding). ProB's constraint solver handles integer overflow by custom implementations to overcome the limited range of CLP(FD) and deal with unbounded domains. It also supports symbolic representations for infinite values. Of course, the ProB constraint solver might fail to solve constraints over unbounded domains, *e.g.*, due to a timeout or a virtual timeout, which is the case when ProB detects that a domain cannot be enumerated exhaustively and all solutions are required.

Other prominent constraint solvers such as Z3 [47] implement a conflict-driven clause learning modulo theories (CDCL(T)) architecture, which combines SAT and theory solving called Satisfiability Modulo Theories (SMT). In contrast to CLP(FD) and ProB's constraint solver, SMT solvers are able to learn from contradictions [87, 88] and possibly leave dead-end parts of the search tree earlier and more aggressively by applying backjumping instead of chronological backtracking. The SMT-LIB standard [100, 182] defines the input language for SMT solvers.

In prior work, Krings and Leuschel presented a high-level translation from B to SMT-LIB to integrate the Z3 SMT solver into ProB [48]. The authors have shown that, on the one hand, Z3 can be superior to ProB when disproving formulas, especially over unbounded domains. On the other hand, Z3 often fails to find solutions for satisfiable constraints involving relations or set comprehensions. The translation uses existing operators in SMT-LIB or Z3 wherever possible [48]. Unfortunately, SMT-LIB does not have native support for set comprehensions, which are frequently used in the B language. The authors thus suggested translating B set comprehensions using a universal quantification which constrains all the members of a set variable. Unfortunately, this axiomatic translation often leads to complex constraints for which Z3 fails to find a solution. Several other B operators are also not supported by the SMT-LIB standard such as the relational composition, iteration and closure, or quantified union $\bigcup_{x\in S}$ and intersection $\bigcap_{x\in S}$. As their axiomatic translation to SMT-LIB using universal quantifiers is complex, these operators were not supported in [48].

While trying to improve Z3's performance and analyzing satisfiable B constraints which can be solved by ProB's constraint solver but not by Z3, we found an alternate translation using lambda functions instead of quantifiers. It turned out that this alternate approach can considerably improve performance. Take for example the (right) relational override operator $r \Leftarrow s$, which joins two relations by adding tuples of s to r. A tuple in r is replaced by a tuple in s if both tuples have the same first element. For instance, a simple satisfiable constraint is given by $f = \{1 \mapsto 2\} \wedge g = f \Leftarrow \{2 \mapsto 3\}$, which has the solution $g = \{1 \mapsto 2, 2 \mapsto 3\}$. With the axiomatic translation Z3 is not able to solve this constraint while Z3 can solve it when encoding the override operator using a lambda function. Z3 supports such lambda functions, even though they are not part of the latest SMT-LIB standard 2.6. Note that from version 3.0 lambda functions will be part of the SMT-LIB standard as well. Nevertheless, we observed that the axiomatic translation from B to SMT-LIB has benefits. In order to achieve the best performance, we decided to run several configurations of the Z3 solver with both translations in parallel.

While our empirical evaluation in Section 6.7 shows that the new integration of Z3 improves performance and coverage compared to the prior integration [48], it still has limitations. For example, it cannot deal well with constraints involving set cardinalities, which are frequently used in B. B sets are translated as characteristic functions using Z3's array theory [183]. This array theory allows defining nested and infinite sets, which is important in the context of the B language. Unfortunately, Z3 does not provide a cardinality constraint. B's set cardinality is thus translated as a total bijection, which itself is rewritten using universal quantification. For instance, the B predicate $c = \mathrm{card}(s)$ is encoded as an existentially quantified total bijection $\exists t.(t \in s \rightarrowtail\!\!\!\!\twoheadrightarrow 1 \,..\, c) \wedge c \geq 0$ ($\rightarrowtail\!\!\!\!\twoheadrightarrow$ is the symbol for a total bijection in B) [48]. A simple constraint for which the integration of Z3 spends a disproportional amount of time to find a solution is $x \in \mathbb{P}(\mathbb{Z}) \wedge \mathrm{card}(x) > 10$. The reason for this is that Z3 often has trouble solving formulas that contain many quantifiers. Other examples of constraints which have no direct counterpart in SMT-LIB and exhibit similar performance issues are the power set or maximum and minimum of a set of integers.

In this article, we thus also investigated a third approach to SMT solving. We additionally implemented state-of-the-art SMT solving techniques directly in ProB to tightly connect ProB's constraint solving core for finding solutions with a CDCL(T)-based learning scheme to prune the search space early and improve the identification of contradictions. The ProB constraint solver is particularly strong at solving set cardinalities (for finite sets) which are encoded using bit vectors and coroutines while a constraint of CLP(FD) that computes the sum of a list of integers is used to compute the actual cardinality from a corresponding bit vector. Our expectations are thus that the use of ProB's constraint solver as the theory solver for an SMT solver enables to overcome the aforementioned shortcomings of the integration of Z3. Furthermore, our implementation can be of interest in the SMT community since the B language entails well-definedness conditions which are not considered in common SMT solvers.

The presented constraint solving backends are integrated into ProB which is available at:

`https://prob.hhu.de`

This article is the extended version of our original submission to the FMICS conference [49]. We extend the former work in different aspects by providing

– a brief introduction to B and SMT-LIB (Section 6.2),

– a more formal description of the translation from B to SMT-LIB by providing constructive definitions for sorts in SMT-LIB (Section 6.4.1),

– a decomposition of B constraints into independent components to investigate the impact on constraint solving for the integration of Z3 (Section 6.4.3),

– a direct implementation of SMT solving for B and Event-B in Prolog using ProB's constraint solver as a theory solver (Section 6.5),

– an integration of an additional theory solver for integer difference logic alongside ProB's constraint solver (Section 6.6),

– and an extended empirical evaluation including a justification for the decision of running different Z3 solvers in parallel, more benchmarks from bounded model checking, and benchmarks from constraint-based proofs of inductive invariants as well as for deadlock freedom (Section 6.7).

6.2. Background

In the following, we give a brief introduction to the B formal specification language and the SMT-LIB language. We focus on the parts of the languages that we use for our translation from B to SMT-LIB as well as our empirical evaluation.

6.2.1. Primer on B

The formal specification language B [25] is rooted in set-theory, integer arithmetic, and first-order logic and follows the correct-by-construction approach. B has been developed for the specification and design of software systems. Specific properties can be proven mathematically using theorem provers, *e.g.*, using AtelierB [121], or be checked using a model checker such as ProB [27–29]. The B language supports unbounded domains and higher order data types such as arbitrarily nested relations. Nowadays, the B language is referred to as classical B. Event-B [26] is the successor of classical B which improves the language in several aspects and puts the focus on systems modeling by extending refinement. Note again that in this article we only refer to the B language for the sake of simplicity, which covers predicates and expressions that are present in classical B and Event-B.

The development in classical B and Event-B is incremental starting with a high-level abstract specification which is successively refined and decomposed to increase the maintainability and ease the specification of complex models. A model thus consists of a collection of so-called machines. All refinement steps are linked by proof obligations which have to be discharged in order to ensure that the refinement does not diverge from the prior specification. A machine consists of variable and type definitions as well as initial values. A state is defined by the current values of the machine variables. One can specify transitions between states by defining machine operations (called events in Event-B) that compute successor states including all variables. An operation (or event) can have a precondition (called guard in Event-B), allowing or prohibiting execution based on the current state. Certain behavior can be ensured by defining machine invariants, which are safety properties that have to hold in every reachable state. The correctness of a formal model thus refers to its specified invariants.

In addition to native B types such as $\mathbb{Z}$ or $\mathbb{B}$, one can provide user-defined sets. These sets can be defined by a finite enumeration of distinct elements (enumerated sets) or left open (deferred sets). For instance, $S = \{s\}$ defines an element s of type S, which both can be accessed by name within the machine. Deferred sets are assumed to be non-empty during proof and also finite for animation in ProB.

A set comprehension in B is a quantified formula constraining the elements of a set. If quantifying two variables, the elements of a set comprehension are tuples. For instance, the equations $\{x \mid x \in 1..2\} = \{1, 2\}$ and $\{x, y \mid x \in 1..2 \land y = 0\} = \{(1 \mapsto 0), (2 \mapsto 0)\}$ are true in B. There is no limit to the amount of quantified variables other than that it is a finite number. For instance, the elements of a set corresponding to a set comprehension quantifying three variables are triples.

B is statically and strongly typed while ProB further executes runtime checks to ensure well-definedness. For instance, a function application $f(1)$ is well-defined if 1 is an element of the domain of the function f. Other exemplary B operators that entail a well-definedness condition are the minimum and maximum of a set of integers which has to be non-empty or integer division. Type domains can be unbounded, possibly resulting in a model with an infinite state space. While B has a strict type system, there is no distinction between sets of pairs, relations, functions, and sequences. For instance, the sequence $[-1]$ is the function $\{1 \mapsto -1\}$, which is also a relation, which in turn is a set of pairs. It is thus possible that sequences interact with sets of pairs resulting in a set of pairs which is not a sequence anymore. For instance, the equation $[-1] \cup \{3 \mapsto 2\} = \{1 \mapsto -1, 3 \mapsto 2\}$ is true in B, but the right-hand side of the equation is not a well-defined sequence since the domain is not enumerated coherently.

6.2.2. Primer on SMT-LIB

The SMT-LIB initiative [184] defines a standard input language for common SMT solvers called SMT-LIB [182] as well as a set of benchmarks for different background theories.

The SMT-LIB language is based on many-sorted first-order logic with equality [182] that allows defining sorts, *i.e.*, types, and sorted terms. Its syntax is defined in a Lisp style. Exemplary sorts are the integers (`Int`), Boolean (`Bool`) or arrays (`Array`). For instance, (`Array Int Bool`) defines an array sort that maps integers to Boolean. SMT-LIB allows defining function symbols that are associated with a rank. The rank of a function symbol defines the sorts of the inputs as well as the output. In general, a function symbol with rank $\sigma_1 \cdots \sigma_n \sigma$ has n inputs of sort $\sigma_1 \cdots \sigma_n$ and one output of sort σ [182]. One is able to introduce uninterpreted functions using the `declare-fun` keyword, or interpreted functions using `define-fun`. For instance, (`declare-fun` x () `Int`) declares an uninterpreted function x that returns an integer, *i.e.*, x is an integer variable, and (`define-fun` f ((x `Int`) (y `Int`)) `Int` ($+$ x y)) defines a function f that adds two integers. All functions in SMT-LIB are total which entails that every function call is well-defined. In fact, there is no concept of well-definedness in SMT-LIB. For instance, the equality (`=` (`div` x 0) (`div` x 0)) is true for an arbitrary integer symbol x although the division by zero is not defined in mathematics. It is possible to define recursive functions using the `define-fun-rec` keyword. Furthermore, the SMT-LIB language allows defining algebraic data types along arbitrary function declarations that have to hold for a data type using the `declare-datatype` keyword. For instance, a tuple type that provides two projection functions to access its first and second element can be declared as follows:

(`declare-datatype` Tuple (`par` (X Y) ((tuple (first X) (second Y))))

Formulas in SMT-LIB are terms of sort `Bool` that can be asserted to hold using the `assert` keyword. Such formulas can reason over function symbols that have been declared beforehand. A dedicated SMT solver holds a stack of assertions that consists of formulas, declarations, and definitions. Besides reasoning over globally declared function symbols within an SMT formula, it is possible to reason over local function symbols using different kinds of binders such as `let`, `exists` or `forall`. The scoping is defined to refer to the last declaration of a function symbol. For instance, the following example shows a simple SMT-LIB model that defines a global integer symbol x as well as an existential quantifier that reasons over a local integer symbol x:

```
(declare-fun x () Int)
(assert (= x 1))
(assert (exists ((x Int)) (> x 1)))
(check-sat)
(get-model)
```

The keywords `check-sat` and `get-model` instruct a solver to check for the satisfiability of all assertions and return a model for all global function symbols if the assertions are satisfiable. For the above example, we receive a model stating that x is equal to 1 which is represented in SMT-LIB as well. In particular, we receive a list of function definitions for global function symbols ((`define-fun` x () `Int` 1)) as a model.

6.3. Former Z3 Integration

In the following we revise the workflow of the former integration of Z3 in ProB as well as the high-level translation from B to SMT-LIB presented by Krings and Leuschel [48].

6.3.1. High-Level Translation

The former high-level translation [48] uses corresponding operators of SMT-LIB wherever possible. B sets are translated as characteristic functions in SMT-LIB mapping set elements to either true or false as defined by Z3's array theory [183]. This theory allows defining nested and infinite sets. For instance, for the predicate $x \subseteq \mathbb{P}(\mathbb{Z})$, the variable x is defined as a characteristic function of sort (`Array` (`Array Int Bool`) `Bool`). All logical B predicates ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$), all integer expressions except for division ($+$, $-$, mod, $**$, $\geq$, $>$, $<$, $\leq$), simple set expressions ($\in$, $\subset$, $\subseteq$, $\cup$, $\cap$, $-$), and quantifiers ($\forall$, $\exists$) are supported by SMT-LIB and can be translated with equivalent operators.

Since the B language does not distinguish between sets of pairs, relations, functions, and sequences, all of these data types are translated as sets of pairs as is defined in B. Unfortunately, this prevents us from using certain features of Z3 which would probably be more efficient. For instance, B sequences could be directly translated as arrays in SMT-LIB instead of rewriting them to sets of pairs beforehand. Yet, this translation to arrays could only be performed if sequences only interoperate with other sequences since B set operators can be called on sequences yielding a relation which is not a sequence anymore.

Another difference between B and SMT-LIB is that B implements a concept of well-definedness [185] which is not present in SMT-LIB. Axioms for well-definedness ensure that certain operators are only applied when they make sense and that the proof rules of classical two-valued logic can be applied. For instance, B prohibits division by zero while in SMT-LIB integer division is a total function, *e.g.*, `(= (div 1 0) (div 1 0))` is true in SMT-LIB and not well-defined in B. The same applies if the divisor is a variable that can be assigned to 0. Another difference is that B's integer division rounds toward zero while SMT-LIB follows Boute's Euclidean definition [155]. Boute defined division as rounding to positive infinity when the divisor is negative and rounding to negative infinity otherwise. B's integer division a/b is thus translated to SMT-LIB as follows:

```
(ite (or (= (rem a b) 0) (> a 0)) (div a b)
     (ite (> b 0) (+ (div a b) 1) (- (div a b) 1)))
```

For the well-definedness of a/b, we assert that b is not equal to zero. Other operators with a well-definedness condition are, *e.g.*, B's function application or minimum and maximum of a set of integers. For the translation of these operators, additional well-definedness conditions are added.

A frequently used construct in B is set comprehension which has no direct counterpart in SMT-LIB. Set comprehensions are thus rewritten as axiomatized formulas using quantifiers [48]. In particular, an existentially quantified variable is defined for each quantified variable of a set comprehension. For instance, the set comprehension $\{x \mid x \in \mathbb{Z} \wedge x > 0\}$ is encoded as a fresh existential integer set variable tmp alongside the axiom $\forall v.(v \in \text{tmp} \Leftrightarrow v > 0)$ [48].

Several B set operators which cannot be directly translated to SMT-LIB such as the domain of a relation are rewritten as set comprehensions. For instance, $\text{dom}(r)$ is rewritten as $\{x \mid \exists y.(x \mapsto y \in r)\}$. Yet, the operators $\min(s)$, $\max(s)$, and $\text{card}(s)$

cannot be rewritten as set comprehensions. These operators are instead translated as identifiers which are axiomatized accordingly. For instance, the minimum of an integer set $\min(s)$ is replaced by an identifier m which is axiomatized by $\forall x.(x : s \Rightarrow m \leq x) \wedge \exists x.(x \in s \wedge m = x)$. The maximum of an integer set is encoded analogously.

Computing the cardinality of a set in SMT-LIB is expensive due to the employed encoding of sets as characteristic functions. A total bijection has to be computed mapping the elements of a set to a coherent interval of indices starting at 1 while the largest index corresponds to the cardinality of the set. For instance, the B predicate $c = \text{card}(s)$ is encoded as an existentially quantified total bijection $\exists t.(t \in s \rightarrowtail\!\!\!\!\twoheadrightarrow 1 \,..\, c) \wedge c \geq 0$ [48]. It has to be ensured that the variable c is greater than or equal to zero since the empty set could have any negative cardinality otherwise. The authors refer to such rewritten predicates as normalized B. A normalized predicate is then passed to the actual translator to SMT-LIB.

B supports user-defined types in the form of deferred sets and enumerated sets as described in Section 6.2.1. Such B types are translated to corresponding sorts in SMT-LIB. Deferred sets are not limited in size but assumed to be non-empty for proof and also finite for animation in ProB. For enumerated sets, the actual instances are given which are defined as function symbols in SMT-LIB and axiomatized to be distinct.

The authors point out that several operators such as the relational closure or the general union $\bigcup_{x \in S}$ and intersection $\bigcap_{x \in S}$ of a nested set S cannot be translated effectively to SMT-LIB using quantifiers [48], which is why they are not supported.

6.3.2. Workflow

The former integration [48] of Z3 in ProB provides two interfaces. First, full B predicates can be translated to SMT-LIB and be solved by Z3. As described in Section 6.3.1, several B operators are not supported by the former translation to SMT-LIB and thus, are filtered before the translation. In particular, all top-level conjuncts that contain an unsupported operator are removed. If a predicate uses unsupported operators, the result of Z3 can thus only be used if a contradiction has been found. For instance, consider the predicate P ∧ Q where P is any unsatisfiable predicate and Q contains any predicate that is not supported by the former integration of Z3, *e.g.*, the quantified union. We remove the top-level conjunct Q and only translate the contradicting predicate P to SMT-LIB. The unsatisfiability of the overall formula is identified if Z3 is able to identify the contradiction in P. Yet, if P is satisfiable, we cannot use the partial model since Q has not been evaluated.

The second interface intertwines Z3 with ProB's constraint solver by setting up constraints simultaneously and sharing intermediate results. All clauses learned by Z3 are fed to ProB's constraint solver as well, which lifts ProB's search capabilities from backtracking to backjumping. The call to Z3 is delayed after the deterministic propagation phase of ProB [48] since ProB's constraint solver generally shows better performance in model finding over B constraints than Z3. During this phase, ProB might infer new constraints which are then added to Z3.

6.4. New Z3 Integration

In the following we describe the new high-level translation from B to SMT-LIB as supported by Z3 as well as the new parallel solver integration.

6.4.1. High-Level Translation

For the formal description of the translation, we provide two semantic functions for B expressions representing values and predicates representing a truth value. In particular, $E[\![e]\!]i$ is the Z3 encoding of the B expression e, and $P[\![p]\!]i$ is the Z3 encoding of the B predicate p. The variable i is an environment which stores specific information of a translation. The following example shows a series of rewriting steps, applying the rules of $E[\![e]\!]i$ and $P[\![e]\!]i$ (shown further below):

$$\begin{aligned} P[\![x > y \wedge y > x]\!]i \ &\widehat{=}\ (\texttt{and}\ E[\![x > y]\!]i\ E[\![y > x]\!]i) \\ &\widehat{=}\ (\texttt{and}\ (\texttt{>}\ E[\![x]\!]i\ E[\![y]\!]i) \\ &\qquad\qquad (\texttt{>}\ E[\![y]\!]i\ E[\![x]\!]i)) \\ &\widehat{=}\ (\texttt{and}\ (\texttt{>}\ x\ y)\ (\texttt{>}\ y\ x)) \end{aligned}$$

Global B variables such as $E[\![x]\!]i$ are translated as functions using the same name. That means, $E[\![x]\!]i\ \widehat{=}\ x$ but as a side effect a global function symbol for the variable x has been introduced in SMT-LIB. Locally quantified B variables are translated in the same way but do not introduce a global function symbol in SMT-LIB.

The environment i contains a list of translated Z3 expressions and function declarations, a mapping from B expressions to B types Ψ_i, *e.g.*, $\Psi_i(-1) = \mathbb{Z}$, and a mapping from Z3 expressions to Z3 sorts Φ_i, *e.g.*, $\Phi_i(-1) = \texttt{Int}$. For sets, *i.e.*, arrays in SMT-LIB, we introduce the operator $\mathbb{P}^{-1}$ yielding the type of the elements of a set. For instance, $\mathbb{P}^{-1}((\texttt{Array Int Bool})) = \texttt{Int}$ for a basic set of integers, and $\mathbb{P}^{-1}((\texttt{Array (Array Int Bool) Bool})) = (\texttt{Array Int Bool})$ for a nested set of integers.

Furthermore, the environment stores a mapping from B tuple types to Z3 functions Ω_i, which allow accessing the elements of tuples in SMT-LIB. For instance, $\Omega_i(\Psi_i(1 \mapsto 2)) = [\text{first}_{i,\Phi_i(E[\![1\mapsto 2]\!]i)}, \text{second}_{i,\Phi_i(E[\![1\mapsto 2]\!]i)}]$, where $\text{first}_{i,\Phi_i(E[\![1\mapsto 2]\!]i)}$ and $\text{second}_{i,\Phi_i(E[\![1\mapsto 2]\!]i)}$ are the projection functions of the Z3 tuple sort corresponding to the B type $\Psi_i(1 \mapsto 2)$. For better readability, we use the abbreviations $\texttt{first}_{i,\Phi_i(c)}$ and $\texttt{second}_{i,\Phi_i(c)}$ with $c = E[\![x \mapsto y]\!]i$ for the projection functions of the Z3 tuple sort that has been introduced for the B type $\Psi_i(x \mapsto y)$. For instance, $\texttt{first}_{i,\Phi_i(E[\![1\mapsto 2]\!]i)} = \Omega_i(\Psi_i(1 \mapsto 2)).\text{at}(0)$ and $\texttt{second}_{i,\Phi_i(E[\![1\mapsto 2]\!]i)} = \Omega_i(\Psi_i(1 \mapsto 2)).\text{at}(1)$. Furthermore, we drop the type information of the projection functions if their argument is given. For instance, $(\texttt{first}_{i,\Phi_i(c)}\ c) = (\texttt{first}_i\ c)$ with $c = E[\![x \mapsto y]\!]i$. We refer to the Z3 sort of a tuple in SMT-LIB using Θ_i, *e.g.*, $\Theta_i(\texttt{Int}, \texttt{Int})$ corresponds to the Z3 sort of a tuple of integers. The types of the elements of a tuple can be accessed using θ_1 and θ_2, *e.g.*, $\theta_1(\Theta_i(\texttt{Int}, \texttt{Bool})) = \texttt{Int}$ and $\theta_2(\Theta_i(\texttt{Int}, \texttt{Bool})) = \texttt{Bool}$. Last but not least, we allow calling the semantic functions on partially defined B operators, *e.g.*, $E[\![\text{dom}(S)]\!]i\ \ \widehat{=}\ \ (E[\![\text{dom}]\!]i\ E[\![S]\!]i)$.

Tuples

In B, tuples are encoded as nested pairs. Thus, several encodings of tuples exist and the modeler has to know which one is being used. For instance, a triple can be represented as either $(x \mapsto (y \mapsto z))$ or $((x \mapsto y) \mapsto z)$. We use the first left-associative encoding and introduce a unique Z3 sort for each tuple type occurring in a B predicate when translating to SMT-LIB. For this, we declare a new data type using `declare-datatype` as described in Section 6.2.2. B tuples are then translated using their corresponding Z3 sort's constructor which is defined as follows:

$$E[\![(x_1,\dots,x_n)]\!]i \ \ \widehat{=} \ \ (\texttt{tuple}_{i,\Phi_i(E[\![x_1]\!]i),\dots,\Phi_i(E[\![x_n]\!]i)} E[\![x_1]\!]i \ \dots \ E[\![x_n]\!]i)$$

The Z3 function $\texttt{tuple}_{i,\Phi_i(E[\![x_1]\!]i),\dots,\Phi_i(E[\![x_n]\!]i)}$ is the constructor of the Z3 tuple sort which has been introduced for B tuples of type $\Psi_i(x_1 \times \dots \times x_n)$, where $n \in \mathbb{N}$. For the sake of readability, we drop the type information of the tuple constructor since the types are implicitly given by the constructor's arguments. In particular, we use
$(\texttt{tuple}_i \ E[\![x_1]\!]i \ \dots \ E[\![x_n]\!]i)$.

B provides two projection functions to access the elements of a tuple which are translated as follows:

$$E[\![\mathrm{prj}_1(\Psi_i(x),\Psi_i(y))(x \mapsto y)]\!]i \ \ \widehat{=} \quad (\texttt{first}_i \ E[\![x \mapsto y]\!]i)$$

$$E[\![\mathrm{prj}_2(\Psi_i(x),\Psi_i(y))(x \mapsto y)]\!]i \ \ \widehat{=} \quad (\texttt{second}_i \ E[\![x \mapsto y]\!]i)$$

Set Notation

As described in Section 6.3.2, the former high-level translation rewrites many set operators to B set comprehensions since they are not directly supported by SMT-LIB. Set comprehensions themselves are then rewritten using B quantifiers which can be directly translated to SMT-LIB. However, using many quantifiers can lead to unnecessarily complex constraints for which Z3 is not able to find a model. Fortunately, Z3 provides lambda functions which allow defining a set of variables that are constrained by an expression. In general, a lambda function (`lambda` sorts body) in Z3 returns an expression of the sort (`Array` sorts range) where range is the sort of body. For instance, the lambda function (`lambda` ((x `Int`)) (`and` (`>=` x 0) (`<=` x 2))) describes the set of integers $\{0,1,2\}$ as an array that maps integers to either true or false, *i.e.*, the output has the sort (`Array Int Bool`). For our translations, we consistently use such lambda functions that constrain a single variable by a Boolean expression.

First and foremost, we suggest translating B set comprehensions using Z3's lambda function which we define as follows:

$$E[\![\{x \mid p\}]\!]i \ \widehat{=} \ (\texttt{lambda} \ ((E[\![x]\!]i \ \Phi_i(E[\![x]\!]i))) \ P[\![p]\!]i)$$

$$E[\![\{x_1,\dots,x_n \mid p\}]\!]i \ \widehat{=}$$
$$(\texttt{lambda} \ ((c \ \Phi_i(E[\![x_1 \times \dots \times x_n]\!]i))) \ R^*_{i,c,x_1,\dots,x_n}(P[\![p]\!]i)))$$

The first case is a special case for a B set comprehension with a singleton result variable since no tuple has to be created here. In the second case, R^* is a semantic function that replaces the translated variables $x_1, \ldots, x_n$ in the predicate p corresponding to the position in the tuple c. For instance, as can be seen in the following example, where x is translated as ($\texttt{first}_i$ c), y as ($\texttt{first}_{i,\Phi_i(E[\![y \times z]\!]i)}$ ($\texttt{second}_i$ c)), and z as ($\texttt{second}_{i,\Phi_i(E[\![y \times z]\!]i)}$ ($\texttt{second}_i$ c)):

$$
\begin{array}{l}
\{x, y, z \mid x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge z \in \mathbb{N}\} \;\widehat{=} \\
\quad (\texttt{lambda}\ ((c\ \Phi_i(E[\![x \times y \times z]\!]i))) \\
\quad\quad (\texttt{and}\ (\texttt{>=}\ (\texttt{first}_i\ c)\ 0) \\
\quad\quad\quad\quad (\texttt{>=}\ (\texttt{first}_{i,\Phi_i(E[\![y \times z]\!]i)}\ (\texttt{second}_i\ c))\ 0) \\
\quad\quad\quad\quad (\texttt{>=}\ (\texttt{second}_{i,\Phi_i(E[\![y \times z]\!]i)}\ (\texttt{second}_i\ c))\ 0))
\end{array}
$$

A formal description of our syntax-directed translation rules for a subset of B's set operators can be seen in Figure 6.1.

For the translation of the direct product $\otimes$, let
T1 be the sort $\Theta_i(\theta_2(\mathbb{P}^{-1}(\Phi_i(E[\![p]\!]i))), \theta_2(\mathbb{P}^{-1}(\Phi_i(E[\![q]\!]i))))$ and
T2 be $\Theta_i(\theta_1(\mathbb{P}^{-1}(\Phi_i(E[\![p]\!]i))), \mathrm{T1})$):

$$
\begin{array}{l}
E[\![p \otimes q]\!]i \;\widehat{=}\; (\texttt{lambda}\ ((c\ \mathrm{T2})) \\
\quad (\texttt{exists}\ ((\mathrm{c2}\ \mathrm{T1}))\ (\texttt{and} \\
\quad\quad\quad (\texttt{in}\ (\texttt{tuple}_i\ (\texttt{first}_i\ c)\ (\texttt{first}_i\ \mathrm{c2}))\ E[\![p]\!]i) \\
\quad\quad\quad (\texttt{in}\ (\texttt{tuple}_i\ (\texttt{first}_i\ c)\ (\texttt{second}_i\ \mathrm{c2}))\ E[\![q]\!]i) \\
\quad\quad\quad (\texttt{=}\ (\texttt{second}_i\ c)\ \mathrm{c2})))
\end{array}
$$

To translate the parallel product $\|$, let
T1 be the sort $\Theta_i(\theta_1(\mathbb{P}^{-1}(\Phi_i(E[\![p]\!]i))), \theta_1(\mathbb{P}^{-1}(\Phi_i(E[\![q]\!]i))))$ and
T2 be $\Theta_i(\theta_2(\mathbb{P}^{-1}(\Phi_i(E[\![p]\!]i))), \theta_2(\mathbb{P}^{-1}(\Phi_i(E[\![q]\!]i))))$:

$$
\begin{array}{l}
E[\![p \parallel q]\!]i \;\widehat{=}\; (\texttt{lambda}\ ((c\ \Theta_i(\mathrm{T1}, \mathrm{T2}))) \\
\quad (\texttt{exists}\ ((\mathrm{c2}\ \mathrm{T1})\ (\mathrm{c3}\ \mathrm{T2}))\ (\texttt{and} \\
\quad\ (\texttt{in}\ (\texttt{tuple}_i\ (\texttt{first}_i\ \mathrm{c2})\ (\texttt{first}_i\ \mathrm{c3}))\ E[\![p]\!]i) \\
\quad\ (\texttt{in}\ (\texttt{tuple}_i\ (\texttt{second}_i\ \mathrm{c2})\ (\texttt{second}_i\ \mathrm{c3}))\ E[\![q]\!]i) \\
\quad\ (\texttt{=}\ (\texttt{first}_i\ c)\ \mathrm{c2})\ (\texttt{=}\ (\texttt{second}_i\ c)\ \mathrm{c3})))
\end{array}
$$

Finite Subsets

The finite set operators min, max, and card cannot be expressed efficiently using lambda functions. We thus stick to the axiomatic translation using quantifiers for these operators [48] as described in Section 6.3.1. While the same applies for the Event-B operator finite, the operators describing all finite subsets $\mathbb{F}$ and all finite non-empty subsets $\mathbb{F}_1$ can be expressed using lambda functions as is formalized in the following:

$$E[\![\mathrm{finite}(S)]\!]i \equiv E[\![\exists(b, f).(b \in \mathbb{N} \wedge f \in S \rightarrow 0\,..\,b)]\!]i$$

$$\begin{array}{l} E\llbracket m..n\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda ((}k\ \texttt{Int))}\\ \quad\texttt{(and (>= }k\ E\llbracket m\rrbracket i\texttt{) (<= }k\ E\llbracket n\rrbracket i\texttt{)))}\end{array}$$

$$\begin{array}{l} E\llbracket \mathbb{P}(S)\rrbracket i \mathrel{\widehat{=}}\\ \quad\texttt{(lambda ((}x\ \texttt{(Array }\Phi_i(E\llbracket S\rrbracket i)\ \texttt{Bool)))}\\ \qquad\texttt{(subset }x\ E\llbracket S\rrbracket i\texttt{))}\end{array}$$

$$\begin{array}{l} E\llbracket \mathbb{P}_1(S)\rrbracket i \mathrel{\widehat{=}}\\ \quad\texttt{(lambda ((}x\ \texttt{(Array }\Phi_i(E\llbracket S\rrbracket i)\ \texttt{Bool)))}\\ \qquad\texttt{(and (subset }x\ E\llbracket S\rrbracket i\texttt{)}\\ \qquad\qquad\texttt{(not (= }x\ \texttt{emptySet))))}\end{array}$$

$$\begin{array}{l} E\llbracket \mathrm{id}(S)\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}c\ \Theta_i(\mathbb{P}^{-1}(\Phi_i(E\llbracket S\rrbracket i)), \mathbb{P}^{-1}(\Phi_i(E\llbracket S\rrbracket i)))\texttt{))}\\ \quad\texttt{(and (in (first}_i\ c\texttt{)}\ E\llbracket S\rrbracket i\texttt{)}\\ \qquad\quad\texttt{(= (first}_i\ c\texttt{) (second}_i\ c\texttt{))))}\end{array}$$

$$\begin{array}{l} E\llbracket S\times T\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}c\ \Theta_i(\mathbb{P}^{-1}(\Phi_i(E\llbracket S\rrbracket i)), \mathbb{P}^{-1}(\Phi_i(E\llbracket T\rrbracket i)))\texttt{))}\\ \quad\texttt{(and (in (first}_i\ c\texttt{)}\ E\llbracket S\rrbracket i\texttt{)}\\ \qquad\quad\texttt{(in (second}_i\ c\texttt{)}\ E\llbracket T\rrbracket i\texttt{)))}\end{array}$$

$$\begin{array}{l} E\llbracket \mathrm{dom}(r)\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}x\ \theta_1(\mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i)))\texttt{))}\\ \quad\texttt{(exists ((}y\ \theta_2(\mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i)))\texttt{))}\\ \qquad\quad\texttt{(in (tuple}_i\ x\ y\texttt{)}\ E\llbracket r\rrbracket i\texttt{)))}\end{array}$$

$$\begin{array}{l} E\llbracket \mathrm{ran}(r)\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}y\ \theta_2(\mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i)))\texttt{))}\\ \quad\texttt{(exists ((}x\ \theta_1(\mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i)))\texttt{))}\\ \qquad\quad\texttt{(in (tuple}_i\ x\ y\texttt{)}\ E\llbracket r\rrbracket i\texttt{)))}\end{array}$$

$$\begin{array}{l} E\llbracket r^{-1}\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}c\ \Theta_i(\theta_2(\mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i))),\\ \qquad\qquad \theta_1(\mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i))))\texttt{))}\\ \quad\texttt{(in (tuple}_i\ \texttt{(second}_i\ c\texttt{) (first}_i\ c\texttt{))}\ E\llbracket r\rrbracket i\texttt{))}\end{array}$$

$$\begin{array}{l} E\llbracket S\lhd r\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda ((}c\ \mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i))\texttt{))}\\ \qquad\texttt{(and (in }c\ E\llbracket r\rrbracket i\texttt{)}\\ \qquad\qquad\texttt{(in (first}_i\ c\texttt{)}\ E\llbracket S\rrbracket i\texttt{)))}\end{array}$$

$$\begin{array}{l} E\llbracket S\mathbin{\lhd\!\!\!-} r\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda ((}c\ \mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i))\texttt{))}\\ \qquad\texttt{(and (in }c\ E\llbracket r\rrbracket i\texttt{)}\\ \qquad\qquad\texttt{(not (in (first}_i\ c\texttt{)}\ E\llbracket S\rrbracket i\texttt{))))}\end{array}$$

$$\begin{array}{l} E\llbracket r\rhd T\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda ((}c\ \mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i))\texttt{))}\\ \qquad\texttt{(and (in }c\ E\llbracket r\rrbracket i\texttt{)}\\ \qquad\qquad\texttt{(in (second}_i\ c\texttt{)}\ E\llbracket T\rrbracket i\texttt{)))}\end{array}$$

$$\begin{array}{l} E\llbracket r\mathbin{-\!\!\!\rhd} T\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda ((}c\ \mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i))\texttt{))}\\ \quad\texttt{(and (in }c\ E\llbracket r\rrbracket i\texttt{)}\\ \qquad\qquad\texttt{(not (in (second}_i\ c\texttt{)}\ E\llbracket T\rrbracket i\texttt{))))}\end{array}$$

$$E\llbracket \mathrm{r1} \mathbin{\lhd\!\!\!+} \mathrm{r2}\rrbracket i \equiv E\llbracket \mathrm{r2} \cup (\mathrm{dom}(\mathrm{r2}) \mathbin{\lhd\!\!\!-} \mathrm{r1})\rrbracket i$$

$$\begin{array}{l} E\llbracket r[S]\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda ((}y\ \theta_2(\mathbb{P}^{-1}(\Phi_i(E\llbracket r\rrbracket i)))\texttt{))}\\ \quad\texttt{(exists ((}x\ \mathbb{P}^{-1}(\Phi_i(E\llbracket S\rrbracket i))\texttt{))}\\ \qquad\texttt{(and (in }x\ E\llbracket S\rrbracket i\texttt{)}\\ \qquad\qquad\texttt{(in (tuple}_i\ x\ y\texttt{)}\ E\llbracket r\rrbracket i\texttt{))))}\end{array}$$

$$\begin{array}{l} E\llbracket \mathrm{union}(S)\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}e\ \mathbb{P}^{-1}(\mathbb{P}^{-1}(\Phi_i(E\llbracket S\rrbracket i)))\texttt{))}\\ \quad\texttt{(exists ((}\mathrm{sub}\ \mathbb{P}^{-1}(\Phi_i(E\llbracket S\rrbracket i))\texttt{))}\\ \qquad\texttt{(and (in }\mathrm{sub}\ E\llbracket S\rrbracket i\texttt{) (in }e\ \mathrm{sub}\texttt{))))}\end{array}$$

$$\begin{array}{l} E\llbracket \mathrm{inter}(S)\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}e\ \mathbb{P}^{-1}(\mathbb{P}^{-1}(\Phi_i(E\llbracket S\rrbracket i)))\texttt{))}\\ \quad\texttt{(forall ((}\mathrm{sub}\ \mathbb{P}^{-1}(\Phi_i(E\llbracket S\rrbracket i))\texttt{))}\\ \qquad\texttt{(implies (in }\mathrm{sub}\ E\llbracket S\rrbracket i\texttt{) (in }e\ \mathrm{sub}\texttt{))))}\end{array}$$

$$\begin{array}{l} E\llbracket \lambda z.(\mathrm{Pred}\ \mid\ \mathrm{Expr})\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}c\ \Theta_i(\Phi_i(E\llbracket z\rrbracket i), \Phi_i(E\llbracket \mathrm{Expr}\rrbracket i))\texttt{))}\\ \quad\texttt{(exists ((}E\llbracket z\rrbracket i\ \Phi_i(E\llbracket z\rrbracket i)\texttt{))}\\ \qquad\texttt{(and }P\llbracket \mathrm{Pred}\rrbracket i\\ \qquad\qquad\texttt{(= }c\ \texttt{(tuple}_i\ E\llbracket z\rrbracket i\ E\llbracket \mathrm{Expr}\rrbracket i\texttt{)))))}\end{array}$$

Figure 6.1.: A formal description of our syntax-directed translation rules for translating a subset of B's set operators to SMT-LIB as understood by Z3. In particular, lambda functions are not part of the latest SMT-LIB standard version 2.6, but are supported by Z3. The environment i and the functions Θ_i, θ_1, θ_2, Φ_i, and $\mathbb{P}^{-1}$ are defined in the introduction of Section 6.4.1.

$$\begin{array}{l} E\llbracket \mathbb{F}(S)\rrbracket i \mathrel{\widehat{=}} \texttt{(lambda}\\ \quad\texttt{((}x\ \texttt{(Array }\Phi_i(E\llbracket S\rrbracket i)\ \texttt{Bool)))}\\ \qquad\texttt{(and (subset }x\ E\llbracket S\rrbracket i\texttt{) (}E\llbracket \mathrm{finite}\rrbracket i\ x\texttt{)))}\end{array}$$

```
E⟦𝔽₁(S)⟧i ≙ (lambda
  ((x (Array Φᵢ(E⟦S⟧i) Bool)))
    (and (subset x E⟦S⟧i) (E⟦finite⟧i x)
          (not (= x emptySet))))
```

Rewriting Set Cardinality and Power Set

Since Z3 often lacks performance when solving quantified formulas [48], we provide special rewriting rules for B set cardinality and power set constraints to equivalent representations which do not lead to quantified formulas in SMT-LIB. In particular, we provide the following rewriting rules, where the set $\{x_1, \ldots, x_n\}$ with $n \in \mathbb{N}$ contains at least one variable:

$$S \in \mathbb{P}(R) \equiv S \subset R$$

$$S \in \mathbb{P}_1(R) \equiv S \subset R \wedge S \neq \varnothing$$

$$S \in \mathbb{F}(R) \equiv S \subset R \text{ if R is finite}$$

$$S \in \mathbb{F}_1(R) \equiv S \subset R \wedge S \neq \varnothing \text{ if R is finite}$$

$$\text{card}(S) > 0 \equiv \text{card}(S) \geq 1 \equiv \text{card}(S) \neq 0 \equiv S \neq \varnothing$$

$$\text{card}(S) = 0 \equiv \text{card}(S) < 1 \equiv S = \varnothing$$

$$\text{card}(\{x_1, \ldots, x_n\}) = n \equiv \text{card}(\{x_1, \ldots, x_n\}) >= n \equiv \\ \text{card}(\{x_1, \ldots, x_n\}) > n - 1 \equiv \text{all\_different}(\{x_1, \ldots, x_n\})$$

$$\text{card}(\{x_1, \ldots, x_n\}) < n \equiv \text{card}(\{x_1, \ldots, x_n\}) \leq n - 1 \equiv \\ \text{at\_least\_one\_eq}(\{x_1, \ldots, x_n\})$$

$$\text{card}(\{x_1, \ldots, x_n\}) \leq k \equiv \text{card}(\{x_1, \ldots, x_n\}) < k + 1 \equiv \top \text{ if } k \geq n$$

$$\text{card}(\{x_1, \ldots, x_n\}) > k \equiv \text{card}(\{x_1, \ldots, x_n\}) \geq k + 1 \equiv \bot \text{ if } k \geq n$$

$$q \in 1\,..\,n \to 1\,..\,n \wedge \text{card}(\text{ran}(q)) = n \equiv \bigwedge_{i \in 1\,..\,n-1} q(i) \neq q(i+1)$$

Here, all_different is a constraint that sets up a pairwise distinction between all elements, and at_least_one_eq is a constraint that enforces at least one equality between the set elements using a disjunction of pairwise equalities. Furthermore, we replace set cardinality constraints of enumerated sets that do not contain a variable with integer values. For instance, we can simplify the B constraint $s = 1\,..\,4 \wedge \text{card}(s) > 1 \wedge i = \text{card}(s) - 1$ to $s = 1\,..\,4 \wedge i = 3$ to prevent sending any cardinality constraint to Z3. Such formulas might not be written by hand but do often occur when using an automated translation backend of ProB such as the integration [132] of TLA$^+$ [41] in B.

Relational Composition, Iteration, and Closure

Some relational B operators such as the transitive and reflexive closure are more complex to translate to SMT-LIB and are discussed in the following. The transitive and reflexive closure r^* of a relation $r \in S \leftrightarrow S$ can be mathematically defined as $\bigcup_{n \in \mathbb{N}} r^n$ and the transitive and not reflexive closure r^+ as $\bigcup_{n \in \mathbb{N}_1} r^n$. Here, the transitive and reflexive closure is defined by the union of a relation's iterations for all natural numbers.

The iteration of a relation $r \in S \leftrightarrow S$ can be defined recursively using B's forward composition. This conforms to the formula $r^n = r^{n-1}; r^1$, where the base case is $r^1 = r$. One special case of the relational iteration's definition in B is $r^0 = \mathrm{id}(S)$, which is rewritten before the translation. B's forward composition of two relations $p\,;q$ is defined by the set comprehension $\{x, y \mid \exists z.(x \mapsto z \in p \wedge z \mapsto y \in q)\}$ which can be straightforwardly translated to SMT-LIB using lambda functions. Let T1 be the sort $\Phi_i(E[\![p]\!]i)$ and T2 be $\Phi_i(E[\![q]\!]i)$. We then translate the forward composition $p\,;q$ as follows:

```
(define-fun fcomp ((r1 T1) (r2 T2))
  (Array $\Theta_i(\theta_1(\mathbb{P}^{-1}(\mathrm{T1})), \theta_2(\mathbb{P}^{-1}(\mathrm{T2})))$ Bool)
  (lambda (($c$ $\Theta_i(\theta_1(\mathbb{P}^{-1}(\mathrm{T1})), \theta_2(\mathbb{P}^{-1}(\mathrm{T2})))$))
    (exists (($z$ $\theta_2(\mathbb{P}^{-1}(\mathrm{T1}))$))
      (and (in (tuple$_i$ (first$_i$ $c$) $z$) r1)
           (in (tuple$_i$ $z$ (second$_i$ $c$)) r2)))))
```

$E[\![p\,;q]\!]i \;\widehat{=}\;$ `(fcomp` $E[\![p]\!]i$ $E[\![q]\!]i$`)`

Note that a relational backward composition can be described by a forward composition, *i.e.*, $p \circ q \equiv q\,;p$. We are able to define the iteration of a relation r as a recursive function using the encoding of B's forward composition in SMT-LIB as follows:

```
(define-fun-rec iterate ((r1 $\Phi_i(E[\![r]\!]i)$) ($n$ Int))
  $\Phi_i(E[\![r]\!]i)$ (ite (= $n$ 1) r1 (fcomp (iterate r1 (- $n$ 1)) r1)))
```

$E[\![r^n]\!]i \;\widehat{=}\;$ `(iterate` $E[\![r]\!]i$ $E[\![n]\!]i$`)`

Due to the employed encoding of sets in SMT-LIB that introduces a sort for each type of set, *e.g.*, a set of the integers or a set of Boolean, we have to define the functions `iterate` and `fcomp` for each type that they are applied to. We thus define unique names for the different functions differing in the relation's type and case split on these types before translating to SMT-LIB.

Let `union` be a function passing its only argument to the lambda function for the translation of B's general union as defined in Section 6.4.1. The transitive and reflexive closure of a relation r can then be translated to SMT-LIB straightforwardly:

```
$E[\![r^*]\!]i \;\widehat{=}\;$ (union (lambda (($s$ $\Phi_i(E[\![r]\!]i)$))
  (exists (($n$ Int)) (and (>= $n$ 0) (= $s$ (iterate $E[\![r]\!]i$ $n$))))))
```

B's transitive and not reflexive closure r^+ is translated analogously but using $n \in \mathbb{N}_1$.

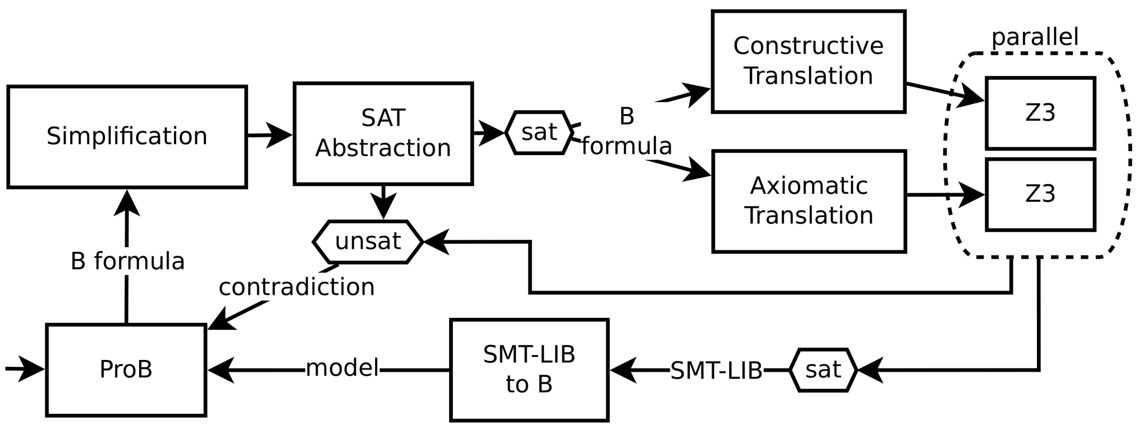


Figure 6.2.: A workflow diagram of the new integration of Z3 in ProB running two Z3 constraint solvers in parallel using the former and new translation from B to SMT-LIB.

Corrigendum. In Z3, the use of lambda functions inside recursive definitions was declared unsupported after a corresponding unsoundness was reported [186]. A developer of Z3 stated that there is still a lot of work to be done before Z3 fully supports lambda functions [186]. The translations of B's relational iteration and closure to SMT-LIB are thus currently unsupported in Z3 version `4.12.2`, which is the latest version at the time of writing this thesis. Their translations have been disabled in ProB until Z3 provides corresponding support. In theory, the presented translations remain valid, and we deem this issue to be an implementation detail within Z3.

6.4.2. New Workflow

The new workflow of ProB's Z3 interface is supposed to replace the former interface which sends full predicates to Z3 as described in Section 6.3.2. Note that ProB also has an interface to Z3 where both solvers share constraints which we do not consider here. A diagram of the workflow is presented in Figure 6.2.

Preprocessing

First, a formula is simplified by ProB as was the case for the former integration [48] of Z3. For instance, formulas are rewritten to use a subset of operators such as only using $\leq$ but not $\geq$.

We decided to apply a static analysis to check syntactically for contradictions before translating to SMT-LIB. The goal is to prevent that those contradictions are no longer detected by Z3, *e.g.*, after adding quantifiers. For this, we extended the simplification rules of ProB to more aggressively replace variables by their value if this value is explicitly given. For instance, the formula $s = \varnothing \land \mathrm{card}(s) > 1$ can be rewritten as $s = \varnothing \land \mathrm{card}(\varnothing) > 1$ in a first phase. Afterward, the cardinality constraint can be replaced by the integer 0 which makes it obvious that the integer comparison is not satisfied. We thus prevented the translation of a cardinality constraint to SMT-LIB. In addition, we compute specific constraints such as the Cartesian product of finite sets up to a predefined limit before the translation. This in turn prevents the introduction of quantifiers or lambda functions in SMT-LIB.

To further extend the static syntax analysis, we decided to abstract a B formula to a SAT formula as is done by lazy SMT solvers [99] and only translate a formula to SMT-LIB if its SAT abstraction is satisfiable as can be seen in Figure 6.2. If it is not satisfiable, we have avoided the overhead of translating B to SMT-LIB and calling the external constraint solver. For instance, the formula $x = y \wedge x \neq y$ can be abstracted to $\mathrm{A} \wedge \neg\mathrm{A}$ where $\mathrm{A} \equiv x = y$. Note that this is not an eager SMT solving [99] where all semantics are translated to propositional logic. We are now able to call a SAT solver to find a solution for an abstracted B formula. For this, we use a small timeout of 50 ms to prevent adding too much overhead due to SAT solving. In particular, a SAT solver should be able to identify simple static contradictions fast.

Z3 Solver Integration

If the SAT abstraction is satisfiable, we apply both translations from B to SMT-LIB: the preexisting one proposed by Krings and Leuschel [48] (Section 6.3.1), and the new one described in Section 6.4.1.

The former integration of Z3 always used the incremental solver where constraints can be pushed on to the solver stack. While this is required when both ProB and Z3 run simultaneously, this is not the case for the integration presented in this article, where we send full predicates to Z3 only. In particular, using the incremental solver incurs an additional overhead since constraints are internalized. We thus decided to run two non-incremental Z3 solvers in parallel with the two different translations as described above. Unfortunately, Z3's incremental solver does not support an existential quantifier at the top-level of a lambda expression.[1] This makes some of our new translation not applicable for running ProB's constraint solver and Z3 simultaneously and sharing constraints.

We use the result of the solver which answers first if a solution or a contradiction has been found. The other solvers are then interrupted. If the fastest solver answers unknown, we do not use this result but wait for another solver. The solver integration returns unknown if all solvers did so as well, or if a formula cannot be translated to SMT-LIB, *e.g.*, because of a missing implementation. The return of unknown is not shown in Figure 6.2.

Note that it is simple to add a Z3 solver configuration to the workflow. Our implementation is able to create a deep copy of a translation with all of its referenced Z3 objects, which are stored in a so-called context in Z3. We then just have to create a new solver object for a copied context and set the desired options.

[1]Z3 throws the error "internalization of exists is not supported".

Postprocessing of Models

A model found by Z3 is represented in SMT-LIB. We parse a model and translate it to B as was the case for the former workflow integration described in Section 6.3.2. Unfortunately, Z3 often fails to compute explicit values from lambda functions or quantifiers while it is able to find contradictions. For instance, for the formula $s = \text{union}(\{\{1\}, \{2\}\})$, Z3 returns a model containing the translated lambda function of the general union defined in Section 6.4.1 while s could be set to $\{1, 2\}$. However, Z3 is able to find contradictions using the general union such as for $\varnothing = \text{union}(\{\{1\}, \{2\}\})$. We thus extend the translation from SMT-LIB to B and the processing of found models to compute remaining quantifiers and lambda functions with ProB's constraint solver. For instance, the lambda function in the above example's model returned by Z3 is translated as a set comprehension in B which results in $s = \{e \mid \exists f.(e \in f \wedge (f = \{1\} \vee f = \{2\}))\}$. The ProB constraint solver is then called to compute an explicit value which results in $s = \{1, 2\}$.

6.4.3. Decomposition of Constraints

In recent work [49] we observed that Z3 often lacks performance for constraints containing many quantifiers.

We mainly attribute these performance issues to the use of set cardinality constraints as well as the definition of functions, which are translated using quantifiers in SMT-LIB to axiomatize their behavior. While Z3 is able to solve many constraints involving such axiomatized translations, it often fails to solve constraints with many quantifiers. In particular, the Z3 solver often answers unknown.

We thus decided to decompose constraints into components that use a distinct set of variables prior to the translation to SMT-LIB and solve each of these components independently using Z3. For instance, the B constraint $x \in \mathbb{N} \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{N} \wedge x \neq \varnothing \wedge y \in \mathbb{N} \rightarrow \mathbb{N} \wedge y \neq \varnothing$ can be decomposed into two independent components $x \in \mathbb{N} \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{N} \wedge x \neq \varnothing$ and $y \in \mathbb{N} \rightarrow \mathbb{N} \wedge y \neq \varnothing$. We suppose that Z3 is able to solve several small constraints better than is the case for one large constraint. Furthermore, the performance should increase for unsatisfiable constraints if a single component is already unsatisfiable in which case not all components have to be solved.

Algorithm 6.1 shows a pseudocode implementation of the solving routine that decomposes constraints into independent components prior to the translation to SMT-LIB. The function DECOMPOSE used in line 2 is a function that decomposes a B constraint into independent components where the output is a set of constraints. We iterate over the set of components and solve each component with Z3 after translating a B constraint to SMT-LIB (line 6 of Algorithm 6.1) by applying the workflow described in Section 6.4.2. If a component is unsatisfiable, we return from the solving routine by stating that a contradiction has been found. In this case, we do not have to solve possibly remaining components since the unsatisfiability of a single component implies the unsatisfiability of the overall constraint. If a component is satisfiable, we combine the variable assignments of this component's solutions with the overall satisfiable variable assignments (line 12 of

Algorithm 6.1 Pseudocode of the constraint solving routine for the integration of Z3 in ProB that decomposes B constraints into independent components.

Input a B formula ϕ
Output a satisfiable assignment of variables occurring in ϕ, a false statement indicating the unsatisfiability or unknown

```
 1: procedure SOLVE_DECOMPOSED($\phi$)
 2:     C ← DECOMPOSE($\phi$)
 3:     u ← $\perp$
 4:     res ← $\varnothing$
 5:     for all $c \in$ C do
 6:         c_res ← SOLVE_WITH_Z3($c$)
 7:         if c_res = $\perp$ then
 8:             return $\perp$
 9:         else if c_res = unknown then
10:             u ← $\top$
11:         else
12:             res ← COMBINE_RESULTS(res, c_res)
13:     if u = $\top$ then
14:         return unknown
15:     return res
```

Algorithm 6.1). This combination of solutions results in appending the lists of variable bindings since each component refers to an independent set of variables.

The Z3 solver possibly answers unknown when solving a constraint. However, if Z3 is unable to decide for the satisfiability of a single component, it might be able to detect the unsatisfiability of a remaining component which determines the unsatisfiability of the overall constraint. We thus store the information that Z3 was unable to solve a single component in a Boolean variable introduced in line 3 of Algorithm 6.1 and do not terminate the solving routine if Z3 is unable to decide for the satisfiability of a single component (line 9 and 10 of Algorithm 6.1). We return unknown if all components have been solved and Z3 was unable to decide for the satisfiability of a single component (line 13 and 14 of Algorithm 6.1). Otherwise, the overall result is returned in line 15 of Algorithm 6.1.

6.5. SMT Solving in ProB

The integration of Z3 in ProB has shown benefits for solving B and Event-B constraints [49]. Yet, the encoding of sets as characteristic functions in SMT-LIB is suboptimal for several constraints such as the set cardinality or the minimum and maximum of a set of integers. We thus decided to implement state-of-the-art SMT solving techniques directly in ProB to tightly connect ProB's constraint solving core for finding solutions with a CDCL(T)-based learning scheme to prune the search space early and improve the

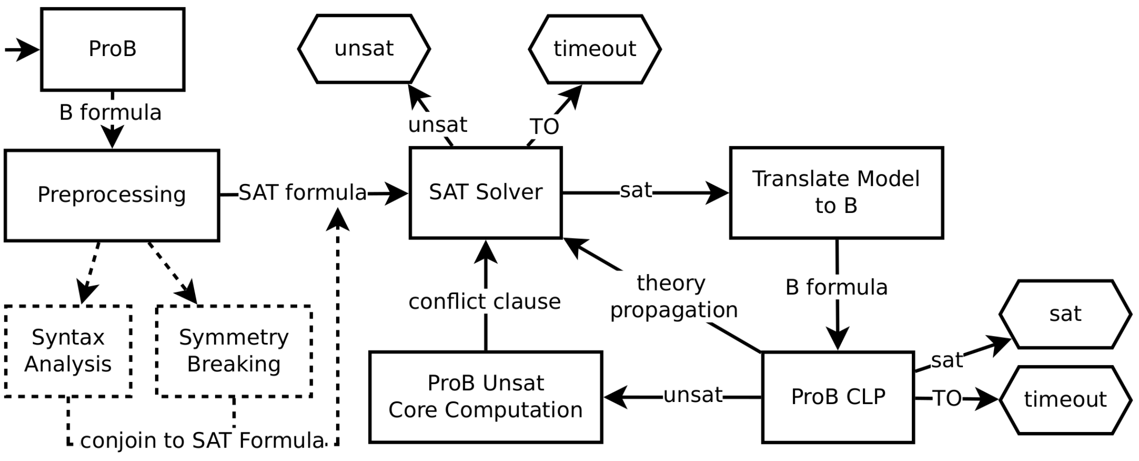


Figure 6.3.: A generalized workflow diagram of the direct implementation of SMT solving in ProB. The dashed paths represent an optional static syntax analysis and symmetry breaking. The workflow diagram does not show the use of features such as early pruning.

identification of contradictions. In the following, we describe our implementation of the lazy SMT approach for the B language in ProB. In the process, we also describe the standard techniques of SMT solving that we have implemented to address a broad audience.

6.5.1. SMT Workflow in ProB

In Figure 6.3 we present the main workflow diagram of our integration of SMT solving in ProB. The input to the SMT solver is a B formula and ProB's constraint solver (ProB CLP) is the only theory solver by default. Note that our SMT solver does not support the SMT-LIB language as input like other SMT solvers usually do. The dashed paths in the workflow diagram represent an optional static syntax analysis and symmetry breaking. Both techniques are independent and can be applied together, alone or not at all. The result of the workflow can be a satisfying assignment of variables (sat), a contradiction (unsat) or a timeout, which can either be caused by the SAT or theory solver. The specific stages and applied techniques of the workflow as well as our implementation are discussed in the remainder of this section.

6.5.2. Preprocessing

First and foremost, B formulas need to be abstracted to SAT formulas for the Boolean satisfiability part of the SMT solver. We transform a propositional logic formula to conjunctive normal form as is the case for most SAT solvers. Additionally, we try to improve SMT solving by deducing different constraints that minimize the search space as explained in the following.

SAT Abstraction

First, we rewrite formulas to only use conjunctions and disjunctions by rewriting implications and equivalences. We define two functions T2B(α) and B2T(β) which translate a theory formula to propositional logic and vice versa. The function T2B replaces conjuncts and disjuncts by unique Boolean variables. For instance, let α be the B predicate $x \neq y \wedge x = y$. The Boolean abstraction is defined as T2B(α) $= \neg \mathrm{A} \wedge \mathrm{A}$ where $\mathrm{A} \equiv x = y$. The negation has been lifted from the inequality to reduce the amount of introduced Boolean variables. Furthermore, contradictions are possibly shifted from the theory level to the Boolean level which improves the performance by preventing unnecessary calls to a theory solver. We deem this to be one of the main improvements of SMT solving compared to saturation-based solving as performed by ProB's constraint solver since the enumeration of theory domains is possibly prevented. This is desirable in the context of the B language and especially ProB's constraint solver since domains can be unbounded, which possibly makes exhaustive domain enumeration and disproving infeasible. In order to reduce the amount of introduced Boolean variables, we normalize a formula by applying ProB's internal rewriting rules for optimization before calling the function T2B. For instance, the arguments of commutative operators are sorted lexicographically, obvious tautologies and contradictions are removed, and a subset of operators is used such as only using $\leq$ but not $\geq$. Note that a quantifier in an SMT formula is abstracted by a single Boolean variable, *e.g.*, introducing a variable $\mathrm{A} \equiv \forall x.(x \in \mathbb{N} \Rightarrow x \geq 0)$. A resulting SAT abstraction thus does not contain any quantifier.

A Boolean formula using only conjunctions and disjunctions can be transformed to conjunctive normal form by applying DeMorgan's laws as well as the distributive law. Yet, rewriting disjunctions of nested conjunctions can lead to an exponential growth in the amount of clauses of a conjunctive normal form [60], which obviously can impact solver performance. Tseitin [60] has shown that the amount of clauses can be reduced by introducing artificial Boolean variables for specific formulas, which we implement as well. For instance, the distributive formula $(\mathrm{A} \wedge \mathrm{B} \wedge \mathrm{C}) \vee (\mathrm{D} \wedge \mathrm{E} \wedge \mathrm{F})$ can be rewritten as $((\mathrm{A} \wedge \mathrm{B} \wedge \mathrm{C}) \vee \mathrm{P}) \wedge (\mathrm{P} \Leftrightarrow (\mathrm{D} \wedge \mathrm{E} \wedge \mathrm{F}))$. Furthermore, nested equivalences and equivalences under disjunctions are rewritten in the same manner because they also expand to disjunctions of conjunctions.

Static Symmetry Breaking

A lot of logical formulas contain symmetries which lead to redundant paths in the search space [187]. In general, a logical formula is a symmetry of another formula if both formulas are syntactically equal except for variable permutations which maintain satisfiability. The size of the search space can be reduced by breaking symmetries either statically before the search or dynamically during the search. For instance, we can deduce the symmetry breaking constraint $x \leq y$ for the formula $x < y \wedge y < x$ since the variables x and y can be exchanged without changing the semantics. We assume that B formulas often contain symmetries since the language is based on set theory and

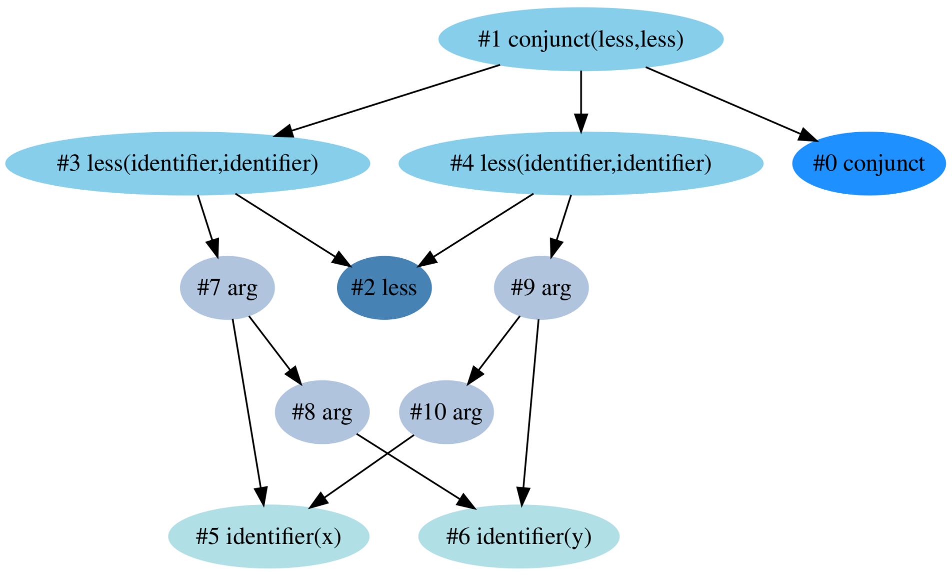


Figure 6.4.: A colored graph to find static symmetry breaking predicates for the formula $x < y \wedge y < x$ by computing graph automorphisms as proposed by Areces et al. [188].

integer arithmetic, which provide several commutative operators. Breaking symmetries supports the theory solver. We thus deem symmetry breaking to be a valuable technique for an SMT solver in the context of the B language. While there exist techniques to break symmetries for SAT formulas, it is a pitfall to use such techniques in the context of SMT solving. The resulting symmetry breaking predicates for a SAT formula neglect the theory and can thus lead to spurious contradictions. For instance, consider the formula $\mathrm{A} \wedge (\mathrm{B} \vee \mathrm{C})$ with $\mathrm{A} \equiv x \in \mathbb{Z}$, $\mathrm{B} \equiv x > 1$, and $\mathrm{C} \equiv x \bmod 2 = 0$. It is valid to break the symmetry for the variables B and C in propositional logic, *e.g.*, allowing the partial model $\mathrm{B} = \top \wedge \mathrm{C} = \bot$ but forbidding $\mathrm{B} = \bot \wedge \mathrm{C} = \top$. Yet, it is not correct to break this symmetry in the context of SMT solving since the corresponding theory constraints $x > 1$ and $x \bmod 2 = 0$ are not symmetric.

Areces et al. [188] presented an algorithm to statically compute symmetry breaking constraints for SMT formulas. The idea is to encode an SMT formula as a colored, directed, and acyclic graph where symmetries of the formula are described by automorphism groups. A graph automorphism is an isomorphism from a graph onto itself, *i.e.*, a bijective mapping $h \in G \rightarrowtail\!\!\!\!\twoheadrightarrow G$ such that $(v, w) \in \mathrm{E} \Leftrightarrow (h(v), h(w)) \in E$ for all edges $(v, w) \in E$. There exist polynomial algorithms for detecting automorphisms in graphs with a bounded degree (number of a node's incident edges) [189].

The process of symmetry breaking is split in two stages which are the creation and coloring of the graph [188]. A node is introduced for each interpreted and uninterpreted symbol as well as for constants. For instance, the colored graph for the above example $x < y \wedge y < x$ can be seen in Figure 6.4. We prefixed each node by a number as our implementation works with numbers instead of names for nodes. For the given example, we add one node for the uninterpreted symbols of the conjunction (number 0) and integer comparison (number 2), one node for the complete interpreted conjunction (number 1) and both integer comparisons (number 3 and 4), and one node for each argument (number 7 to 10) as well as the identifiers (number 5 and 6), which are treated as constants. The edges in the graph are set depending on the commutativity of operators. If an operator is not commutative, its arguments are ordered by adding one edge from the interpreted symbol node to the first argument's node as well as an edge from the first argument's node to the second one and so on. For instance, the integer comparisons in Figure 6.4 are not commutative so that the second argument can only be reached through the first argument in the graph. Otherwise, one edge from the interpreted symbol node to each argument's node is added as is the case for the conjunction in Figure 6.4. The colors of the nodes are split into three classes for interpreted and uninterpreted symbols as well as nodes for interpreted symbols' arguments. Each uninterpreted symbol is assigned a unique color, *e.g.*, nodes number 0 and 2 in Figure 6.4. We implemented this technique for B in PROB's Prolog core and interface bliss [190] using its C++ API to compute graph automorphisms. Each automorphism group is represented as a set of generators by bliss. A symmetry breaking predicate is generated for each set of generators, which allows for only one symmetric solution. For the above example, bliss computes one automorphism group which is represented by the set of generators $\{((3,4),(5,6),(7,9),(8,10))\}$. We can now generate a symmetry breaking predicate by deciding a variable ordering, *e.g.*, lexicographic, and computing the image of each variable under the automorphism group. Here, the nodes with number 5 and 6 correspond to the variables x and y. The image of x under the automorphism group is y so that we add the symmetry breaking constraint $x \leq y$. No symmetry breaking predicate is added for the variable y since its image under the automorphism group is the same variable y.

Note that this technique also ensures finding nested symmetries. For instance, consider the predicate $z > 1 \vee (x > y \wedge y > x)$. The colored graph for symmetry breaking contains a top-level disjunction which right-hand side is the colored graph presented in Figure 6.4. The disjunction's left-hand side is a colored graph for the integer comparison constraint pointing to a node for the variable z, which is independent of the disjunction's right-hand side. We thus find the same graph automorphism as before. Now, consider that the disjunction's left-hand side is the constraint $x > 1$. We then add an edge from the disjunction's left-hand side to the node of the variable x in the graph presented in Figure 6.4. This breaks the graph automorphism since the variables x and y cannot be exchanged anymore without changing the semantics of the predicate. The described technique thus correctly recognizes that this predicate does not contain any symmetries between variables.

Besides searching for a constraint's global symmetries, we further apply symmetry breaking to locally quantified formulas.

Static Syntax Analysis

Besides applying static symmetry breaking, we extend the static syntax analysis to deduce constraints which imply one another but do not necessarily break symmetries. For instance, we can deduce the constraint $x < y \Rightarrow \neg(y < x)$ for the formula $x < y \wedge y < x$. This constraint moves the contradiction from the theory level to the SAT level of the SMT solver which is not the case for the symmetry breaking constraint $x \leq y$. We thus deem this additional static analysis to be valuable for SMT solving in the context of B and ProB's constraint solver since possibly more enumerations of domains in the theory solver are prevented. Note that this syntax analysis only considers subformulas that are present in the input formula and does not introduce new formulas. For instance, we do not deduce the constraint $x < y \Rightarrow \neg(y \leq x)$ since $y \leq x$ is not part of the input formula.

For this analysis, we only consider direct implications of pairs of formulas which share at least one variable. Due to performance regards, we do not consider transitive or other variable dependencies between formulas. Furthermore, we define a set of operators which we want to check for whether they imply one another. In particular, we use the equality, set membership, subset relations, and integer comparisons. We collect all candidate constraints and group them by their types as well as the amount of used variables, which is either one or two. Afterward, we check for all pairs of constraints c_1, c_2 with $\mathrm{c}_1 \neq \mathrm{c}_2$ in each set of candidates if $\mathrm{c}_1 \Rightarrow \mathrm{c}_2$, $\mathrm{c}_1 \Rightarrow \neg\mathrm{c}_2$, $\mathrm{c}_2 \Rightarrow \mathrm{c}_1$, and/or $\mathrm{c}_2 \Rightarrow \neg\mathrm{c}_1$. For the above example, this results in solving the constraint $\forall(x, y).(x \in \mathbb{Z} \wedge y \in \mathbb{Z} \Rightarrow (x < y \Rightarrow \neg(y < x)))$. Alternatively, a counterexample can be searched for the negated formula resulting in an existentially quantified formula. To prevent possible performance issues due to the enumeration of (unbounded) domains, we use ProB's prover [191] to prove such constraints instead of its constraint solver. Therewith, we are able to drop the universal or existential quantifier to prove the actual constraint.

6.5.3. SAT Solving

The problem of satisfiability solving is NP-completeness and many possible improvements of decision procedures have been suggested to date. The basis of our SAT solver is the solver presented by Howe and King [192] which implements the watched literals scheme [89] by using coroutines in Prolog. We extend this implementation by different variable selection heuristics, conflict-driven clause learning with the reduction of learned clauses, and restarts with phase saving.

Conflict-Driven Clause Learning

The DPLL algorithm decides the value of a selected variable if no unit clause is present. This decision poses a choice point and leads to backtracking when finding a conflict. Yet, the last decision might not be the root cause of a conflict. In this case, chronological backtracking leads to unnecessary overhead. Furthermore, a constraint solver should not find the same conflict again in an ongoing search. The idea of conflict-driven clause

$$\dfrac{\dfrac{\dfrac{\mathrm{E} \lor \underline{\neg \mathrm{F}} \qquad \neg \mathrm{A} \lor \mathrm{D} \lor \underline{\mathrm{F}}}{\neg \mathrm{A} \lor \mathrm{D} \lor \underline{\mathrm{E}}} \qquad \mathrm{D} \lor \underline{\neg \mathrm{E}}}{\neg \mathrm{A} \lor \underline{\mathrm{D}}} \qquad \neg \mathrm{A} \lor \neg \mathrm{B} \lor \neg \mathrm{C} \lor \underline{\neg \mathrm{D}}}{\neg \mathrm{A} \lor \neg \mathrm{B} \lor \neg \mathrm{C}}$$

Figure 6.5.: Exemplary conflict analysis using resolution for the formula $(\neg A \lor \neg B \lor \neg C \lor \neg D) \land (D \lor \neg E) \land (\neg A \lor D \lor F) \land (E \lor \neg F) \land (C \lor \neg D)$ with the sequence of variable assignments $A^d, B^d, C^d, \neg D, \neg E$, and F. The clause $(E \lor \neg F)$ is a conflict. The superscript "d" indicates that this variable was assigned by decision. The other variables were assigned by unit propagation. Each step corresponds to a resolution between two clauses while the variables used for resolution are underlined. The example shows the complete conflict analysis, which can be stopped after deducing $(\neg A \lor D)$.

learning (CDCL) [87, 88] is to analyze the root cause of a conflict clause to learn a formula which prevents this conflict as well as a level in the search tree to backjump to. Learning means to add a clause to the current set of clauses. We deem clause learning to be one of the main improvements compared to ProB's constraint solver since it uses chronological backtracking and does not learn from conflicts. This often prevents ProB's constraint solver from disproving formulas, especially when using unbounded domains. In fact, this is a general downside of plain saturation-based solvers.

The cause of a Boolean conflict in the DPLL algorithm can be analyzed by applying resolution in a certain order or by building and analyzing an implication graph [87]. For both of these techniques, we have to keep track of the sequence of variable assignments, the clause which led to each specific assignment of a variable, the level of each variable propagation, the assigned polarity, and the type of the propagation, which is either a branching decision or a unit propagation. We decided to analyze conflicts by implementing the concept of resolution, which is more performant since no implication graph has to be built.

In both techniques, the idea is to trace the antecedent variable assignments that led to a specific unit propagation which is involved in the conflict. For instance, consider the propositional logic formula $(\neg A \lor \neg B \lor \neg C \lor \neg D) \land (D \lor \neg E) \land (\neg A \lor D \lor F) \land (E \lor \neg F) \land (C \lor \neg D)$. Further, assume that the SAT solver made the sequence of assignments $A^d, B^d, C^d, \neg D, \neg E$, and F. A superscript "d" represents a variable assignment made by decision while all other assignments are caused by unit propagation. The assignment of variables constitutes a contradiction on decision level 2 where $E \lor \neg F$ is the conflicting clause. Conflict analysis is performed backwards starting from the conflicting clause. The antecedent assignments of the unit propagation of F are the decision of A and the unit propagation of $\neg D$ due to the clause $\neg A \lor D \lor F$. When performing resolution with this clause and the conflicting clause, we derive a new clause $\neg A \lor D \lor E$ as can be seen in Figure 6.5. The analysis can be stopped once a clause has been derived that contains only one variable which has been assigned on the conflict level. Variables assigned by decision are not resolved by resolution. In the currently derived clause, the variables D and E are both assigned on level 2 so that resolution is continued. For instance, the variable E can be resolved by the clause $\mathrm{D} \vee \neg\mathrm{E}$ resulting in $\neg\mathrm{A} \vee \mathrm{D}$. This clause contains only the variable D that has been assigned on the current conflict level. We can thus terminate and learn the derived clause. The level in the search tree to backjump to is the highest decision level in the learned clause other than the conflict level. In our example, we backjump to the decision of A on level 0. This results in a unit propagation which changes the assignment of the identified root cause of the conflict, *i.e.*, the assignment $\neg\mathrm{D}$ in our example. One special case is that we always backjump to level 0 when learning a unit clause. This technique guarantees to find the shortest backjump clause by stopping after the first unique implication point (UIP) [193]. A unique implication point is a unit propagated variable assignment which is part of every path between the last variable decision that occurred before the unit propagation and the conflicting assignment. The complete conflict analysis for our example including one more resolution can be seen in Figure 6.5. While the clause $\neg\mathrm{A} \vee \neg\mathrm{B} \vee \neg\mathrm{C}$ prevents the conflict, the clause learned at the first UIP is more concise. Furthermore, terminating after the first UIP prevents unnecessary computations [193].

Reducing Learned Clauses

An SMT solver possibly uncovers many conflicts before deciding for satisfiability. While learning clauses from conflicts reduces the search space, the accumulation of too many clauses can slow down the search and can lead to an explosion of consumed memory. It is thus important to forget learned clauses once in a while. Audemard and Simon [194] proposed a technique to forget weak clauses which uses the measure of the literal block distance (LBD) to definitely keep strong clauses that we implement. The literal block distance of a clause is the number of different decision levels in this clause [194]. The authors state that clauses with an LBD of two are most important because they connect two decision levels. In particular, clauses with a small LBD, *e.g.*, between two and five, should not be removed. The half of all other clauses is removed occasionally considering the amount of performed reductions of the set of learned clauses so far. In particular, an SMT solver forgets fewer clauses over time. The LBD score of a clause is computed and stored when it is learned and thus refers to the state of the search tree at that time.

Variable Selection Heuristics

The selection of the next variable and polarity to assign influences the performance of SAT solving. Many variable selection heuristics have been proposed to date. Moskewicz et al. evaluated different variable selection heuristics during the development of the SAT solver Chaff and proposed an improved heuristic called the variable state independent decaying sum (VSIDS) [89]. The VSIDS heuristic assigns a float value to each variable where a variable with the highest score is assigned next. Initially, all values are set to the corresponding variable's occurrences among all clauses, which is how we implement it. Alternatively, all values can be initialized with a score of zero to only use knowledge gained during an ongoing search. The main idea of the VSIDS heuristic is to favor variables which took part in recent conflict analyses. In order to do so, the scores of variables that were involved in conflict analyses are increased by a constant value for every i-th conflict. The parameter i is usually set to one. Furthermore, all scores are periodically divided by a constant value, *e.g.*, by two, to favor variables that occurred in recent conflict analyses. Note that it is also possible to store values as described above for each variable with a specific polarity. As first tests did not show any performance improvement but rather drawback, we decided to store values for variables only and initially assign decision variables a positive polarity.

Biere proposed an improvement of the VSIDS heuristic called the exponential variable-state independent decaying sum (EVSIDS) [90]. The heuristic adds f^{-i} to each variable's score at each i-th conflict instead of a constant value. Here, f is a float between zero and one which is usually around 0.9 [90]. This adaption favors variables occurring in recent conflict analyses in the long run and thus does not require worsening scores as is done in the VSIDS heuristic. This is a benefit since it prevents updating heuristic values, which is additional overhead. Biere and Fröhlich further proposed a heuristic called the average conflict-index decision score (ACIDS) [195]. Here, a heuristic score s is updated by adding $(s + i)/2$ at each i-th conflict. Similar to the EVSIDS heuristic, this heuristic favors variables that occurred in recent conflict analyses but does not grow as fast as the EVSIDS heuristic.

While many other heuristics have been built to improve the VSIDS heuristic, Biere and Fröhlich have shown empirically that the EVSIDS heuristic can perform as well as other heuristics in practice [195]. We thus decided to use the EVSIDS heuristic in our SAT solver by default. Furthermore, we achieved better results when only increasing the scores of variables occurring in a computed backjump clause instead of all variables that occurred during the conflict analysis. However, this generally depends on the respective problem and can therefore be set using an option.

Restarts with Phase Saving

The decision for the next variable to assign during SAT solving is guided by a heuristic and thus not necessarily the best decision for all problem instances. In order to recover from bad branching decisions, modern SAT solvers implement restart policies for which the solver backjumps to level 0 in the search tree. Here, the crucial point is to decide how often a search should be restarted to guarantee converging to a solution.

Audemard and Simon [91] proposed a restart policy that includes knowledge gained during a search by using the literal block distance of learned clauses (see Section 6.5.3). The idea is to restart a search if new learned clauses do not provide much new knowledge. This is implemented by comparing a current short-term average LBD score with a long term average LBD score. In order to prevent restarting right before a solution would have been found, the authors further suggest tracking the size of the stack of variable assignments. The idea is to recognize if a partial assignment is considerably closer to a model than was the case for any prior partial assignment [91].

Pipatsrisawat and Darwiche further observed that frequent restarts can decrease the performance of SAT solving in some cases [92]. To counter this, the authors suggested a partial component caching scheme for SAT solvers [92] which we implement as well. Here, all variable assignments made by decisions are cached. A SAT solver then picks the cached polarity first when deciding to branch on a variable. This guides the search in a similar direction than before and prevents solving components of a formula again. If no polarity is cached for a variable, the SAT solver uses the implemented heuristic that assigns a polarity. We implement phase saving by asserting and retracting facts in Prolog to cache variable assignments made by decision.

6.5.4. SMT Solving

The variables assigned in a (partial) model of a Boolean abstraction are conjoined and translated to first-order logic using the function B2T defined in Section 6.5.2. Afterward, the derived SMT formula is solved by one or more theory solvers.

Early Pruning

One bottleneck for performance is to wait for the SAT solver to find a (partial) model before sending formulas to the theory solvers. The implementation spends unnecessary time in the SAT solver in cases where a theory solver can already decide for unsatisfiability using a partial assignment. One important implementation detail is thus to send a constraint to a theory solver as soon as its corresponding Boolean variable is assigned by the SAT solver, this is called early pruning [196]. Theory solvers need to be set up incrementally for early pruning, which is possible with ProB's constraint solver. We implement early pruning by using coroutines in Prolog for each Boolean variable which abstracts a B formula and use ProB's constraint solver as the only theory solver by default. Such a coroutine is defined to be triggered if the corresponding Boolean variable is set to either true or false. In this case, the corresponding B formula or its negation is incrementally added to ProB's constraint solver. We ensure that the effect of coroutines as well as incrementally adding constraints is undone on backtracking, which is simple due to the nature of Prolog being based on backtracking.

One implementation detail is that we connect the SAT variables with the theory solver after possible unit propagations on level 0 of the search tree since these variables can be propagated to the theory solver directly. If we connect the SAT and theory solver before the first unit propagations, we would add the additional overhead of registering predicates in the theory solver for that the solver assumes that they can be either true or false although their truth value is already established.

We refer to the phase of incrementally adding constraints to ProB's constraint solver as its deterministic propagation phase. Here, the solver is already able to identify theory conflicts but does not ground domains to find an exact solution. It is therefore possible that a conflict is only recognized after grounding the domains of all variables. We enter the grounding phase of ProB's constraint solver if the SAT solver reports satisfiability for the Boolean abstraction of the input formula. The SMT solver has found a model if the SAT solver and the theory solver report satisfiability. If a contradiction is found by a theory solver (theory conflict), the assignment of SAT variables can be used as a conflict clause for conflict-driven clause learning as described in Section 6.5.3.

Unsatisfiable Core

When learning from a theory conflict, not necessarily all assigned variables contribute to the actual conflict. In order to learn strong clauses from theory conflicts, it is important to find an unsatisfiable core of a theory conflict before translating it to propositional logic. In particular, a minimal unsatisfiable core is desired. A locally minimal unsatisfiable core of a formula is a subformula which still describes the contradiction but cannot be reduced any further without making it satisfiable. The globally minimal unsatisfiable core of a formula is the smallest formula of all local minima.

One key feature of ProB is that it retains the well-definedness of unsatisfiable cores according to the B language. This is important in the context of clause learning from B formulas since a theory solver would otherwise throw a well-definedness error or would not be able to decide for the satisfiability of a formula when learning a not well-defined clause. For instance, learning a unit clause which corresponds to a theory formula that divides by zero would result in a well-definedness error.

ProB implements a technique to find an unsatisfiable core by gradually removing subformulas from the end of an unsatisfiable formula. Each derived formula is successively checked for satisfiability by ProB's constraint solver. If a derived formula is satisfiable, we know that the removed subformula is definitely part of the unsatisfiable core. Otherwise, we can remove this subformula from the unsatisfiable core since it does not contribute to the unsatisfiability of the overall formula. If removing a subformula results in a well-definedness error, we know that we have to keep this subformula in order to ensure the well-definedness of the unsatisfiable core. Furthermore, it can be the case that the constraint solver is not able to decide for the satisfiability of a derived formula within a reasonable amount of time. We thus use a small solver timeout (25ms) to decide for the satisfiability of a derived formula to prevent exceeding the predefined solver timeout, and keep a subformula if ProB's constraint solver cannot decide for the satisfiability in time. In this case, an unsatisfiable core might contain a subformula that does not contribute to the unsatisfiability of the formula.

We deem it to be sufficient for conflict analysis to compute a locally minimal unsatisfiable core instead of a global minimum to save performance.

Another aspect to consider when learning from theory conflicts is the time that a theory conflict is detected. When learning from Boolean conflicts in the SAT solver, the last propagated variable is always part of the actual conflict. Yet, this is not necessarily the case for theory conflicts, especially when using ProB's constraint solver which has to consider the well-definedness of constraints. We reify Boolean variables with ProB's constraint solver as explained in Section 6.5.4, which is part of the deterministic propagation phase. Here, domains are not necessarily enumerated to prevent exceeding the predefined solver timeout. When propagating a constraint, it can be the case that other constraints are sent to the solver afterward which allow for a stronger propagation. For instance, consider the propagation of the two constraints $r \in 1..n \to \mathbb{Z}$ and $x \in \mathrm{dom}(r)$. The constraint solver is currently not able to decide for the satisfiability of these constraints since the domain of r is unknown. After propagating another constraint $n = 0$, the constraint solver is able to do so.

A conflict may not be detected until the domains of variables are grounded as described in Section 6.5.4, especially when constraints entail a well-definedness condition such as a function application. It can thus be the case that the Boolean propagation from the SAT solver that caused the propagation of the conflicting theory constraints was not on the last decision level. For instance, consider the two constraints from above but let n be an element of the interval $0..2$. The constraint solver can now first decide for the satisfiability of all constraints after the variable n has been grounded. Yet, there can be an arbitrary amount of other propagations in the SAT solver which do not affect these constraints but are necessary to solve the whole Boolean formula. We thus do not necessarily consider the last decision level of the SAT solver as the level where a theory conflict occurred if the conflict is detected in the grounding phase after the SAT solver has found a solution. Instead, we compute the maximum decision level of the variables that are part of the unsatisfiable core of the theory conflict, which is then used for conflict-driven clause learning.

Theory Propagation

Until now, the only knowledge passed from the theory solver to the SAT solver is gained by theory conflicts. Yet, a theory solver might deduce formulas which provide new knowledge for the SAT solver as well. Sending such formulas to the SAT solver is called theory propagation [98, 197]. We extend the coroutines which are set up in Prolog for each SAT variable to be triggered if the corresponding SMT formula becomes true as well. For instance, let $\Phi := x < 0 \land x < 1$ be a B predicate and $\mathrm{A} \land \mathrm{B}$ be the corresponding propositional logic formula with $\mathrm{A} \equiv x < 0$ and $\mathrm{B} \equiv x < 1$. We set up two coroutines for the SAT variables A and B that are reified with the corresponding theory formulas. The SAT solver possibly starts with setting A to true which triggers the corresponding coroutine to send the formula $x < 0$ to ProB's constraint solver. In this case, the solver is able to deduce that the formula $x < 1$ has to be true as well. This triggers the other coroutine which now propagates knowledge from the theory solver to the SAT solver by setting the variable B to true. Note that one important aspect of theory propagation is to only deduce constraints which already occur in the original formula. Otherwise, new SAT variables would need to be introduced for each new SMT formula. This unnecessarily increases the search space since no new knowledge can be gained by the SAT solver from such formulas. For instance, we do not want to deduce $x < 2$ for the above example.

The Prolog code that is responsible for the theory propagation of a "less than" comparison in ProB's constraint solver can be seen in Listing 6.1. The entry point is the Prolog predicate `check_arith_op/4`. First, a coroutine `check_lt/3` is set up which is suspended as long as the first and second, or the second and third arguments are

Listing 6.1: Prolog code for the implementation of the theory propagation for a "less than" comparison in ProB's constraint solver using coroutines.

```
 1  check_arith_op('<',X,Y,Res) :-
 2    check_lt(X,Y,Res),
 3    ( nonvar(Res) -> true
 4    ; (X#<Y) #<=> R01, prop_pred_01(Res,R01)).
 5
 6  :- block prop_pred_01(-,-).
 7  prop_pred_01(A,B) :- B==1, !, A=pred_true.
 8  prop_pred_01(pred_true,1).
 9  prop_pred_01(pred_false,0).
10
11  :- block check_lt(-,?,-), check_lt(?,-,-).
12  check_lt(X,Y,Res) :- nonvar(Res),!,
13     ( Res=pred_true -> lt_direct(X, Y)
14     ; lt_equal_direct(Y,X)).
15  check_lt(X,Y,Res) :- X<Y, !, Res=pred_true.
16  check_lt(_,_,pred_false).
```

variables (indicated by the dash in the block declaration). If the third argument is nonvariable, we know the truth value of the integer comparison and enforce that the Prolog variable X is lower than Y (line 13) or that Y is lower than or equal to X (line 14). Otherwise, both integer variables have concrete values and the integer comparison is checked to set the result variable to either true (line 15) or false (line 16). In line 4 of Listing 6.1 a CLP(FD) constraint is set up to reify (`#<=>`) the integer comparison with a two-valued result variable which is either 0 (false) or 1 (true). Afterward, a coroutine `prop_pred_01/2` is set up to be triggered if one of its two arguments becomes nonvariable. The first argument corresponds to the overall result of the integer comparison returned in `check_arith_op/4` (line 1) while the second argument corresponds to the reified two-valued variable. Here, the actual theory propagation is implemented which reifies the truth value of the integer comparison in CLP(FD) with the integer comparison's result in ProB's constraint solver. In our example above, `check_arith_op/4` is called for each integer comparison. After propagating $x < 0$ to ProB's constraint solver, CLP(FD) enforces the constraint `X #< 1`, which has been set up in line 4 of Listing 6.1, to be true as well. This triggers the corresponding two-valued variable to be set to 1 which unblocks the coroutine `prop_pred_01/2` that now sets the overall result of the integer comparison $x < 1$ to true. Thus, this truth value has been propagated by ProB's constraint solver which now triggers the coroutine that is connected to the SAT variable to set this variable to true.

Explaining Theory Propagations

It can be the case that a conflict clause contains a SAT variable which has been propagated by a theory solver when using theory propagation with CDCL. In the SAT solver, this propagation is the same as a unit propagation since the theory propagation is a logical consequence of the current assignment of variables. In order to analyze a conflicting assignment, we need to know which clause led to the unit propagation of a variable as explained in Section 6.5.3. However, PROB's constraint solver does not provide an explanation for a theory propagation.

We thus implement a technique to find explanations for B formulas by computing an unsatisfiable core. Let ϕ be an SMT formula which has been propagated by a theory solver, and Φ be the SMT formula corresponding to the current partial assignment without ϕ. In order to explain the propagation of ϕ, we compute the unsatisfiable core of $\Phi \wedge \neg\phi$. For instance, let $\Phi := x < 10 \wedge x > 1$ and consider the formula $x < 10 \wedge x > 1 \wedge x > 0$. Furthermore, assume that $x < 10$ and $x > 1$ have been propagated by the SAT solver while $\phi := x > 0$ has been propagated by a theory solver after $x > 1$ has been set. The unsatisfiable core of the formula $x < 10 \wedge x > 1 \wedge \neg x > 0$ is $x > 1 \wedge \neg x > 0$. By removing the negated theory propagation, we can conclude that $x > 1$ is the explanation for the theory propagation of $x > 0$. We are then able to add the Boolean abstraction of the negated theory formula $\neg x > 1 \vee x > 0$ to the SAT solver which now enforces a unit propagation corresponding to the theory propagation.

Computing theory explanations by default is too expensive and not necessary in most cases. We thus explain theory propagations lazily to improve performance, *i.e.*, we only explain a theory propagation if conflict-driven clause learning requires an explanation for conflict analysis.

Well-Defined SMT Solving

The B language has many operators that entail a well-definedness condition. For instance, a well-defined function application requires that the applied element is in the domain of the function. Not well-defined constraints are neither true nor false in PROB's constraint solver. As explained in Section 6.5.4, the SMT solver sends single B predicates to the theory solver via constraint reification. If a predicate is sent to PROB's constraint solver which is not well-defined, the SMT solver possibly reports unsatisfiability for satisfiable constraints. The reason is that the SAT solver requires a variable to be either satisfiable or unsatisfiable, which is not the case if a reified predicate is not well-defined. For instance, consider the B formula $(0 \in \mathrm{dom}(\varnothing) \wedge (\varnothing(0) = a \Rightarrow \neg(\varnothing(0) < a))) \vee b = 0$. Note that functions in B are relations, which in turn are sets of pairs. The empty set can thus be considered a function. While the function application in this formula is not well-defined, the whole formula is well-defined since the not well-defined function application is guarded by the corresponding well-definedness condition $0 \in \mathrm{dom}(\varnothing)$. In B, constraints ensuring the well-definedness are placed before the operators that entail the well-definedness condition. Yet, this structure usually gets lost when transforming a B formula to conjunctive normal form as is required for SMT solving. Further, the SAT solver might propagate a predicate that entails a well-definedness condition although the well-definedness is not ensured yet. For instance, the SMT solver might start with propagating the single predicate $\varnothing(0) = a$. In this case, PROB's constraint solver would neither confirm the satisfiability nor the unsatisfiability of this predicate since it is not well-defined. This results in finding a spurious counterexample in the SAT solver. For the above example, the solver would report unsatisfiability for the whole formula after evaluating the left-hand side of the disjunction although $b = 0$ is a solution.

We counter this issue by adding constraints that ensure the well-definedness for each predicate that is reified with a SAT variable and entails a well-definedness condition. This ensures that ProB's constraint solver is able to decide for the satisfiability of a predicate. For instance, for the above formula we would usually introduce four SAT variables for the unique predicates occurring in the formula, *i.e.*, $\mathrm{A} \equiv 0 \in \mathrm{dom}(\varnothing)$, $\mathrm{B} \equiv \varnothing(0) = a$, $\mathrm{C} \equiv \varnothing(0) < a$, and $\mathrm{D} \equiv b = 0$. In order to ensure the well-definedness of each predicate independently of the whole formula, we now adapt the predicates that are reified with ProB's constraint solver for the second and third predicate to be $0 \in \mathrm{dom}(\varnothing) \land \varnothing(0) = a$ and $0 \in \mathrm{dom}(\varnothing) \land \varnothing(0) < a$.

Additionally, we enforce the well-definedness in the SAT solver by adding an implication for each predicate entailing a well-definedness condition to the conjunctive normal form. This ensures the unit propagation of a well-definedness condition as soon as the predicate entailing the condition is propagated by the SAT solver. For the above example, this results in adding the implications $\mathrm{B} \Rightarrow \mathrm{A}$ and $\mathrm{C} \Rightarrow \mathrm{A}$. If an input formula is not well-defined, we introduce a new SAT variable for corresponding well-definedness conditions. For instance, the above formula is not well-defined without the predicate $0 \in \mathrm{dom}(\varnothing)$. We then proceed in the same way as described above but introduce a new SAT variable corresponding to the variable A, which is not present in the original formula. ProB's SMT solver thus always solves well-defined formulas.

6.6. Additional Theory Solver

In SMT solving, theory constraints are usually sent to a single dedicated theory solver. Alternatively, constraints can be sent to several solvers which support the same theory to improve performance by using the result of the fastest solver. The only prerequisite is that a theory solver can be used incrementally as described in Section 6.5.4.

Empirical results have shown that ProB's CLP(FD) backend sometimes lacks performance for unsatisfiable constraints over unbounded integer domains. We thus decided to integrate an alternative theory solver for the integer difference logic (IDL) [198, 199] which does not enumerate domains but uses a solver based on graphs and negative cycle detection [4]. We do not have evidence that such constraints often occur in B but see that this alternative technique can improve constraint solving over unbounded integer domains compared to CLP(FD). Moreover, this solver can be used as a fallback in cases where ProB's constraint solver generates a virtual timeout.

In the following, we describe the theoretical foundation of the implemented constraint solver for IDL constraints and its integration in ProB's SMT solver.

6.6.1. Integer Difference Logic Solver

Integer difference logic [198, 199] has shown to be useful, e.g., for reasoning about clocks in timed systems. Atomic IDL constraints are of the form $v_i - v_j \leq c$, where v_i and v_j

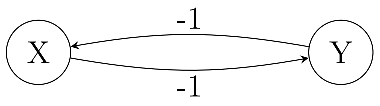


Figure 6.6.: A graph representation of the integer difference logic constraint $x \in \mathbb{Z} \wedge x > y \wedge y > x$ as suggested by Wang et al. [4]. The graph contains a negative cycle which means that the corresponding constraint is unsatisfiable. The constraints corresponding to the edges of the negative cycle are the unsatisfiable core.

are integer variables and c is a constant integer value. Conjunction and negation are the only admitted logical operators.

Many integer constraints can be rewritten to integer difference logic (see Appendix B for a complete list of our rewriting rules). This might require the introduction of an artificial variable for the constant 0. For instance, the integer constraint $x < 1$ can be rewritten to $x - \text{zero} \leq 0$, where zero is the artificial variable.

Wang et al. presented a solver for integer difference logic based on weighted directed graphs with an algorithm for incremental negative cycle detection [4].

Each node of a graph represents an integer variable. An edge (v_j, v_i) in a graph with weight c describes the constraint $v_i - v_j \leq c$, while the negated constraint is described by the edge (v_i, v_j) with weight $-c-1$. A difference logic constraint system is satisfiable if its corresponding graph does not contain a negative cycle. If this is the case, the shortest path to a variable yields its solution. Otherwise, the conjunction of constraints corresponding to the edges of the negative cycle is an unsatisfiable core. As can be seen, this technique provides an unsatisfiable core for unsatisfiable formulas without further computations. This makes this technique especially suited for conflict-driven clause learning, where unsatisfiable cores have to be computed when a theory conflict occurs (see Section 6.5.4).

The authors further propose an incremental decision procedure based on the Bellman-Ford algorithm [200, 201]. The algorithm uses the fact that any created cycle must use the recently added edge [4].

For instance, consider the B constraint $x \in \mathbb{Z} \wedge x > y \wedge y > x$. ProB's constraint solver is not able to solve this constraint with its default CLP(FD) backend since the domains of x and y cannot be narrowed down. The new theory solver first rewrites constraints to integer difference logic if possible. The above constraint is rewritten to $y - x \leq -1 \;\wedge\; x - y \leq -1$. Afterward, we create nodes for the variables x and y and edges for the two constraints as can be seen in Figure 6.6. It can be easily seen that the graph contains a negative cycle between the variables x and y, *i.e.*, the path $[(x, y), (y, x)]$. This means that the constraint is not satisfiable and the conjunction of constraints corresponding to the edges of the negative cycle are an unsatisfiable core. For our example, the unsatisfiable core is $x > y \wedge y > x$.

6.6.2. Theory Solver Integration

We decided to use the aforementioned theory solver in addition to ProB's constraint solver, *i.e.*, we send integer difference logic constraints to both theory solvers. For this, we extend the coroutines which are set up for each reification between a SAT variable and theory constraint to check if the constraint can be translated to integer difference logic and distribute constraints accordingly.

We observed severe performance degradation when not all integer constraints can be translated to IDL. For instance, consider the formula $a - b \leq -1 \land a * b > 10000$. The first conjunct can be sent to both theory solvers while the second one cannot be rewritten to IDL. The graph-based solver for integer difference logic finds the partial model $a = 1 \land b = 2$ which is not accepted by ProB's constraint solver because of the constraint $a * b > 10000$. The graph-based solver would now enumerate nearly 10,000 partial models which are refuted by ProB's constraint solver until finding a solution, *e.g.*, $a = 1 \land b = 10001$.

To solve this issue, we decided to integrate the integer difference logic solver as follows: IDL constraints are sent to both theory solvers. In case the graph-based solver reports unsatisfiability, the unsatisfiable core is extracted and used as a conflict clause. If the SAT solver and both theory solvers report satisfiability in the deterministic propagation phase, we first propagate the partial model found by the graph-based IDL solver to ProB's constraint solver. Hereby, we want to prevent a possible (virtual) timeout when grounding domains. SMT solving is finished if this solution is accepted. Otherwise, we do not backtrack in the graph-based solver but let ProB's constraint solver enumerate a solution. We fall back to the graph-based solver only if ProB's constraint solver fails to find a solution because of generating a virtual timeout due to unbounded integer domains.

6.7. Empirical Evaluation

In the following, we present an empirical evaluation of the new integration of Z3 in ProB including the new translation from B to SMT-LIB as well as the direct implementation of SMT solving in ProB.[2]

6.7.1. Integration of Z3

We split the evaluation of the integration of Z3 in ProB in three categories. First, we focus on the downsides of our employed translation of selected language constructs which we deem to be responsible for a possibly bad performance when solving constraints. Second, we present selected constraints for which the integration of Z3 is superior to ProB's constraint solver regarding constraint solving to emphasize specific strengths.

[2]The benchmarks can be found in the following Github repository to reproduce the results: `https://github.com/Joshua27/prob_smt_benchmarks`

Third, we evaluate the performance of our translation using a variety of benchmarks from bounded model checking.

Weaknesses of the Integration of Z3

The weaknesses of the integration of Z3 are mainly caused by the employed encoding of sets. Most of B's set theoretic operators are not supported by SMT-LIB such as computing a power set or the cardinality of a set. As discussed in Section 6.3.1 and Section 6.4.1, this can lead to involved quantified constraints for which Z3 is not able to find a solution. We thus employ several rewriting rules and a preprocessing phase to prevent sending quantified formulas to Z3 if this is not necessary. The benefit of this preprocessing is discussed in the following.

Finite Sets. The former and new translation from B to SMT-LIB both support infinite sets. It could be shown by Krings and Leuschel [48] that Z3 is able to solve a variety of B constraints over infinite domains which ProB's constraint solver is not able to solve, especially when a formula is a contradiction. However, the support for infinite domains leads to involved translations for finite set constraints such as the minimum, maximum or the cardinality of a finite set. For instance, the current translation searches for a total bijection mapping sets to their cardinalities to compute the cardinality of a set [48]. A total bijection is rewritten using B quantifiers before the translation to SMT-LIB.

Since Z3 lacks performance when solving quantified formulas, Z3 often fails to find a solution for translated B constraints using set cardinalities. For instance, Z3 is not able to solve the translation of $q \in 1 \,..\, 3 \rightarrow 1 \,..\, 3 \wedge \mathrm{card}(\mathrm{ran}(q)) = 3$. With the use of the rewriting rule for the cardinality of range constraints defined in Section 6.4.1, Z3 is able to solve the constraint in several ms as is ProB's constraint solver. The rewriting rule replaces the cardinality constraint with $q(1) \neq q(2) \wedge q(1) \neq q(3) \wedge q(2) \neq q(3)$. Of course, not all cardinality constraints can be replaced by equivalent constraints and remaining quantifiers are still one of the main culprits for a possibly bad performance of the presented translation from B to SMT-LIB. For instance, the integration of Z3 needs around 20 s to find a solution for the predicate $x \in \mathbb{P}(\mathbb{Z}) \wedge \mathrm{card}(x) > 10$, which can be solved by ProB's constraint solver in several ms. The reason is that Z3 spends a lot of time to solve the existentially quantified total bijection that is introduced for the translation of the set cardinality constraint as described in Section 6.3.1.

The translation of set constraints to SMT-LIB such as card, max or min could be improved by focusing on finite sets only, *e.g.*, as presented by Plagge and Leuschel [46] for B by translating to Kodkod [45] or by Konnov et al. [202] for TLA$^+$ [41] by translating to SMT-LIB [182]. Yet, we would then lose the ability to reason over B constraints involving infinite sets.

Contradictions. Translations which result in quantifiers in SMT-LIB can become too involved to be solved by Z3. In some cases this means that Z3 cannot find contradictions in a translated SMT-LIB formula which are obvious in the corresponding B formula. For instance, Z3 is not able to find the contradiction in the formula $r \in \mathbb{Z} \rightarrowtail\mathrel{\mkern-14mu}\rightarrow \mathbb{Z} \wedge r \notin \mathbb{Z} \rightarrowtail\mathrel{\mkern-14mu}\rightarrow \mathbb{Z}$.

Here, both bijections are translated as quantified formulas in SMT-LIB leading Z3 to report unknown. We are able to detect the contradiction by abstracting the formula to propositional logic and using a SAT solver as described in Section 6.4.2. In particular, we lift negations from B operations before the abstraction which results in $\mathrm{A} \wedge \neg \mathrm{A}$ where $\mathrm{A} \equiv r \in \mathbb{Z} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{Z}$. It can be seen that no translation to SMT-LIB is necessary in such cases. Such constraints do often occur in bounded model checking, where invariants are negated to check for counterexamples.

Strengths of the Integration of Z3

Weaknesses of the ProB constraint solver are often caused by the use of unbounded integer domains. One motivating example which speaks in favor of the integration of Z3 is the constraint $x > y \wedge y > x$. ProB's constraint solver is not able to solve this constraint with its default CLP(FD) backend since the integer domains of x and y cannot be narrowed down. Although ProB's constraint solver is able to solve this constraint by using an additional backend that implements custom constraint handling rules (CHR), the example shows a benefit of using Z3 for unbounded integer domains, in particular for linear integer arithmetic. For example, Z3 is able to solve the constraint $\forall(x, y).(x \in \mathbb{Z} \wedge y \in \mathbb{Z} \Rightarrow \exists z.(x - z = y))$ while ProB's constraint solver is not. The constraint is taken from the 14th SMT competition for quantified integer difference logic [203]. Another constraint which cannot be disproven by ProB's constraint solver but using the integration of Z3 is $\neg((\mathrm{s2} = s \cup \{0\} \wedge \mathrm{s3} = s \cup \{1\} \wedge \mathrm{s4} = \mathrm{s2} \cup \{1\} \wedge \mathrm{s5} = \mathrm{s3} \cup \{0\}) \Rightarrow \mathrm{s4} = \mathrm{s5})$, which stems from a computation that occurred during partial order reduction for B. Again, both constraints contain unbounded sets of the integers which cannot be narrowed down by ProB's constraint solver. The constraints further indicate that this issue affects model finding as well as the disproving of formulas.

We further observed strengths of Z3 regarding the disproving of constraints involving infinite relations. For instance, the integration of Z3 is able to solve the constraint $f \in \mathbb{N} \nrightarrow \mathbb{N} \wedge x \in \mathbb{N} \wedge g = f \Leftarrow\!\!\!+ \{x \mapsto x + 1\} \wedge \neg(g \in \mathbb{N} \nrightarrow \mathbb{N})$ which cannot be solved by ProB's constraint solver. Furthermore, this constraint can only be solved when using the new translation from B to SMT-LIB which uses Z3's lambda functions.

The integration of Z3 is also able to solve several constraints faster than ProB's constraint solver. Such constraints do not necessarily involve unbounded domains but are related to the enumeration of domains as performed by ProB's constraint solver. For instance, the integration of Z3 is able to find a model for the constraint $f = \lambda x.(x \in 1 \mathinner{..} n \mid x + 1) \cup \{n + 1 \mapsto (n/2)\} \wedge x = f[x] \wedge x \neq \varnothing \wedge n = 20$ in around 0.17 s while ProB's constraint solver is not able to solve the constraint within 60 s. The reason is that CLP(FD) enumerates many values before finding a solution which does not seem to be the case for Z3. It should be noted that the aforementioned strengths of Z3 are not related to SMT solving but rather to its strong theory solvers, especially for linear integer arithmetic.

Table 6.1.: Bounded model checking (BMC) constraints from TLA$^+$ benchmarks compiled by Konnov et al. [202] as well as classical B and Event-B benchmarks compiled by Krings and Leuschel [114, 153]. BMC uses a bound of 25 and sets up 26 constraints for depth $0 \ldots 25$ for each benchmark. We further state the mean amount of independent components for all constraints of a benchmark.

			ProB-Z3				
No.	Name	ProB	(axiomatic)	(constructive)	(parallel)	(parallel & decomposed)	∅ Components
1	Prisoners-4	**8 / 2284 s**	0 / 51 s	0 / 49 s	0 / 53 s	0 / 727 s	6
2	Bakery	**2 / 2888 s**	0 / 973 s	1 / 94 s	1 / 970 s	1 / 1267 s	1
3	Paxos-3	**2 / 2888 s**	0 / 110 s	0 / 75 s	0 / 126 s	0 / 444 s	2
4	SimpleTwoPhase	**26 / 0.31 s**	25 / 0.96 s	25 / 0.63 s	25 / 0.60 s	25 / 1 s	29
5	TravelAgency	**15 / 1342 s**	0 / 183 s	0 / 170 s	0 / 184 s	0 / 281 s	10
6	LargeBranching	**26 / 0.99 s**	26 / 81 s	26 / 58 s	26 / 134 s	26 / 149 s	2
7	SearchEvents	3 / 2761 s	2 / 2890 s	**20 / 836 s**	20 / 876 s	20 / 842 s	6
8	ABZ16_m4	**26 / 3 s**	26 / 18 s	26 / 17 s	26 / 17 s	26 / 18 s	11
9	ABZ16_m5	**26 / 4 s**	25 / 23 s	26 / 24 s	26 / 23 s	26 / 25 s	11
10	ABZ16_m6	**25 / 8 s**	18 / 1021 s	5 / 77 s	18 / 998 s	18 / 1075 s	12
11	ABZ16_m7	**26 / 8 s**	19 / 888 s	5 / 46 s	19 / 933 s	19 / 899 s	13
12	R0_GearDoor	**26 / 0.36 s**	26 / 4 s	26 / 3 s	26 / 3 s	26 / 3 s	1
13	R1_Valve	**26 / 0.87 s**	26 / 8 s	26 / 8 s	26 / 8 s	26 / 9 s	5
14	R2_Outputs	**26 / 2 s**	26 / 15 s	26 / 15 s	26 / 15 s	26 / 16 s	11
15	R3_Sensors	12 / 1727 s	**26 / 32 s**	**26 / 32 s**	**26 / 32 s**	26 / 33 s	17
16	R4_Handle	**5 / 2560 s**	1 / 497 s	2 / 350 s	2 / 565 s	2 / 837 s	17
17	R5_Switch	9 / 2142 s	**26 / 237 s**	26 / 251 s	26 / 274 s	26 / 344 s	25
18	R6_Lights	7 / 2344 s	**26 / 344 s**	26 / 385 s	26 / 382 s	26 / 441 s	31
19	Lightbot	3 / 2762 s	2 / 1117 s	**11 / 1878 s**	11 / 1883 s	11 / 1948 s	12
20	M0_AAI	2 / 2904 s	26 / 219 s	**26 / 53 s**	26 / 63 s	26 / 74 s	6
21	M0_AAT	3 / 2761 s	26 / 204 s	**26 / 57 s**	26 / 70 s	26 / 76 s	6
22	M0_AOO	3 / 2761 s	26 / 13 s	**26 / 12 s**	26 / 16 s	26 / 23 s	3
23	M0_VOO	3 / 2761 s	**26 / 21 s**	**26 / 21 s**	26 / 24 s	26 / 31 s	3
24	M0_VVI	3 / 2761 s	26 / 224 s	**26 / 52 s**	26 / 61 s	26 / 72 s	6
25	M0_VVT	3 / 2779 s	26 / 212 s	**26 / 53 s**	26 / 63 s	26 / 71 s	6
26	M1_AOOR	3 / 2762 s	**26 / 36 s**	26 / 37 s	26 / 42 s	26 / 54 s	13
27	M1_VOOR	3 / 2761 s	**26 / 32 s**	**26 / 32 s**	26 / 37 s	26 / 44 s	12
28	M2_AAI	3 / 2761 s	26 / 197 s	**26 / 50 s**	26 / 60 s	26 / 65 s	8
Total		325 / 48 737 s	534 / 9651 s	537 / 4736 s	**564 / 7913 s**	564 / 9869 s	
			Solved constraints / Runtime s				

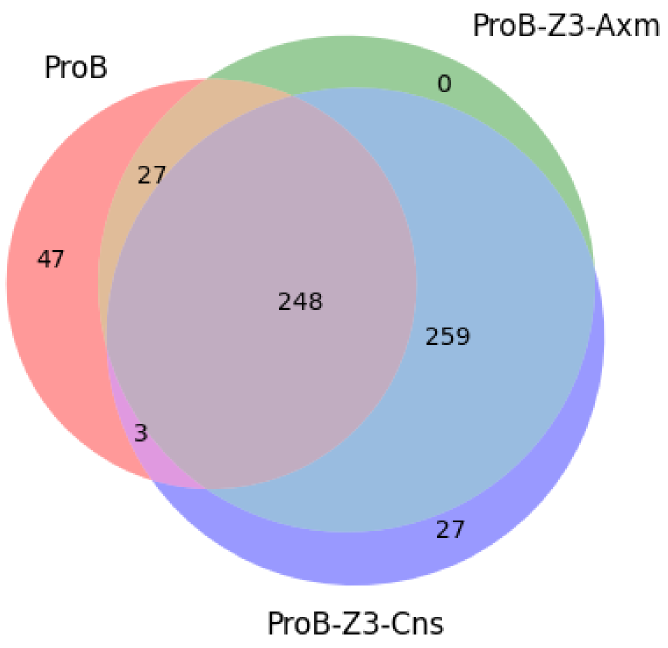


Figure 6.7.: A Venn diagram to visualize and compare the amount of BMC constraints that can be solved by ProB's constraint solver and Z3 using the former axiomatic and the new constructive translation from B to SMT-LIB as can be seen in Table 6.1.

Bounded Model Checking

For a more sophisticated performance evaluation, we decided to use constraints from bounded model checking (BMC). In particular, we use the monolithic bounded model checking implementation [153] of ProB which sends a single formula to a selected constraint solving backend. The goal of bounded model checking is to verify a system's properties symbolically by searching for a counterexample considering a maximum amount of successive state changes. For instance, let I be a B or Event-B machine's invariant, v be a machine's state of variables, $\text{init}(v)$ be the machine initialization, and op be the only machine operation. Further, let $\text{BA}_{\text{op}}(v, v')$ be the before-after predicate applying the operation to the variables in v and assigning the results to the variables in v'. This corresponds to a state change in the machine but represented as a predicate using fresh variables v'. For a bound of 1, we set up the BMC constraint $\text{init}(v) \land \text{BA}_{\text{op}}(v, v') \land \neg I'$, where I' is the machine invariant referring to the variables in v'. If the constraint is satisfiable, its solution corresponds to a machine state that violates the machine invariant and can be reached after a single state change. For our benchmarks, we check the B and Event-B machines from a bound of 0 to 25, *i.e.*, we solve 26 constraints for each machine. We compare the amount of constraints that can be solved by a specific solver as well as the time needed to decide for the satisfiability of all constraints. That means, the presented runtimes are the sum of the times needed to solve all 26 constraints. We use a maximum solver timeout of 2 min for each constraint and compare the ProB constraint

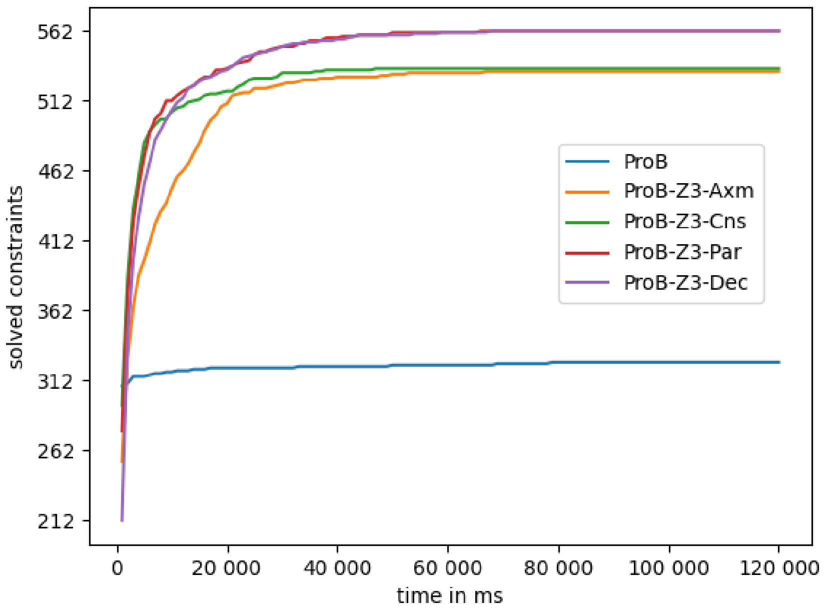


Figure 6.8.: A visualization of the BMC benchmark results presented in Table 6.1 showing the amount of constraints that can be solved by a constraint solver within a specific amount of time. We compare ProB's constraint solver and the four different configurations of the integration of Z3 in ProB. The smallest constraint solver timeout is 1000 ms.

solver, its integration of Z3 using the former translation [48], the new translation as described in Section 6.4.1, the parallel integration of Z3 as described in Section 6.4.2, as well as the parallel integration of Z3 that iteratively solves independent components of a constraint as described in Section 6.4.3. We did not investigate the effects of a larger timeout since Z3 rather answers unknown than exceeding the solver timeout.

The evaluated benchmarks can be seen in Table 6.1. We use four TLA$^+$ [41] benchmarks compiled by Konnov et al. [202]. The authors used the benchmarks to evaluate the performance of their symbolic model checker APALACHE for TLA$^+$ which translates to SMT-LIB. We use the translation from TLA$^+$ to B [132] to load TLA$^+$ models in ProB. Unfortunately, the integration of Z3 is not able to solve many constraints of these benchmarks. We thus only use these four benchmarks which already exhibit this trend. Additionally, we use a set of classical B and Event-B benchmarks compiled by Krings and Leuschel [153]. The benchmarks numbered 8 to 11 are taken from a submission to the ABZ 2016 case study [204] by Hoang et al. [205], the benchmarks 12 to 18 from a submission to the ABZ 2014 landing gear case study [206] by Hansen et al. [207], and the benchmarks 20 to 28 from a model of a pacemaker by Méry and Singh [2]. We deem these models to be suited for a performance evaluation since they represent real-world examples. The classical B and Event-B models are correct according to their specification. Thus, all BMC constraints pose a contradiction. Additionally, we use three classical B machines for which a BMC constraint provides a counterexample (benchmarks 5 to 7), *i.e.*, at least one constraint is satisfiable. Besides the amount of solved constraints and runtimes of each solver we also state the mean amount of independent components for each benchmark. The constraints of all benchmarks have an average amount of 417 unique conjuncts or disjuncts and a median amount of 117. The largest constraint contains 3275 unique conjuncts or disjuncts.

The benchmarks were run on a system with an Intel Core I7-8750H CPU (2.2GHz) and 16 GB of RAM using ProB version `1.11.1`, SICStus Prolog version `4.7.0`, and Z3 version `4.8.16`.

First and foremost, the benchmark's evaluation shows that the new constructive translation using Z3's lambda functions improves performance and coverage. Z3 is able to solve many more constraints than is the case for the former axiomatic translation, e.g., for the benchmarks numbered 7 and 19. The 7th benchmark contains constraints that provide a solution while the 19th benchmark does not. This shows that the constructive translation improves performance for model finding as well as the disproving of constraints. Yet, Z3 is also able to solve several constraints only when using the axiomatic translation. For instance, this is the case for the benchmarks numbered 10 and 11. We therefore consider the decision to run two Z3 solvers with both translations in parallel to be justified. Figure 6.7 shows a Venn diagram to compare the amount of constraints that can be solved by a specific solver. It can be seen that Z3 is able to solve 27 constraints only when using the axiomatic translation and 27 constraints only when using the constructive translation. The parallel integration of Z3 in ProB is able to solve 239 constraints that cannot be solved by ProB's constraint solver as can be seen in Table 6.1. Yet, ProB's constraint solver is also able to solve 47 constraints that cannot be solved by Z3.

A visualization of the benchmark results comparing the amount of constraints that can be solved within a specific amount of time is shown in Figure 6.8. It can be seen that all constraint solvers are not able to solve significantly more constraints for a timeout larger than 1 minute. We thus deem the selection of a timeout of 2 min to be justified.

Surprisingly, the decomposition of constraints into independent components neither improves the performance of constraint solving nor allows solving any more constraints than is the case for the default parallel integration. Almost all constraints can be decomposed into several independent components as can be seen in the last column of Table 6.1. As expected, the computation of the independent components and independent Z3 solver calls add some additional overhead. Apparently, the components that pose a contradiction can still not be solved by Z3 as is the case for the whole constraint. We assume that Z3 itself already divides constraints into independent components so that our decomposition does not provide any improvement. The results further show that the fact that Z3 is not able to decide for the satisfiability of a constraint is not necessarily influenced by the size of a constraint but rather by the use of specific operators. This approach is thus probably not worth it to be used in the future.

When comparing the runtimes of the integration of Z3 and ProB's constraint solver, it can be seen that Z3 is able to solve several constraints better. This does not only affect the performance but more importantly the coverage of constraint solving as can be seen in Figure 6.7. For the benchmarks numbered 7, 15 and 17 to 28 the integration of

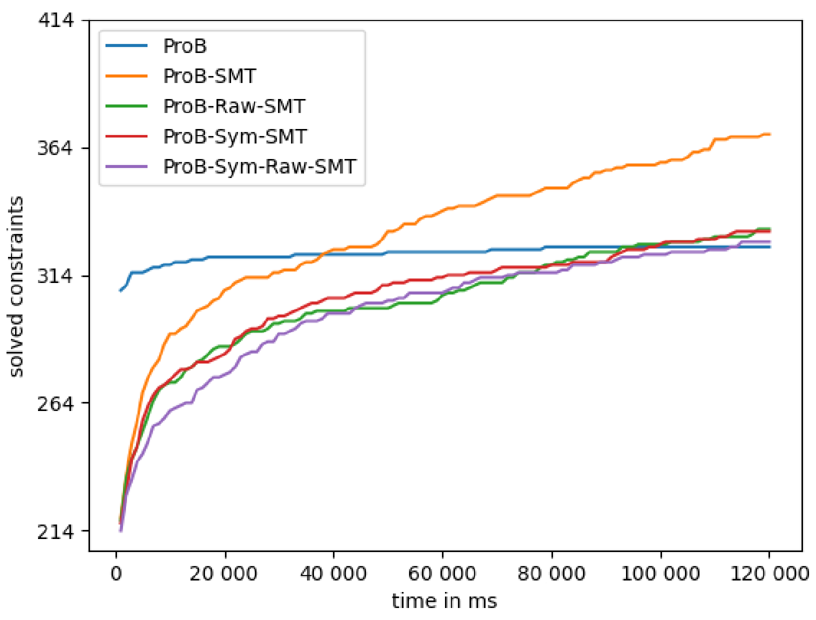


Figure 6.9.: A visualization of the BMC benchmark results presented in Table 6.2 showing the amount of constraints that can be solved by a constraint solver within a specific amount of time. We compare ProB's constraint solver and the four different configurations of ProB's SMT solver. The smallest constraint solver timeout is 1000 ms.

Z3 is able to solve many more constraints than is the case for ProB's constraint solver. For benchmarks 19 to 28, ProB's constraint solver is not able to narrow down the domains to find a contradiction for most constraints but exceeds the predefined solver timeout. The machines contain several unbounded domains over the natural numbers and different integer arithmetic constraints. It can be the case that Z3 is able to solve these constraints due to the Boolean abstraction of formulas or due to its strong theory solvers, especially for linear integer arithmetic.

Nonetheless, ProB's constraint solver is also able to solve several constraints better than the integration of Z3. For the benchmarks numbered 1 to 3, 5, and 8 to 11, ProB's constraint solver is able to solve many constraints which cannot be solved by Z3. Here, it should be noted that Z3 often gives up constraint solving (unknown) without exceeding the predefined timeout, *e.g.*, as is the case for the first benchmark.

All in all, it can be seen that the new integration of Z3 and translation from B to SMT-LIB extends the power of ProB's portfolio of constraint solving backends. Since the decomposition of constraints into independent components does not improve performance for our selected benchmarks, we prefer the plain parallel integration of Z3 in ProB.

Table 6.2.: The same set of BMC benchmarks as used in Table 6.1 but comparing the different configurations of ProB's SMT solver with ProB's constraint solver and the new parallel integration of Z3.

			ProB-Z3	ProB			
No.	Name	ProB	(parallel)	SMT	Raw-SMT	Sym-SMT	Sym-Raw-SMT
1	Prisoners-4	8 / 2284 s	0 / 53 s	**10 / 2214 s**	8 / 2344 s	8 / 60 s	8 / 2338 s
2	Bakery	2 / 2888 s	1 / 970 s	16 / 1344 s	17 / 1321 s	**17 / 710 s**	17 / 1541 s
3	Paxos-3	**2 / 2888 s**	0 / 126 s	1 / 12 s	1 / 12 s	1 / 228 s	1 / 229 s
4	SimpleTwoPhase	**26 / 0.31 s**	26 / 0.60 s	26 / 0.56 s	26 / 0.52 s	26 / 1 s	26 / 2 s
5	TravelAgency	**15 / 1342 s**	0 / 184 s	4 / 2861 s	2 / 2882 s	3 / 399 s	4 / 2790 s
6	LargeBranching	**26 / 0.99 s**	26 / 134 s	26 / 4 s	26 / 5 s	26 / 5 s	26 / 6 s
7	SearchEvents	3 / 2761 s	**20 / 876 s**	18 / 1507 s	7 / 2473 s	11 / 1268 s	7 / 2469 s
8	ABZ16_m4	**26 / 3 s**	26 / 17 s	26 / 11 s	26 / 11 s	26 / 9 s	26 / 9 s
9	ABZ16_m5	**26 / 4 s**	26 / 23 s	26 / 11 s	26 / 15 s	26 / 13 s	26 / 12 s
10	ABZ16_m6	**25 / 8 s**	18 / 998 s	5 / 2563 s	13 / 2140 s	5 / 2553 s	11 / 2230 s
11	ABZ16_m7	**26 / 8 s**	19 / 933 s	5 / 2672 s	12 / 2206 s	7 / 2577 s	11 / 2262 s
12	R0_GearDoor	**26 / 0.36 s**	26 / 3 s	26 / 1 s	26 / 0.82 s	26 / 0.84 s	26 / 0.85 s
13	R1_Valve	**26 / 0.87 s**	26 / 8 s	26 / 2 s	26 / 3 s	26 / 3 s	26 / 3 s
14	R2_Outputs	**26 / 2 s**	26 / 15 s	26 / 6 s	26 / 5 s	26 / 5 s	26 / 6 s
15	R3_Sensors	12 / 1727 s	26 / 32 s	26 / 41 s	**26 / 27 s**	26 / 68s	26 / 59s
16	R4_Handle	**5 / 2560 s**	2 / 565 s	4 / 2732 s	4 / 2732 s	4 / 2878 s	4 / 3342 s
17	R5_Switch	9 / 2142 s	**26 / 274 s**	9 / 2207 s	9 / 2285 s	6 / 1364 s	7 / 2483 s
18	R6_Lights	7 / 2344 s	**26 / 382 s**	6 / 2523 s	5 / 2563 s	7 / 1463 s	5 / 2606 s
19	Lightbot	3 / 2762 s	**11 / 1883 s**	6 / 2472 s	5 / 2658 s	6 / 91 s	4 / 2686 s
20	M0_AAI	2 / 2904 s	**26 / 63 s**	11 / 2129 s	4 / 2771 s	6 / 791 s	5 / 2703 s
21	M0_AAT	3 / 2761 s	**26 / 70 s**	8 / 2510 s	3 / 2796 s	5 / 487 s	3 / 2799 s
22	M0_AOO	3 / 2761 s	**26 / 16 s**	4 / 2752 s	6 / 2572 s	4 / 2390 s	6 / 2569 s
23	M0_VOO	3 / 2761 s	**26 / 24 s**	13 / 2222 s	5 / 2680 s	4 / 29 s	4 / 2689 s
24	M0_VVI	3 / 2761 s	**26 / 61 s**	9 / 2437 s	4 / 2743 s	5 / 77 s	5 / 2722 s
25	M0_VVT	3 / 2779 s	**26 / 63 s**	10 / 2033 s	5 / 2718 s	8 / 1736 s	4 / 2741 s
26	M1_AOOR	3 / 2762 s	**26 / 42 s**	5 / 2703 s	4 / 2786 s	3 / 143 s	3 / 2795 s
27	M1_VOOR	3 / 2761 s	**26 / 37 s**	8 / 2288 s	6 / 2533 s	6 / 288 s	6 / 2471 s
28	M2_AAI	3 / 2761 s	**26 / 60 s**	10 / 2451 s	4 / 2687 s	7 / 1510 s	4 / 2688 s
Total		325 / 48 737 s	**564 / 7913 s**	370 / 44 709 s	332 / 47 969 s	331 / 21 047 s	327 / 49 020 s
				Solved constraints / Runtime s			

6.7.2. Direct Implementation of SMT Solving in ProB

In the following we present an empirical evaluation of the direct implementation of SMT solving in ProB. Here, we compare the runtimes of the plain ProB constraint solver, its integration of Z3 [49] (ProB-Z3) running two solvers in parallel without the decomposition of constraints into independent components, the presented SMT solver using ProB's constraint solver as its only theory solver with and without static syntax analysis as described in Section 6.5.2 (SMT, Raw-SMT), in addition to the static syntax analysis using static symmetry breaking (Sym-SMT) as described in Section 6.5.2, and using no static syntax analysis but static symmetry breaking (Sym-Raw-SMT). We use a linear domain enumeration order for ProB's constraint solver in each solver configuration to ensure that the propagation is deterministic. The benchmarks were run on the same settings as used in Section 6.7.1.

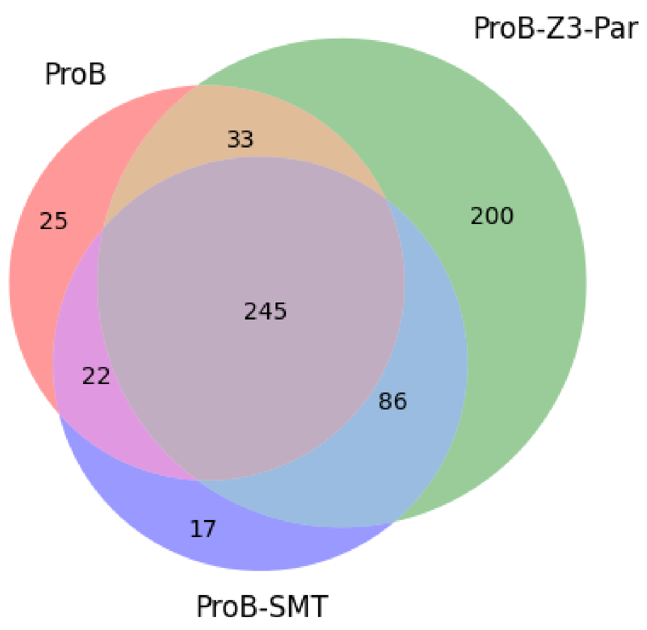


Figure 6.10.: A Venn diagram to visualize and compare the amount of BMC constraints that can be solved by ProB's constraint solver, the parallel integration of Z3 in ProB, and the best performing configuration of ProB's SMT solver, *i.e.*, using static syntax analysis but no symmetry breaking.

Bounded Model Checking

We use the same set of benchmarks as used in Section 6.7.1 for bounded model checking. The evaluated benchmarks can be seen in Table 6.2.

When comparing the results of the SMT and Raw-SMT solver configurations, it can be seen that the static syntax analysis improves the performance of constraint solving for several benchmarks, in particular for the benchmarks numbered 1, 7, 10, 20, 21, 23, 25, and 28. However, for some benchmarks the additional constraints seem to lead the SMT solver in a wrong direction, *e.g.*, for the benchmarks numbered 10 and 11. This can lead to a constellation of theory constraints for which ProB's constraint solver exceeds the predefined solver timeout or the SAT solver spends a lot of time backtracking between variable decisions. We suppose that the reason is that our decision heuristic is initialized with the occurrences of variables among all clauses as described in Section 6.5.3. This initialization changes when adding additional clauses which can lead the SAT solver in a different direction than is the case for the original formula. Table 6.3 shows more detailed statistics of the different SMT solver configurations. Here, it can be seen that the additional static syntax analysis (SMT, Sym-SMT) reduces the amount of theory propagations by several orders of magnitude compared to the other solver configurations while the amount of Boolean decisions increases slightly. This shows that the deduced constraints successfully pass knowledge from the theory to the SAT solver. Both SMT solver configurations that do not use the static syntax analysis seem to often exceed

Table 6.3.: Detailed statistics of the different configurations of ProB's SMT solver considering all BMC constraints presented in Table 6.2. Each cell presents two values which are the maximum (top) and mean (bottom) values. The presented mean values have been rounded.

Solver	Conflicts	Theory Propagations	Restarts	Boolean Decisions
SMT	5352	1 360 087	33	343 576
	176	4963	0	24 949
Raw-SMT	3222	3 727 792	21	273 901
	305	8151	1	29 248
Sym-Raw-SMT	4312	4 533 060	21	267 666
	314	12 736	1	28 049
Sym-SMT	3943	1 360 087	18	283 952
	69	4558	0	8802

the predefined timeout in the theory solver, which is probably correlated with the high amount of theory propagations.

Adding symmetry breaking constraints does not improve the performance of constraint solving for the selected benchmarks significantly. Only for the benchmarks numbered 2, 11, and 18, one more constraint can be solved. For the second benchmark, the time needed for constraint solving can be reduced as can be seen when comparing the SMT and Sym-SMT solver configurations in Table 6.2. We initially expected greater performance improvements of static symmetry breaking, but it should be noted that we do not know how many constraints contain symmetries. Furthermore, a symmetry breaking constraint does not necessarily shift a contradiction to the Boolean level of SMT solving but possibly just supports the theory solver as explained in Section 6.5.2. It can thus be the case that the theory solver or the SAT solver still exceeds a predefined solver timeout if the responsible constraints are not affected by symmetry breaking. Unfortunately, in some cases the performance of constraint solving is worse when adding symmetry breaking predicates, *e.g.*, for the benchmarks numbered 7 and 23. Again, we suppose that the additional constraints lead the SMT solver in a wrong direction which results in exceeding the predefined solver timeout due to the enumeration of domains in the theory solver or backtracking between variable decisions in the SAT solver.

We cannot evaluate the usefulness of restarts since the SMT solver configurations do not apply many restarts for the selected benchmarks. Yet, an SMT solver only restarts a search if it recognizes that not much new knowledge can be gained from the current search path as explained in Section 6.5.3. It can thus be a good sign that only a few restarts were performed in our empirical evaluation. In Table 6.3, it can be seen that the mean amount of restarts over all BMC constraints is 0 or 1. The maximum amount of restarts when solving a constraint is 33.

The SMT solver configurations reduce the amount of learned clauses only a few times for the selected benchmarks. In most cases, the total amount of conflicts is less than the threshold defined by the implemented policy as described in Section 6.5.3 (see Table 6.3). In particular, we remove half of the learned clauses which have an LBD score higher than 5 after $4000 + 300 * x$ conflicts, where x is the amount of reductions performed so far.

The theory solver, *i.e.*, ProB's constraint solver, is able to deduce many constraints which are propagated to the SAT solver as can be seen in Table 6.3. We would need to explain such theory propagations if they are necessary for a conflict analysis since ProB's constraint solver does not provide an explanation by default as described in Section 6.5.4. This would add the additional overhead of computing an unsatisfiable core. Yet, the SMT solver configurations do not require a single explanation of a theory propagation for the selected benchmarks.

The results of ProB's SMT solver for the benchmarks numbered 20 to 28 are not much better than the ones of using only ProB's constraint solver. The integration of Z3 is still the dominant solver. We suppose that Z3 is able to solve these constraints better than the other solvers not because of conflict-driven clause learning but due to its strong theory solvers for linear integer arithmetic.

Overall, ProB's SMT solver is able to solve many constraints better than ProB's constraint solver, *e.g.*, the benchmarks numbered 1, 2, 7, 15, 20, 23, and 28. In Figure 6.10, it can be seen that the constraint solver configuration ProB-SMT is able to solve 17 constraints that cannot be solved by Z3 or ProB's constraint solver, 22 constraints that cannot be solved by Z3, and 86 constraints that can be solved by Z3 but not by ProB's constraint solver.

In Figure 6.9, it can be seen that ProB's constraint solver is able to solve more constraints than ProB's SMT solver for a timeout smaller than 40 s. However, for larger timeouts, the SMT solver configuration using static syntax analysis but no symmetry breaking has a better performance than the other constraint solvers. It could thus be beneficial to combine the constraint solvers by first calling ProB's constraint solver with a timeout of around 20 s and resorting to ProB's SMT solver if the timeout is exceeded. The results show that CDCL can be beneficial to find contradictions in such large constraints that contain many Boolean decisions as selected from bounded model checking compared to plain saturation-based solving as performed by ProB's constraint solver. We thus deem this direct implementation of SMT solving in ProB to be of value for constraint solving in B and Event-B, and to further increase the power of ProB's portfolio of constraint solving backends.

Inductive Invariant Proofs

In order to provide a more diverse performance evaluation, we decided to additionally solve constraints from constraint-based proofs for the inductivity of invariants. The goal is to prove that a classical B machine operation or event in Event-B is not able to reach a state that violates the invariant. For this, a constraint is set up for each machine operation or event which is solved independently. In contrast to bounded model checking, the constraint-based proof for the inductivity of an invariant does not include the machine's initialization but allows any instantiation. These constraints thus often contain larger or unbounded domains. Further, the constraints are considerably smaller

Table 6.4.: A subset of the classical B and Event-B models from Table 6.1 and Table 6.2, but checking the inductivity of the invariant I for each operation or event op by solving the constraint $I \wedge \text{BA}_{\text{op}}(v, v') \wedge \neg I'$. The amount of machine operations (events) is equal to the amount of constraints to be solved.

				ProB-Z3	ProB			
No.	Name	Ops.	ProB	(parallel)	SMT	Raw-SMT	Sym-SMT	Sym-Raw-SMT
1	Prisoners-4	3	1 / 120 s	**1 / 0.55 s**	1 / 120 s	1 / 120 s	1 / 120 s	1 / 120 s
2	Bakery	10	0 / 1081 s	0 / 20 s	1 / 961 s	**1 / 951 s**	1 / 962 s	1 / 964 s
3	Paxos-3	5	0 / 482 s	0 / 4 s	**2 / 123 s**	**2 / 123 s**	2 / 124 s	2 / 124 s
4	TravelAgency	10	**6 / 625 s**	0 / 5 s	6 / 791 s	6 / 786 s	6 / 800 s	6 / 783 s
5	LargeBranching	2	**2 / 0.01 s**	2 / 0.26 s	**2 / 0.01 s**	**2 / 0.01 s**	**2 / 0.01 s**	2 / 0.02 s
6	SearchEvents	4	3 / 120 s	**4 / 1 s**	3 / 122 s	4 / 3 s	3 / 122 s	4 / 2 s
7	ABZ16_m4	19	19 / 0.16 s	19 / 0.09 s	**19 / 0.05 s**	19 / 0.07 s	**19 / 0.05 s**	19 / 0.12 s
8	ABZ16_m5	22	**22 / 0.08 s**	22 / 0.1 s	22 / 0.12 s	22 / 0.15 s	22 / 0.14 s	22 / 0.13 s
9	ABZ16_m6	24	**24 / 0.17 s**	24 / 0.46 s	24 / 0.45 s	24 / 0.37 s	24 / 1 s	24 / 1 s
10	ABZ16_m7	26	**26 / 0.12 s**	26 / 0.37 s	26 / 0.52 s	26 / 0.37 s	26 / 1 s	26 / 1 s
11	R0_GearDoor	8	**8 / 0.01 s**	**8 / 0.01 s**	**8 / 0.01 s**	**8 / 0.01 s**	**8 / 0.01 s**	**8 / 0.01 s**
12	R1_Valve	16	16 / 0.02 s	**16 / 0.01 s**	**16 / 0.01 s**	16 / 0.02 s	16 / 0.02 s	16 / 0.02 s
13	R2_Outputs	24	24 / 0.04 s	**24 / 0.03 s**	**24 / 0.03 s**	24 / 0.04 s	24 / 0.04 s	24 / 0.04 s
14	R3_Sensors	24	24 / 0.08 s	**24 / 0.07 s**	**24 / 0.07 s**	**24 / 0.07 s**	24 / 0.11 s	24 / 0.13 s
15	R4_Handle	32	**32 / 0.36 s**	24 / 4 s	32 / 6 s	32 / 19 s	32 / 2 s	32 / 2 s
16	R5_Switch	32	**32 / 0.15 s**	32 / 1 s	32 / 0.35 s	32 / 0.34 s	32 / 0.38 s	32 / 0.36 s
17	R6_Lights	39	**39 / 0.24 s**	39 / 1 s	39 / 0.64 s	39 / 0.64 s	39 / 0.75 s	39 / 0.72 s
18	Lightbot	7	6 / 120 s	7 / 1 s	**7 / 0.56 s**	7 / 0.57 s	7 / 0.65 s	7 / 1 s
19	M0_AAI	6	**6 / 0.03 s**	6 / 1 s	4 / 240 s	5 / 121 s	4 / 240 s	5 / 120 s
20	M0_AAT	6	**6 / 0.03 s**	6 / 1 s	4 / 240 s	4 / 240 s	4 / 240 s	4 / 241 s
21	M0_AOO	4	**4 / 0.02 s**	4 / 0.59 s	3 / 120 s	4 / 0.63 s	3 / 120 s	4 / 0.7 s
22	M0_VOO	4	**4 / 0.02 s**	4 / 0.66 s	3 / 120 s	4 / 1 s	3 / 121 s	4 / 1 s
23	M0_VVI	6	**6 / 0.1 s**	6 / 1 s	4 / 240 s	5 / 120 s	4 / 241 s	5 / 121 s
24	M0_VVT	6	**6 / 0.03 s**	6 / 1 s	4 / 242 s	5 / 120 s	4 / 241 s	5 / 120 s
25	M1_AOOR	6	**6 / 0.04 s**	6 / 0.99 s	5 / 124 s	6 / 2 s	5 / 124 s	6 / 2 s
26	M1_VOOR	6	**6 / 0.03 s**	6 / 0.98 s	5 / 125 s	6 / 2 s	5 / 124 s	6 / 2 s
27	M2_AAI	7	6 / 120 s	**7 / 1 s**	5 / 240 s	4 / 360 s	5 / 240 s	4 / 361 s
	Total		**332 / 2670 s**	321 / 47 s	323 / 3817 s	330 / 2971 s	323 / 3825 s	330 / 2968 s
				Solved constraints / Runtime s				

than the ones of bounded model checking since they only consider a single machine operation or event at once. We use a subset of the benchmarks used in Section 6.7.1 and Section 6.7.2 with the same solver settings and compare the runtimes of ProB's constraint solver, the parallel integration of Z3 (ProB-Z3), as well as all four settings of ProB's SMT solver (SMT, Raw-SMT, Sym-SMT, Sym-Raw-SMT). We dropped the benchmark SimpleTwoPhase from the evaluation since the B machine only provides a single machine operation for which the constraint to prove the inductivity of the machine invariant is a static contradiction. The benchmarks with the corresponding amount of machine operations or events, *i.e.*, the amount of constraints to be solved, can be seen in Table 6.4. The constraints of all benchmarks have an average amount of 23 unique conjuncts or disjuncts and a median amount of 56. The largest constraint contains 72 unique conjuncts or disjuncts.

First and foremost, it can be seen that the different configurations of ProB's SMT

Table 6.5.: Detailed statistics of the different configurations of ProB's SMT solver considering all constraints for inductive invariant proofs presented in Table 6.4. Each cell presents two values which are the maximum (top) and mean (bottom) values. The presented mean values have been rounded.

Solver	Conflicts	Theory Propagations	Restarts	Boolean Decisions
SMT	17	2 345 870	0	1601
	1	8972	0	37
Raw-SMT	20	2 728 235	0	1598
	1	9821	0	37
Sym-Raw-SMT	14	2 102 837	0	373
	0	8365	0	25
Sym-SMT	11	1 965 481	0	348
	0	9722	0	27

solver do not show crucial differences. The additional static syntax analysis and symmetry breaking do not improve but rather drop performance, *e.g.*, for the benchmarks numbered 19, 21, 23, 24, and 26. Only for the 27th benchmark one more constraint can be solved when using the static syntax analysis. Further, the runtime for the 15th benchmark reduces when using symmetry breaking. For the benchmarks numbered 2, 3, and 18, conflict-driven clause learning improves the coverage compared to ProB's constraint solver.

The integration of Z3 in ProB shows benefits for the benchmarks numbered 1, 6, and 27, where it is able to solve the maximum amount of constraints. For the 15th benchmark, Z3 is not able to solve 8 constraints which can be solved by the other constraint solvers. These constraints contain several nested functions and set cardinalities which result in quantified formulas in SMT-LIB.

Interestingly, ProB's constraint solver is the dominant solver for the selected benchmarks regarding performance since it is able to solve the constraints of the benchmarks numbered 15 to 17 and 19 to 26 the fastest. Especially for the benchmarks numbered 19 to 26, it can be seen that ProB's SMT solver lacks performance while ProB's constraint solver alone is able to solve the constraints in a short amount of time. Here, the SAT solver again guides the theory solver in an inconvenient direction which results in exceeding the solver timeout as was the case for several benchmarks presented in Section 6.7.2. Table 6.5 shows detailed statistics of the different SMT solver configurations for the benchmarks presented in Table 6.4. It can be seen that the factor between the amount of Boolean decisions and theory propagations is several orders of magnitude higher than was the case for the benchmarks from bounded model checking. We thus suppose that the advantage of ProB's constraint solver can be attributed to the constraints' smaller amount of Boolean decisions compared to the ones of bounded model checking, where conflict-driven clause learning is not necessarily better than setting up all theory constraints at once as is done by ProB's constraint solver.

Deadlock Freedom Proofs

In order to further enrich the diversity of the selected benchmarks for our empirical evaluation, we decided to additionally use benchmarks from constraint-based proofs for deadlock freedom. For this, a single constraint is solved for a classical B or an Event-B machine to search for a state which has no successor state, *i.e.*, a deadlock state. We use the same models and settings as in Section 6.7.2, but this time we dropped the benchmark R6_lights from the evaluation since the constraint to prove deadlock freedom is a static contradiction. The evaluated benchmarks can be seen in Table 6.6. A dash indicates that a constraint cannot be solved by a specific constraint solver within the predefined timeout of 2 min. The constraints have a similar size than the ones used for the proofs of the inductivity of invariants but are considerably smaller than the ones of bounded model checking. In particular, the constraints of all benchmarks have an average amount of 25 unique conjuncts or disjuncts and a median amount of 22. The largest constraint contains 48 unique conjuncts or disjuncts.

First and foremost, it can be seen that ProB's constraint solver is the dominant solver for the presented benchmarks. It is able to solve all constraints except for the first and second one. Yet, the other constraint solvers are not able to solve these constraints within the predefined timeout too.

The four configurations of ProB's SMT solver do not show significant differences in general. Their results are mostly comparable to the results of ProB's constraint solver. The constraints do not lead to many Boolean decisions as can be seen in Table 6.7. While conflict-driven clause learning is not necessarily beneficial in such cases, it does not seem to add too much overhead compared to registering all variables at once as is done by ProB's constraint solver.

The integration of Z3 in ProB is not able to solve 10 constraints in total. Here, Z3 does not exceed the predefined solver timeout but answers unknown in a short amount of time. This shows that Z3's inability to solve a constraint is not necessarily caused by the size of a constraint but rather by the use of specific operators. We are not sure which operators exactly reduce Z3's performance, but we suppose that the main reason are quantifiers introduced for the translations of set cardinality constraints and functions.

All in all, the benchmarks selected from proofs of deadlock freedom show that ProB's constraint solver is superior when it comes to solving constraints with a small amount of Boolean decisions. In such cases, SMT solving usually does not improve performance.

Additional Theory Solver

The various machines (no. 20–28) of the pacemaker model in Table 6.2 highlighted some of the drawbacks of ProB's default and our new SMT solver. Indeed, the pacemaker model contains timing constraints, some over unbounded domains, and also has events with an infinite number of parameter values. ProB's constraint solver was not able to narrow down these domains to a finite interval. We thus decided to combine our new SMT solver with a new additional theory solver for integer difference logic as described in Section 6.6 and evaluate it on the pacemaker constraints from Table 6.2. We use

Table 6.6.: A subset of the classical B and Event-B benchmarks used in Table 6.1 and Table 6.2 but solving constraints to prove deadlock freedom. One constraint is solved for each benchmark. A dash indicates that a constraint cannot be solved by a specific constraint solver within the predefined timeout of 2 min.

			ProB-Z3	ProB			
No.	Name	ProB	(parallel)	SMT	Raw-SMT	Sym-SMT	Sym-Raw-SMT
1	Prisoners-4	-	-	-	-	-	-
2	Bakery	-	-	-	-	-	-
3	Paxos-3	**0.01 s**	0.32 s	0.07 s	0.06 s	0.27 s	0.37 s
4	SimpleTwoPhase	**0.01 s**	-	**0.01 s**	**0.01 s**	0.03 s	0.06 s
5	TravelAgency	0.03 s	-	0.06 s	0.06 s	0.14 s	0.14 s
6	SimpleTwoPhase	**0.01 s**	12 s	**0.01 s**	**0.01 s**	0.02 s	**0.01 s**
7	SearchEvents	**0.01 s**	0.61 s	**0.01 s**	**0.01 s**	0.02 s	0.12 s
8	ABZ16_m4	**0.01 s**	0.18 s	0.03 s	**0.01 s**	0.04 s	0.03 s
9	ABZ16_m5	**0.01 s**	0.18 s	0.03 s	0.02 s	0.04 s	0.03 s
10	ABZ16_m6	**0.01 s**	-	0.03 s	0.03 s	0.06 s	0.08 s
11	ABZ16_m7	**0.01 s**	-	0.03 s	0.02 s	0.05 s	0.04 s
12	R0_GearDoor	**0.01 s**	-	**0.01 s**	**0.01 s**	**0.01 s**	**0.01 s**
13	R1_Valve	**0.01 s**	-	**0.01 s**	**0.01 s**	0.02 s	0.02 s
14	R2_Outputs	**0.01 s**	0.18 s	**0.01 s**	**0.01 s**	0.03 s	0.02 s
15	R3_Sensors	**0.01 s**	-	0.02 s	**0.01 s**	0.04 s	0.03 s
16	R4_Handle	**0.01 s**	1 s	1 s	2 s	1 s	3 s
17	R5_Switch	**0.01 s**	0.29 s	0.07 s	0.08 s	0.12 s	0.14 s
18	Lightbot	**0.01 s**	-	0.05 s	0.06 s	0.07 s	0.09 s
19	M0_AAI	**0.01 s**	0.28 s	0.1 s	0.07 s	0.11 s	0.08 s
20	M0_AAT	**0.01 s**	0.28 s	0.03 s	0.03 s	0.04 s	0.04 s
21	M0_AOO	**0.01 s**	0.25 s	**0.01 s**	**0.01 s**	0.03 s	0.02 s
22	M0_VOO	**0.01 s**	0.18 s	**0.01 s**	**0.01 s**	0.02 s	0.02 s
23	M0_VVI	**0.01 s**	0.25 s	0.1 s	0.07 s	0.11 s	0.07 s
24	M0_VVT	**0.01 s**	0.26 s	0.1 s	0.07 s	0.12 s	0.07 s
25	M1_AOOR	**0.01 s**	0.23 s	0.02 s	0.03 s	0.04 s	0.03 s
26	M1_VOOR	**0.01 s**	0.28 s	0.02 s	0.02 s	0.03 s	0.03 s
27	M2_AAI	**0.01 s**	0.32 s	0.03 s	0.04 s	0.07 s	0.13 s
	Total	**25 / 0.27 s**	17 / 17.09 s	25 / 1.87 s	25 / 2.76 s	25 / 2.54 s	25 / 4.68 s
				Solved constraints / Runtime s			

Table 6.7.: Detailed statistics of the different configurations of ProB's SMT solver considering all constraints for deadlock freedom proofs presented in Table 6.6. Each cell presents two values which are the maximum (top) and mean (bottom) values. The presented mean values have been rounded.

Solver	Conflicts	Theory Propagations	Restarts	Boolean Decisions
SMT	5	11	0	316
	0	2	0	41
Raw-SMT	8	28	0	406
	1	6	0	50
Sym-Raw-SMT	8	28	0	406
	1	6	0	51
Sym-SMT	5	11	0	316
	0	2	0	42

the same system settings as for the other empirical evaluations and use the additional theory solver for integer difference logic for each configuration of ProB's SMT solver (columns 6 to 9 in Table 6.8). It should be noted that the other benchmarks presented in Table 6.2 do not contain any (or only very few) integer difference logic constraints. The use of the additional theory solver would thus not make any difference.

The evaluated benchmarks are presented in Table 6.8. It can be seen that the additional theory solver for integer difference logic (SMT-IDL) allows solving 173 more constraints than ProB's constraint solver and 121 more constraints than ProB's SMT solver with the default theory solver backend. The SMT solver configurations perform many Boolean decisions as can be seen in Table 6.9. The additional static syntax analysis improves performance and enables more theory propagations in ProB's constraint solver, which seem to be beneficial for constraint solving regarding the selected set of benchmarks. Yet, using the additional theory solver prevents solving 7 constraints in total as can be seen in Figure 6.11. We assume that the unsatisfiable cores provided by the integer difference logic solver lead the SMT solver in a different and in this case unfavorable direction. Overall, the integration of Z3 is still the superior constraint solver for the selected benchmarks.

While this brief empirical evaluation serves to demonstrate the usefulness of the additional theory solver for ProB's SMT solver, a more quantitative study is needed to gain more insight on the general strengths of the different constraint solving backends.

6.8. Related Work

In the following, we describe different related work in the area of SMT solving for B and first-order logic in general.

Déharbe et al. [208–210] presented an integration of an SMT solver for B and Event-

Table 6.8.: A set of benchmarks from BMC of an Event-B model of a pacemaker by Méry and Singh [2] comparing the different configurations of ProB's SMT solver using the new theory solver for IDL with ProB's constraint solver and the new parallel integration of Z3. All constraints contain at least one IDL constraint.

			ProB-Z3	ProB	ProB			
No.	Name	ProB	(parallel)	SMT	SMT-IDL	Raw-SMT-IDL	Sym-SMT-IDL	Sym-Raw-SMT-IDL
1	M0_AAI	2 / 2904 s	**26 / 63 s**	11 / 2129 s	19 / 1384 s	17 / 1382 s	15 / 636 s	16 / 1504 s
2	M0_AAT	3 / 2761 s	**26 / 70 s**	8 / 2510 s	24 / 1122 s	15 / 1781 s	22 / 696 s	16 / 1781 s
3	M0_AOO	3 / 2761 s	**26 / 16 s**	4 / 2752 s	25 / 2281 s	20 / 8027 s	21 / 1507 s	20 / 7938 s
4	M0_VOO	3 / 2761 s	**26 / 24 s**	13 / 2222 s	26 / 203 s	19 / 1253 s	23 / 223 s	19 / 1183 s
5	M0_VVI	3 / 2761 s	**26 / 61 s**	9 / 2437 s	20 / 1347 s	17 / 1447 s	16 / 684 s	17 / 1444 s
6	M0_VVT	3 / 2779 s	**26 / 63 s**	10 / 2033 s	19 / 1406 s	13 / 1964 s	16 / 664 s	13 / 1999 s
7	M1_AOOR	3 / 2762 s	**26 / 42 s**	5 / 2703 s	24 / 934 s	14 / 1939 s	16 / 558 s	13 / 1922 s
8	M1_VOOR	3 / 2761 s	**26 / 37 s**	8 / 2288 s	21 / 1307 s	15 / 1727 s	17 / 880 s	15 / 1686 s
9	M2_AAI	3 / 2761 s	**26 / 60 s**	10 / 2451 s	21 / 1200 s	9 / 2178 s	16 / 859 s	9 / 2184 s
	Total	26 / 25 011 s	**234 / 436 s**	78 / 21 525 s	199 / 11 184 s	139 / 21 698 s	162 / 6707 s	138 / 21 641 s

Solved constraints / Runtime s

B by translating to SMT-LIB. The goal was to support automated theorem provers by disproving single proof-obligations. The authors presented two translations which support a subset of the B language. Sets are translated as uninterpreted characteristic functions. One translation specifically interfaces an SMT solver and uses its lambda expressions, but only basic sets are supported in this case. Our implementation uses Z3's array theory [183] to translate sets which supports defining nested sets. In the other translation, set operations are axiomatized to support nested sets. The axiomatic translation presented by Krings and Leuschel [48] and described in Section 6.3.1 is similar to this translation, but uses Z3's array theory [183] instead of uninterpreted functions. An empirical evaluation by Déharbe et al. has shown that the amount of proof obligations which can be proven automatically has improved [210]. Krings and Leuschel have shown that their derived high-level translation improves the one by Déharbe et al. regarding constraint solving [48].

The mathematical foundations of TLA$^+$ and B have quite a few similarities, and translations between both formalisms exist [132, 133]. TLC [211] is an explicit state model checker for TLA$^+$ that relies on simple domain enumeration. Konnov et al. [202] presented a translation from TLA$^+$ to SMT-LIB to improve symbolic model checking by interfacing to SMT solvers. The translation only supports finite sets, which avoids many downsides of our translation from B to SMT-LIB. For instance, the authors suggest translating a set membership as a disjunction of equalities, which is feasible for finite sets only. Furthermore, quantifiers are unfolded, *e.g.*, an existential quantification is replaced by a disjunction. In the future, we plan to conduct an empirical comparison with APALACHE's SMT solver integration [202], which will require a fair translation of TLA$^+$ constraints to B and backwards, and isolating the constraint solving performed by APALACHE from the symbolic verification algorithms.

Davidson et al. [212] presented a portfolio of constraint solving backends for the high-

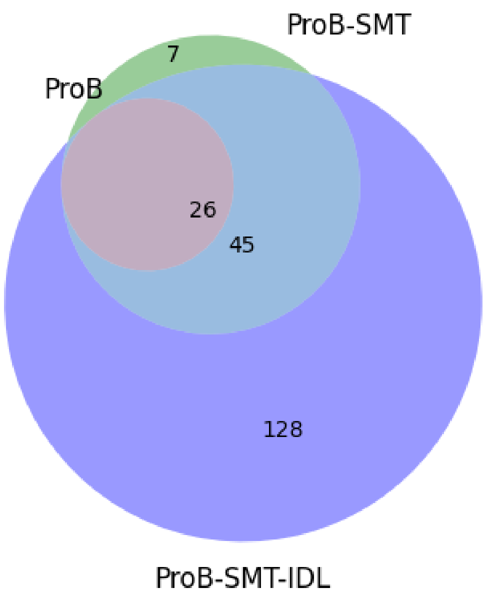


Figure 6.11.: A Venn diagram to visualize and compare the amount of BMC constraints that can be solved by ProB's constraint solver, ProB's SMT solver with static syntax analysis, and the same SMT solver configuration but additionally using the theory solver for integer difference logic (SMT-IDL).

level language Essence Prime. One backend interfaces different SMT solvers including Z3 by translating to SMT-LIB, and supports four different SMT-LIB logics for quantifier-free formulas. The translation uses bit-blasting and only supports finite domains. Besides that the authors' tool enables to interface different SAT solvers by translating to propositional logic or other constraint solvers using their specific input language. The authors have shown that their SMT solver integration outperforms the baseline approaches for the selected benchmarks. In addition, they also emphasize the need for a portfolio of constraint-solving backends to reliably solve various problems.

El Ghazi and Taghdiri [213] presented a translation from Alloy to SMT-LIB. Abbazzi et al. [214] presented an integration of SMT solvers in the Alloy analyzer, as well as an evaluation of different translations from Alloy to SMT-LIB. The Alloy analyzer usually translates Alloy to Kodkod [45] which applies SAT solving. Yet, this eager approach to SMT solving can result in large propositional formulas depending on the size of domains. This possibly leads to bad performance. For instance, sets can be translated as bit vectors where one bit is reserved for each domain element. The authors have shown performance improvements of model finding for Alloy by translating to SMT-LIB. Furthermore, the translation enables reasoning over infinite sets.

Weber [215] presented an SMT solver integration for the HOL4 theorem prover which supports the first-order subset of the language. The translation to SMT-LIB employs an axiomatized style for operators that are not supported by SMT-LIB such as the minimum of a set of integers.

Bride et al. [216] conducted an empirical evaluation and comparison of SMT solving

Table 6.9.: Detailed statistics of the different configurations of ProB's SMT solver using the additional theory solver for integer difference logic considering all BMC constraints presented in Table 6.8. Each cell presents two values which are the maximum (top) and mean (bottom) values. The presented mean values have been rounded.

Solver	Conflicts	Theory Propagations	Restarts	Boolean Decisions
SMT-IDL	3915	71 479	25	302 609
	837	973	4	85 307
Raw-	6551	35 545	68	481 976
SMT-IDL	1744	1085	10	141 389
Sym-Raw-	6520	35 545	65	501 815
SMT-IDL	1729	927	10	140 248
Sym-	3391	59 062	19	290 325
SMT-IDL	509	529	3	51 411

and constraint logic programming for workflow nets. In particular, they interface Z3 and SICStus Prolog as is the case for our implementations. Their results show benefits of SMT solving for unsatisfiable formulas, and benefits of constraint logic programming for satisfiable ones, which fits also with our experience.

6.9. Future Work

In the future, we plan to provide alternative translations for B sequences using lambda functions. Furthermore, the translation of B sequences to SMT-LIB can be improved by translating sequences as finite arrays in SMT-LIB. Yet, this is only possible if sequences interact among other sequences guaranteeing the well-definedness of resulting sequences. This is not necessarily the case since B sequences are relations and might interact with other relations which are not sequences as described in Section 6.3.1. We thus need a static analysis to detect if a translation of sequences as finite arrays in SMT-LIB is applicable for a constraint.

As discussed in Section 6.7.1, the support for infinite sets entails several suboptimal translations, *e.g.*, for set cardinality constraints. If only finite sets are used in a formula, we are able to translate sets to a more concise representation, *e.g.*, using a bit vector encoding. Of course, we then have to provide translations for all set operators for this new encoding of finite sets which requires some implementation effort. Yet, this might not be worth it since ProB already provides an interface to Kodkod [45, 46] which has shown to provide good performance on finite set operations [149] and uses a bit vector encoding of sets.

Furthermore, we plan to compile other configurations of the Z3 constraint solver to run in parallel, *e.g.*, using different solver tactics.

Another future work is to use other SMT solvers to solve SMT-LIB models. Currently, the new translation presented in this article uses Z3 specific lambda functions. Once the SMT-LIB standard officially supports lambda functions we should be able to interface to other SMT solvers as well for the new translation. It is worth mentioning that the implementation of an automated translation which interfaces a solver specific programming API is a tedious and error-prone task. Mann et al. [217] presented a solver-agnostic programming API for SMT solving which should be considered for future implementations.

Regarding our direct implementation of SMT solving in ProB we plan to implement more sophisticated CNF rewriting rules to decrease the number of clauses and their size, *e.g.*, our implementation lacks a heuristic to reduce the introduction of artificial variables [64, 218] as proposed by Tseitin [60]. Besides that the SAT solver's branching heuristic could use knowledge from theory constraints to improve the ordering (see benchmarks 19 to 27 in Table 6.4). This requires a more detailed analysis of selected constraints to investigate whether we can deduce any rules to improve the branching heuristic. In addition, we want to consider the model-constructing satisfiability calculus (mcSAT) [219] framework in the future to investigate whether we can further improve the overall performance and apply the presented ideas to set theory. Last but not least, there may be some low-hanging fruit in our implementation to improve performance.

6.10. Conclusion

In conclusion, we have presented a formal description and implementation of a new translation from B to SMT-LIB as well as a parallel integration of the Z3 constraint solver in ProB. Empirical results have shown that the new translation and workflow improves performance and coverage compared to the prior integration in ProB [48] by utilizing Z3's lambda functions. The integration of Z3 is also able to decide a lot of constraints where ProB's constraint solver times out (see Section 6.7.1). In most cases, such constraints contain bounded or unbounded integer domains and function applications. Besides improving the integration of Z3 in ProB we were able to identify two bugs in Z3 involving lambdas using ProB's regression tests.

Unfortunately, the integration of Z3 is not effective for constraints involving set cardinality or many quantifiers. We thus also developed a direct implementation of SMT solving in ProB using its constraint solver as a theory solver (see Section 6.7.2). This new approach was able to solve some constraints that neither ProB nor Z3 were able to solve. The static syntax analysis in Section 6.5.2 derives implied constraints and was useful for identifying contradictions. Yet, the additional constraints can also be counterproductive and lead to timeouts. Using a branching heuristic in the underlying SAT solver that considers the style of the actual theory constraints could improve this issue in the future. Adding static symmetry breaking predicates improved the performance for some benchmarks, but not as much as initially expected. The empirical evaluation has shown that the benefit of CDCL compared to plain saturation-based solving, as performed by ProB's constraint solver, is most notable for large constraints with many disjunctions or implications. These occur for example in bounded model checking (see

Section 6.7.2) with a monolithic transition predicate (consisting of a disjunction of the effect of a model's individual operations).

The use of an additional theory solver for integer difference logic in ProB's SMT solver has shown to be beneficial for models involving timing constraints (see Section 6.7.2). This theory solver also provides unsatisfiable cores without requiring further computations.

Generally, the integration of Z3 shows a better performance for most bounded model checking constraints than ProB's SMT solver. We mainly attribute these differences to the strong theory solvers of Z3, especially for linear integer arithmetic (see Section 6.7.1). The decomposition of constraints into independent components prior to the translation to SMT-LIB did not improve performance for Z3 (see Section 6.7.1). Possibly, Z3 is able to infer these components directly or indirectly during the solving process.

Last but not least, ProB's constraint solver sometimes performs better than the integration of Z3 or the new SMT solver, especially for checking inductivity of invariants and deadlock freedom (see Section 6.7.2). These constraints are smaller than the ones of bounded model checking, as there is no repeated inclusion of the transition predicate.

Finally, our empirical evaluation has shown that no constraint solver is the best for all types of constraints. Hence, it is beneficial to have a diverse portfolio of constraint solving backends for the B language. We could either call all available solvers in parallel or iteratively call different constraint solvers. Our empirical evaluation has shown that it could be useful to first call ProB's constraint solver with a small timeout and successively resort to the integration of Z3 as well as ProB's SMT solver if necessary. Further, we could extend the machine learning backend in ProB that is able to predict the best solver for a specific constraint as suggested by Dunkelau et. al [149] (see also Healy et al. [220] for Why3) to combine the strengths of all presented backends into a single constraint solving routine.

Acknowledgments We would like to thank the anonymous reviewers for their very thorough and very useful feedback that helped improve this article.

7. Additional Experiments and Considerations

In the following, we first describe how ProB's SMT solver and its integration of Z3 handle deferred sets as is done by ProB. Further, we discuss drawbacks of Z3 regarding memory consumption and mention peculiarities of the new constraint solving backends regarding well-definedness.

One finding in Section 6.7.1 was that Z3 often lacks performance for constraints involving quantified formulas. We thus present an additional quantifier instantiation to investigate the impact on the performance of ProB's different constraint solving backends.

During the development of ProB's interface to Z3, we noted that Z3's performance and soundness in solving constraints using lambda functions fluctuates. After we identified a performance regression in a new version of Z3, a developer of Z3 stated that "nested existentials under lambdas were not tested and sound in earlier versions" [221], which surprised us. We thus revisit the empirical evaluation presented in Section 6.7.1 to investigate the impact of recent changes in Z3 on its performance in solving translated B and Event-B constraints. Besides that, we present a quantitative empirical evaluation of ProB's constraint solving backends involving more benchmarks than was the case in Section 6.7.

Last but not least, we describe how we assessed the soundness of the new constraint solving backends.

7.1. Unfixed Deferred Sets

The B language allows defining custom types as deferred or enumerated sets. While enumerated sets are finite and their elements are explicitly defined by name, deferred sets are not limited in size. However, ProB assumes deferred sets to be non-empty during proof and also finite for animation. The size of deferred sets is determined when loading a machine in ProB, which can be set by option as described in Section 5.1. In this case, a deferred set is referred to as an unfixed deferred set. A fixed deferred set, on the other hand, is a deferred set whose cardinality is explicitly defined in a B machine's properties. When finding a contradiction in a constraint that uses an unfixed deferred set, ProB's constraint solver is not able to decide for the satisfiability but indicates that no solution could be found due to the use of unfixed deferred sets (unknown). The reason is that the constraint could be satisfiable when changing the assumed size of unfixed deferred sets. For instance, consider a B machine defining an unfixed deferred set S whose cardinality is assumed to be three by option. ProB's constraint solver is then not able to decide for the satisfiability of the membership to the total surjection $f \in \{1,2\} \twoheadrightarrow S$ since S is an unfixed deferred set and the constraint is satisfiable if assuming a cardinality of two.

In ProB's SMT solver, there are two possibilities for finding contradictions that might be affected by the use of unfixed deferred sets, which are both referred to as theory conflicts in SMT solving. First, it can be the case that the theory solver, *i.e.*, ProB's constraint solver, refutes a single reified constraint that uses at least one unfixed deferred set. Second, it can be the case that a contradiction is detected after grounding domains in ProB's constraint solver, *i.e.*, after the SAT solver has found a solution for a formula's Boolean abstraction, while at least one unfixed deferred set is used in the overall theory constraint. We deem such contradictions to be spurious counterexamples and log their occurrence programatically. Apart from that, we proceed as usual in CDCL and learn from such contradictions. If finding a solution for the overall constraint, learning from spurious counterexamples is irrelevant. Yet, if the overall result of SMT solving is a contradiction and at least one spurious counterexample has been found during the search, the SMT solver cannot decide for the satisfiability and indicates that no solution could be found due to the use of unfixed deferred sets (unknown). This behavior is similar to the one of ProB's constraint solver.

A considerable difference between ProB's SMT solver and its constraint solver is that the SMT solver is able to refute constraints that use unfixed deferred sets. First, it can be the case that a contradiction is found by the SAT solver without entering the theory solver. For instance, consider the extended constraint from above $f \in \{1,2\} \twoheadrightarrow S \wedge x = 1 \wedge x \neq 1$, where S is an unfixed deferred set whose cardinality is again assumed to be three. The contradiction $x = 1 \wedge x \neq 1$ can be detected by the SAT solver since this subformula is, *e.g.*, abstracted to $\mathrm{A} \wedge \neg\mathrm{A}$ with $\mathrm{A} \equiv x = 1$. This also applies for the integration of Z3 since we check a formula's Boolean abstraction prior to the translation to SMT-LIB as described in Section 6.7.1. ProB's constraint solver, on the other hand, is not able to decide for the satisfiability due to the use of the unfixed deferred set S. Second, it can be the case that clause learning leads to finding a contradiction but without propagating any constraint containing an unfixed deferred set.

For ProB's integration of Z3, we provide an option `z3_solve_for_animation` in which case the cardinality of unfixed deferred sets in SMT-LIB is defined according to the size assumed by ProB. That means, Z3 behaves as ProB's constraint solver and is possibly not able to decide for the satisfiability of a constraint if a contradiction has been found and unfixed deferred sets are used. This mode is particularly useful for animating a model's state space. If the option is set to false, Z3 does not restrict the size of unfixed deferred sets and thus possibly behaves differently than ProB's constraint and SMT solver. This mode can be used for proving and disproving constraints. If a solution has been found for a constraint but Z3 used a different deferred set size than is assumed by ProB, a warning is presented to the user indicating that this solution does not correspond to ProB's current settings for deferred sets.

7.2. Memory Consumption of Z3

During the development of ProB's interface to Z3, we noted that Z3 occasionally allocates memory that is not freed during the lifetime of our application. In the worst case, this leads to an out-of-memory exception that kills the current system process. We further observed that Z3 occasionally kills its system process due to reaching unexpected code (assertion violation). Since the current integration of Z3 in ProB runs on the main thread, this also leads to ProB being killed. Another finding is that Z3 sometimes consumes a lot of memory when instantiating quantifiers as described in Section 7.6.1, which can also lead to throwing an out-of-memory exception or killing the complete process [222]. We were able to improve but not remove this issue by throttling specific parameters of Z3, *e.g.*, limiting the maximum amount of iterations and instantiations for model-based quantifier instantiation.

We analyzed our implementation that uses Z3's C++ interface using Valgrind [223]. The tool did not find any memory leak or vulnerability in our implementation when running a subset of ProB's unit and integration tests that use Z3. Unfortunately, Valgrind slows down an application by several orders of magnitude. This makes it difficult to investigate and reproduce memory leaks for large constraints. For instance, Z3 seems to sometimes leak memory when solving dozens of constraints that use many quantifiers, strings, and integer arithmetic at once. Further, the parallel use of different Z3 solvers aggravates this issue. We were not able to find a minimal reproducible example for which the analysis using Valgrind is feasible. Fortunately, the Z3 community is great and vigilant. We found an issue in Z3's Github repository that discusses a memory leak when disposing a Z3 context [224]. Another user further emphasized this issue when using several Z3 solvers in parallel (see Figure C.1), which corresponds to our implementation as described in Section 6.4. Nikolaj Bjørner, one of Z3's main developers, confirmed the issue's existence and stated that the error's location „within std::unordered_map constructor is very weird“ (see Figure C.2). As of November 2023, the issue has not been solved yet. One conclusion of the discussion is that it could possibly be caused by a compiler bug or an error in the native C++ library for unordered maps in the release builds from 2017.

ProB's interface to Z3 currently uses threads for parallelization that operate in the same system process as ProB. In retrospective, it would have been beneficial for ProB to run all instances of Z3 in their own system process. Z3's memory management would then not have any effect on ProB. For the future, we suggest using the C interface of 0MQ [225] to distribute the use of Z3 in ProB across several system processes. 0MQ, an open-source networking library and concurrency framework, provides sockets that allow sending atomic messages between system processes. In our application, we send abstract syntax trees of B and Event-B constraints to Z3. To use 0MQ, we thus have to provide a wrapper for each function provided by our C++ interface to Z3 which receives atomic data, *e.g.*, a string, and transforms the data accordingly. This task is not difficult but requires great diligence. Yet, it should be noted that our current implementation that runs on the same system process as ProB is almost certainly faster than the new proposal since no communication between different system processes is required.

$$\tau(\mathrm{p}) \mathrel{\widehat{=}} \mathrm{WD}(\mathrm{p}) \wedge \mathrm{p}$$

$$\tau(\mathrm{q}_1 \wedge \ldots \wedge \mathrm{q}_n) \mathrel{\widehat{=}} \tau(\mathrm{q}_1) \wedge \ldots \wedge \tau(\mathrm{q}_n)$$

$$\tau(\mathrm{q}_1 \vee \ldots \vee \mathrm{q}_n) \mathrel{\widehat{=}} \tau(\mathrm{q}_1) \vee \ldots \vee \tau(\mathrm{q}_n)$$

$$\tau(\mathrm{q}_1 \Rightarrow \mathrm{q}_2) \mathrel{\widehat{=}} \tau(\mathrm{q}_1) \Rightarrow \tau(\mathrm{q}_2)$$

$$\tau(\mathrm{q}_1 \Leftrightarrow \mathrm{q}_2) \mathrel{\widehat{=}} \tau(\mathrm{q}_1) \Leftrightarrow \tau(\mathrm{q}_2)$$

$$\tau(\lambda(x_1, \ldots, x_n).(\mathrm{q} \mid \mathrm{e})) \mathrel{\widehat{=}} \\ \quad \lambda(x_1, \ldots, x_n).(\mathrm{WD}(\mathrm{e}) \wedge \mathrm{q} \mid \mathrm{e})$$

$$\tau(\exists(x_1, \ldots, x_n).(\mathrm{q})) \mathrel{\widehat{=}} \exists(x_1, \ldots, x_n).(\tau(\mathrm{q}))$$

$$\tau(\forall(x_1, \ldots, x_n).(\mathrm{q}_1 \Rightarrow \mathrm{q}_2)) \mathrel{\widehat{=}} \\ \quad \forall(x_1, \ldots, x_n).(\tau(\mathrm{q}_1 \Rightarrow \mathrm{q}_2))$$

$$\tau(\{x_1, \ldots, x_n \mid \mathrm{q}\}) \mathrel{\widehat{=}} \{x_1, \ldots, x_n \mid \tau(\mathrm{q})\}$$

$$\tau(\mathrm{IF}\ \mathrm{q}_1\ \mathrm{THEN}\ \mathrm{q}_2\ \mathrm{ELSE}\ \mathrm{q}_3\ \mathrm{END}) \mathrel{\widehat{=}} \\ \quad \mathrm{IF}\ \tau(\mathrm{q}_1)\ \mathrm{THEN}\ \tau(\mathrm{q}_2)\ \mathrm{ELSE}\ \tau(\mathrm{q}_3)\ \mathrm{END}$$

$$\tau(\Sigma(x_1, \ldots, x_n).(\mathrm{q} \mid \mathrm{e})) \mathrel{\widehat{=}} \\ \quad \Sigma(x_1, \ldots, x_n).(\mathrm{WD}(\mathrm{e}) \wedge \tau(\mathrm{q}) \mid \mathrm{e})$$

$$\tau(\Pi(x_1, \ldots, x_n).(\mathrm{q} \mid \mathrm{e})) \mathrel{\widehat{=}} \\ \quad \Pi(x_1, \ldots, x_n).(\mathrm{WD}(\mathrm{e}) \wedge \tau(\mathrm{q}) \mid \mathrm{e})$$

$$\tau(\bigwedge(x_1, \ldots, x_n).(\mathrm{q} \mid \mathrm{e})) \mathrel{\widehat{=}} \\ \quad \bigwedge(x_1, \ldots, x_n).(\mathrm{WD}(\mathrm{e}) \wedge \tau(\mathrm{q}) \mid \mathrm{e})$$

$$\tau(\bigvee(x_1, \ldots, x_n).(\mathrm{q} \mid \mathrm{e})) \mathrel{\widehat{=}} \\ \quad \bigvee(x_1, \ldots, x_n).(\mathrm{WD}(\mathrm{e}) \wedge \tau(\mathrm{q}) \mid \mathrm{e})$$

Figure 7.1.: A formal description of our syntax-directed rules for adding all necessary well-definedness conditions to B or Event-B predicates represented by the function τ. p is an arbitrary predicate which is not a quantified formula, conjunction, disjunction, implication or equivalence, q is an arbitrary predicate, e is an arbitrary expression, and WD is a function returning a conjunction of a predicate's or expression's well-definedness conditions.

7.3. Well-Definedness

ProB's SMT solver enforces the well-definedness of constraints as described in Section 6.5.4. The same applies for the integration of Z3, which, *e.g.*, asserts divisors to be unequal to zero. This changes the semantics compared to ProB's constraint solver in cases of a not well-defined input. In particular, it can be the case that the SMT solver is able to solve a constraint while ProB's constraint solver returns an error stating that the input formula is not well-defined. For instance, the integration of Z3 reports unsatisfiability for the formula $x \in \mathbb{Z} \wedge 1 = 1/x$ since the well-definedness condition $x \neq 0$ is added automatically while ProB reports a well-definedness error [226]. Both ProB's SMT solver and its integration of Z3 thus present the user a message indicating that well-definedness conditions have been added. Here, the generation of well-definedness conditions ensuring the finiteness of sets are skipped. For instance, one well-definedness condition of card(S) is S $\in \mathbb{F}$(S). This condition is irrelevant for constraint solving in ProB since all of its constraint solving backends report a (virtual) timeout if S $\in \mathbb{F}$(S) cannot be proven. Further, well-definedness conditions defining the type of variables are skipped since we work with typed abstract syntax trees, which already contain all necessary typing information.

In Figure 7.1, we present a formal description of our syntax-directed rules for adding all necessary well-definedness conditions to B or Event-B predicates represented by the function τ. It should be noted that so-called LET-expressions and if-then-else expressions (not standard B, only supported by ProB) as well as LET-predicates are expanded prior to the application of τ. These rules are an addition to Section 6.5.4, and describe how we actually enforce the well-definedness of B and Event-B constraints before applying ProB's SMT solver and its integration of Z3. Full constraints are transformed for Z3 while we apply τ for each predicate that is reified with ProB's constraint solver for ProB's SMT solver.

It is trivial to see that adding well-definedness conditions to an already well-defined B or Event-B predicate using τ cannot change its semantics. For this, an adequate hypothesis is that a well-defined predicate contains all its necessary well-definedness conditions. Otherwise, the predicate would not be well-defined. The concept of well-definedness in B and Event-B requires that well-definedness conditions are conjuncted in front of the predicate entailing the conditions. When adding well-definedness conditions to a well-defined B or Event-B predicate according to the rules described in Figure 7.1, these well-definedness conditions are duplicates since they are already present according to the hypothesis. The well-definedness conditions added by τ can thus be removed. Therefore, the presented rules result in logical equivalent formulas when applied to well-defined B or Event-B predicates.

7.4. Revisiting Empirical Evaluation

The benchmarks presented in Section 6.7.1 do not use B's relational iteration or closure and are thus not affected by Z3's missing support for lambda functions inside recursive definitions as described in the corrigendum in Section 6.4.1. To update and revisit the benchmarks presented in Table 6.1, we used the same system and software settings but Z3 version `4.12.3` built from commit `cc4ac0e6`, ProB version `1.12.2` built from commit `05f1e64c`, and SICStus Prolog version `4.8.0`. The reason for using a pre-release version of Z3 is that we found and reported a bug in Z3 version `4.12.2` [177], where an erroneous rewriting rule implementing destructive equality resolution lead to finding solutions for unsatisfiable formulas. The bug has been fixed but a stable release of version `4.12.3` was not present at the time of writing this thesis.

The evaluated benchmarks can be seen in Table 7.1. A corresponding Venn diagram can be seen in Figure 7.2. First and foremost, it can be seen that the overall performance of the parallel integration of Z3 has decreased compared to using version `4.8.16` as was the case in Section 6.7.1. Yet, we do not attribute these differences to changes in Z3 but rather to the consideration of unfixed defered sets. For the benchmarks numbered 10 and 11, Z3 is not able to solve most constraints due to the use of unfixed deferred sets, which were not considered in Section 6.7.1. In total, 546 constraints can be solved using the constructive translation from B to SMT-LIB, which are 6 constraints more compared to using Z3 version `4.8.16`. The total runtime when using the constructive translation has increased, which might be attributed to the fact that additional implementations for lambda functions have been provided in Z3 fixing prior bugs and making Z3 less likely to give up constraint solving (unknown). Besides the benchmarks numbered 10 and 11, the performance of Z3 when using the axiomatic translation has improved, *e.g.*,

Table 7.1.: The BMC constraints presented in Table 6.1 but using updated versions of ProB, Z3, and SICStus Prolog.

			ProB-Z3				
No.	Name	ProB	(axiomatic)	(constructive)	(parallel)	(parallel & decomposed)	∅ Components
1	Prisoners-4	8 / 2196 s	11 / 31 s	**12 / 106 s**	12 / 114 s	12 / 204 s	6
2	Bakery	4 / 2771 s	0 / 2886 s	**5 / 2319 s**	5 / 2585 s	5 / 2608 s	1
3	Paxos-3	**2 / 2895 s**	0 / 352 s	0 / 269 s	0 / 434 s	0 / 493 s	2
4	SimpleTwoPhase	**26 / 0.36 s**	25 / 1 s	26 / 1 s	26 / 1 s	26 / 1 s	29
5	TravelAgency	**10 / 2226 s**	1 / 1236 s	1 / 76 s	1 / 1336 s	1 / 1124 s	10
6	LargeBranching	**26 / 1 s**	26 / 59 s	26 / 47 s	26 / 83 s	26 / 125 s	2
7	SearchEvents	3 / 2764 s	2 / 136 s	**20 / 797 s**	20 / 800 s	20 / 820 s	6
8	ABZ16_m4	**26 / 0.03 s**	26 / 0.09 s	26 / 0.09 s	26 / 0.23 s	26 / 0.81 s	11
9	ABZ16_m5	**25 / 0.02 s**	25 / 0.09 s	25 / 0.06 s	25 / 0.14 s	25 / 0.71 s	11
10	ABZ16_m6*	0 / 73 s	2 / 117 s	**2 / 91 s**	2 / 122 s	1 / 151 s	12
11	ABZ16_m7*	0 / 82 s	2 / 134 s	**2 / 108 s**	2 / 129 s	1 / 173 s	13
12	R0_GearDoor	**26 / 0.01 s**	26 / 0.04 s	26 / 0.04 s	26 / 0.16 s	26 / 0.06 s	1
13	R1_Valve	**26 / 0.01 s**	26 / 0.03 s	26 / 0.03 s	26 / 0.08 s	26 / 0.29 s	5
14	R2_Outputs	**26 / 0.01 s**	**26 / 0.01 s**	**26 / 0.01 s**	26 / 0.06 s	26 / 0.67 s	11
15	R3_Sensors	12 / 1781 s	**26 / 22 s**	26 / 23 s	26 / 24 s	26 / 23 s	17
16	R4_Handle	6 / 2691 s	5 / 1234 s	**6 / 885 s**	5 / 1185 s	5 / 1253 s	17
17	R5_Switch	9 / 2213 s	20 / 86 s	**20 / 81 s**	19 / 85 s	19 / 115 s	25
18	R6_Lights	7 / 2439 s	25 / 110 s	25 / 108 s	25 / 110 s	**26 / 147 s**	31
19	Lightbot	3 / 2769 s	2 / 1290 s	12 / 1639 s	**12 / 1816 s**	12 / 1821 s	12
20	M0_AAI	2 / 2890 s	26 / 55 s	**26 / 18 s**	26 / 23 s	26 / 30 s	6
21	M0_AAT	2 / 2890 s	26 / 46 s	**26 / 17 s**	26 / 21 s	26 / 27 s	6
22	M0_AOO	3 / 2766 s	**26 / 8 s**	**26 / 8 s**	26 / 9 s	26 / 14 s	3
23	M0_VOO	3 / 2765 s	**26 / 7 s**	26 / 6 s	26 / 8 s	26 / 12 s	3
24	M0_VVI	2 / 2888 s	26 / 41 s	**26 / 13 s**	26 / 17 s	26 / 22 s	6
25	M0_VVT	2 / 2890 s	26 / 65 s	**26 / 22 s**	26 / 25 s	26 / 35 s	6
26	M1_AOOR	3 / 2766 s	26 / 18 s	**26 / 15 s**	26 / 21 s	26 / 25 s	13
27	M1_VOOR	3 / 2764 s	26 / 14 s	**26 / 13 s**	26 / 16 s	26 / 21 s	12
28	M2_AAI	2 / 2889 s	26 / 30 s	**26 / 12 s**	26 / 14 s	26 / 16 s	8
Total		267 / 50 415 s	510 / 7983 s	**546 / 6679 s**	544 / 8983 s	543 / 9265 s	

Solved constraints / Runtime s
* many unknowns due to the use of unfixed deferred sets

for the benchmarks numbered 1 and 16. One considerable difference compared to the empirical evaluation presented in Section 6.7.1 is that the axiomatic translation is now not able to solve any of the selected benchmarks that cannot be solved when using the constructive translation as can be seen in Figure 7.2. Further, Z3 is able to solve two more constraints when using the constructive translation that cannot be solved when parallelizing the use of both translations. It is not clear why this is the case since we set random seeds for Z3 which should result in deterministic behavior. Yet, we are not absolutely sure that this is always guaranteed.

ProB's constraint solver is able to solve fewer constraints compared to the results presented in Table 6.1. For the benchmarks numbered 10 and 11, ProB is not able to find a solution due to the use of unfixed deferred sets, which was not the case in Section 6.7.1. While this needs to be investigated, it is not a loss of performance but rather a configuration issue. The performance actually dropped for the benchmarks

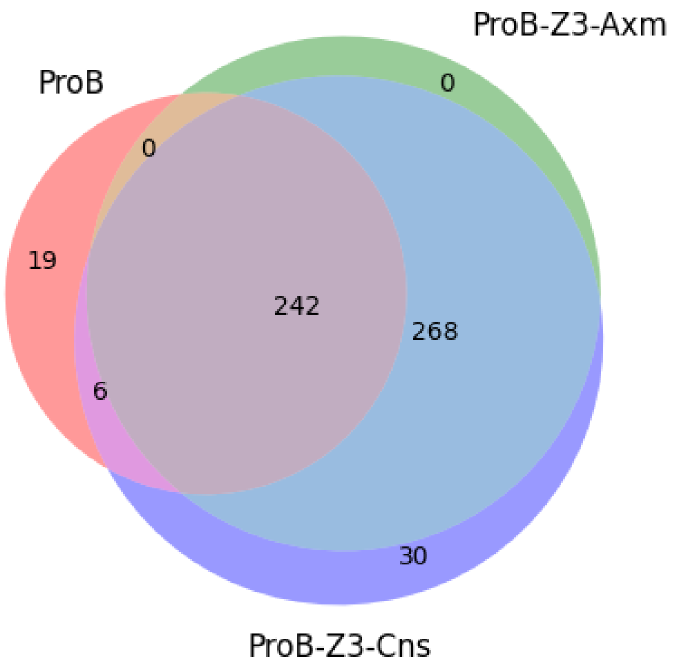


Figure 7.2.: A Venn diagram showing the amount of BMC constraints that can be solved by ProB's constraint solver and the different configurations of Z3 using version `4.12.3` built from commit `1d62964c` as can be seen in Table 7.1.

numbered 2 and 5.

All in all, concerns about a possible loss of performance for Z3 when using lambda functions have not materialized.

7.5. Quantifier Instantiation

Finite quantifiers can be transformed to equivalent formulas that do not use a quantification which is referred to as quantifier instantiation. A universal quantifier can be rewritten as a conjunction of predicates corresponding to each combination of quantified variables. For an existential quantifier, a disjunction is used. For instance, the universal quantification $\forall x.(x \in 1..2 \mid \forall y.(y \in 1..2 \mid r(x) + (y - x) \neq r(y))$ can be unfolded to the conjunction $(r(1) + (1 - 1) \neq r(1)) \wedge (r(1) + (2 - 1) \neq r(2)) \wedge (r(2) + (1 - 2) \neq r(1)) \wedge (r(2) + (2 - 2) \neq r(2))$. We observed that the Z3 constraint solver has difficulties in solving quantified formulas (see Section 6.7.1), and expect that the instantiation of quantifiers improves the performance of ProB's integration of Z3. In the case of ProB's SMT solver, quantifier instantiation changes a lot as well. A quantifier is usually abstracted as a single SAT variable in DPLL(T) (or CDCL(T)) as described in Section 6.5.2. If instantiating quantifiers, several SAT variables are introduced for the different predicates that have been instantiated from the quantifier. For instance, in the example from above, four SAT variables are introduced when applying quantifier instan-

Listing 7.1: B encoding of the n queens problem using two universal quantifiers that can be transformed by quantifier instantiation.

```
n = 10 &
queens : 1..n >-> 1..n &
!(q1).(q1:1..n =>
  !(q2).(q2:2..n & q2>q1
            => queens(q1)+(q2-q1) /= queens(q2) &
               queens(q1)+(q1-q2) /= queens(q2)))
```

Table 7.2.: Performance evaluation of different configurations of the n queens problem comparing ProB's constraint solver, its parallel integration of Z3 (ProB-Z3), and its SMT solver (ProB-SMT) with (suffix QI) and without quantifier instantiation.

n	ProB	ProB-Z3	ProB-SMT	SAT variables	ProB-QI	ProB-Z3-QI	ProB-SMT-QI
4	0.10 s	0.16 s	0.04 s	14	0.10 s	0.16 s	0.04 s
6	0.01 s	1.70 s	0.01 s	32	0.01 s	0.10 s	0.01 s
8	0.01 s	0.09 s	0.01 s	58	0.01 s	0.15 s	0.01 s
10	0.01 s	1.98 s	0.01 s	92	0.01 s	0.27 s	0.01 s
12	0.02 s	3.67 s	0.01 s	134	0.02 s	0.46 s	0.01 s
14	0.02 s	12.39 s	0.01 s	184	0.02 s	0.79 s	0.01 s
16	0.03 s	27.35 s	0.01 s	242	0.03 s	1.19 s	0.01 s
18	0.04 s	29.28 s	0.01 s	308	0.04 s	1.93 s	0.01 s
20	0.05 s	-	0.01 s	382	0.04 s	3.85 s	0.01 s

tiation. Otherwise, only one SAT variable is used for the universal quantification. While the instantiation of quantifiers can improve the performance, it can also slow down the search depending on the size of finite domains. For instance, unfolding a quantifier that results in thousands of instantiations might not be beneficial. For ProB's constraint, we do not expect performance improvements of quantifier instantiation since its theory solver is already able to detect if quantified domains are finite and can be unfolded.

We implemented quantifier instantiation for universally and existentially quantified formulas in ProB. For this, we use a limit for the maximum amount of a quantifier's instantiations to be unfolded, which can be computed by inspecting the domains of quantified variables. For the integration of Z3, the quantifier instantiation is applied after rewriting B or Event-B formulas that have no direct counterpart in SMT-LIB using quantifiers. Listing 7.1 shows a B encoding of the n queens problem using two universal quantifiers that can be transformed by quantifier instantiation. We use this example as a micro benchmark for an evaluation of the quantifier instantiation using a timeout of 1 min for each constraint solver. A performance evaluation of different configurations of the n queens problem comparing ProB's constraint solver, its integration of Z3 (ProB-Z3), and its SMT solver (ProB-SMT) using static syntax analysis with (suffix QI) and without quantifier instantiation can be seen in Table 7.2. We also state the amount of unique predicates occurring in a formula after quantifier instantiation (fifth column in

Table 7.3.: Performance comparison of ProB's constraint solver, its backend to Kodkod, its parallel integration of Z3 (ProB-Z3), and its SMT solver (ProB-SMT) for checking Alloy models with (suffix QI) and without quantifier instantiation.

		Runtime in ms							
			ProB						
No.	Model	ProB	Kodkod	Z3	SMT	QI	Kodkod-QI	Z3-QI	SMT-QI
1	einstein_puzzle	14 642	2158	> 300 000	10 977	29 348	3377	> 300 000	2530
2	crewalloc	812	867	unknown	76	1734	2076	unknown	172
3	abstract_memory	> 300 000	unf. (29)	unknown	> 300 000	> 300 000	> 300 000	unf. (834)	> 300 000
4	cache_memory	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000	> 300 000
5	railway	> 300 000	unf. (83)	unknown	> 300 000	> 300 000	> 300 000	unknown	> 300 000
6	dijkstra_2_process	9	24	unknown	19	11	36	369	16
7	chord	> 300 000	> 300 000	unknown	> 300 000	> 300 000	> 300 000	unknown	> 300 000
8	handshake	> 300 000	387	> 300 000	> 300 000	> 300 000	> 300 000	unknown	2552
9	farmer	> 300 000	> 300 000	unknown	> 300 000	> 300 000	> 300 000	unknown	> 300 000
10	overlapping_ranges	27	43	440	287	28	86	> 300 000	269
Total Solved Constraints		4	5	1	4	4	4	1	5

Table 7.2), which is the amount of SAT variables introduced for the Boolean abstraction in SMT solving. The original formula without quantifier instantiation has three unique predicates for all configurations of n.

First and foremost, it can be seen that both ProB's constraint solver and ProB's SMT solver have no difficulties in solving the benchmarks. The integration of Z3, on the other hand, shows performance drawbacks for an increasing size of quantified domains. For $n = 20$, the integration of Z3 is not able to solve the constraint in 1 min. When instantiating the quantifiers, the resulting formula has 382 unique predicates for $n = 20$ and can be solved in around 4 s by Z3. All in all, instantiating quantifiers increases the performance of Z3 by several orders of magnitude for the selected benchmarks. This is an interesting fact since the integration of Z3 already uses Z3's model-based quantifier instantiation. It seems that quantified formulas remain a difficult task for Z3. In contrast to that, the instantiation of quantifiers neither has a positive nor negative effect on the performance of ProB's constraint solver or its SMT solver. Yet, this might be caused by the already small runtime of both constraint solvers for the selected benchmarks.

In order to provide a more sophisticated performance evaluation of the quantifier instantiation's effect on the performance of constraint solving, we provide an evaluation of a subset of constraints for checking Alloy commands in B used in Section 5.2. We experienced that Alloy models often use quantifiers and thus, deem these constraints to be adequate for this task. A solver timeout of 5 min was used. The benchmarks were run on a system with an Intel Core I7-8750H CPU (2.2GHz) and 16 GB of RAM using ProB version `1.12.0`, SICStus Prolog version `4.7.0`, and Z3 Version `4.12.3` built from commit `1d62964c`. The evaluated benchmarks can be seen in Table 7.3.

It can be seen that the instantiation of quantifiers does not fundamentally improve the constraint solvers' performance for the selected benchmarks. For ProB's constraint solver and its backend to Kodkod, instantiating quantifiers increases the total runtime.

In particular, for the first benchmark in the case of ProB, and for the third and fifth benchmark in the case of the backend to Kodkod. These results are as expected since ProB already detects finite quantifiers that can be unfolded, and Kodkod is also not known for having difficulties in solving finite quantifiers. It appears to be better to not use quantifier instantiation for both solvers. For ProB's SMT solver, instantiating quantifiers improves the runtime for the first and eighth benchmark. In total, one more constraint can be solved when using quantifier instantiation. Here, it should be noted that the quantifier instantiation leads to more Boolean variables in the SAT solver, which can improve the search compared to using a single Boolean variable for a complete quantifier, *e.g.*, by leading to more conflicts and clause learning. The integration of Z3 is able to solve two more constraints when instantiating quantifiers, namely the benchmarks numbered three and six. For the third benchmark, a contradiction has been found but Z3 can actually not decide for the satisfiability due to the use of unfixed deferred sets. Z3's performance decreased for the tenth benchmark. Overall, instantiating too many quantifiers does not seem to be beneficial. Due to the performance improvement presented in Table 7.2, we decided to apply quantifier instantiation for Z3 by default. However, we use a limit for unfolding quantifiers which can be set via an option in ProB.

In Z3, it is possible to provide triggers for universal quantifiers defining patterns that are used to find relevant instances, which is also known as E-matching [227]. In the future, we want to investigate whether the use of triggers is more performant than instantiating universal quantifiers prior to the translation to SMT-LIB, *e.g.*, as was recently presented by Rosalie Defourné for TLA$^+$ [228]. Here, we can use our quantifier instantiation to generate appropriate triggers.

7.6. Additional Empirical Evaluation

The empirical performance evaluation presented in Section 6.7 and Section 7.4 served the purpose of showing general benefits of ProB's constraint solver, its integration of Z3, and its SMT solver. In order to gain a deeper insight in the strengths of the different constraint solving backends, we provide a rather quantitative than qualitative empirical evaluation in the following. For this, we use 150 classical B, Event-B, and TLA$^+$ models that are publicly available in ProB's specification repository [229]. We created constraints from bounded model checking and constraint-based checking as done in Section 6.7.[1] Each model has at least 25 states, 1 invariant, and 1 machine operation or event. Around 75% of the machines have at least 5 machine operations or events and at least 10 invariants.

For the following empirical evaluations, we used the same software and system settings as in Section 7.4. Unfortunately, we experienced several exceptions when using Z3 for the selected benchmarks (out-of-memory, assertion violation, segmentation fault). Throttling specific parameters of Z3 as described in Section 7.2 has mitigated the occur-

[1]The benchmarks can be found in the following Github repository to reproduce the results: `https://github.com/Joshua27/cbc_benchmarks/`

rence of out-of-memory exceptions during quantifier instantiations. Yet, it can still be the case that Z3 runs out of memory when generating quantifier instantiations. Further, ProB's SMT solver ran out of memory for a few benchmarks, which needs to be investigated more thoroughly. In the worst case, an out-of-memory exception of Z3 leads to killing the main thread including ProB. This is always the case for an assertion violation and segmentation fault since they cannot be caught by exception handling. Corresponding benchmarks were removed from the evaluation.

ProB implements common subexpression elimination as described in Section 2.6. This technique is similar to SMT solving in the sense that common variables and predicates are pooled in order to be only solved once. Further, ProB's constraint solver provides an additional backend defining constraint-handling rules (CHR) to improve the performance for finding contradictions. We consider both additional backends in the following evaluation. Last but not least, we also use static symmetry breaking for ProB's constraint solver to investigate the impact of symmetry breaking independently of SMT solving.

7.6.1. Bounded Model Checking

We used ProB to check 133 different models for a depth of 1, 5, and 10 state changes by generating corresponding monolithic BMC constraints and searching for a solution, *i.e.*, a counterexample to a machine invariant. In total, we gathered 399 distinct constraints. The constraints of all benchmarks have an average amount of 196 unique conjuncts or disjuncts, a median amount of 175, and a maximum amount of 2349. The evaluated benchmarks can be seen in Table D.1 and Table D.2.

Figure 7.3 shows a Venn diagram including the best performing configurations of ProB's constraint solver, its integration of Z3, and its SMT solver. It can be seen that each constraint solver has its own benefits while the parallel integration of Z3 is able to solve the most amount of constraints. In total, 60 constraints can be solved using Z3 which cannot be solved using ProB's constraint solver. 58 of these constraints are unsatisfiable and 2 are satisfiable. Some of these constraints use linear integer arithmetic and unbounded domains for which ProB's CLP(FD) backend is not able to identify contradictions within the predefined timeout. This again shows that Z3 has strong theory solvers for linear integer arithmetic, which extend the constraint solving capabilities of ProB. Further, we deem clause learning of Z3 to improve the identification of contradictions compared to ProB's constraint solver. 27 of the constraints that can be solved by ProB's constraint solver but not Z3 are satisfiable, which shows that Z3 often has problems in finding solutions for satisfiable B and Event-B constraints. For instance, Z3 is often not able to decide for the satisfiability of constraints if set cardinality constraints or functions are used as discussed in Section 6.7.1. This was one motivation to provide a direct implementation of SMT solving in ProB using its constraint solver as a theory solver. ProB's constraint solver allows solving 18 constraints that cannot be solved by the other constraint solvers while its SMT solver is able to do so for 2 constraints. In total, the SMT solver allows solving 19 constraints that cannot be solved by Z3, of which 11 constraints are satisfiable. Compared to ProB's constraint solver, the SMT solver

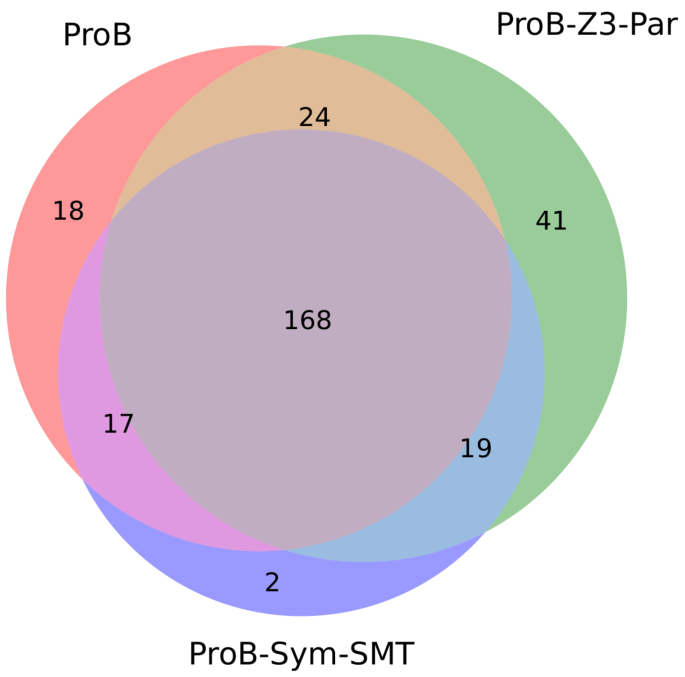


Figure 7.3.: A Venn diagram showing the amount of BMC constraints that can be solved by ProB's constraint solver, its parallel integration of Z3, and its SMT solver using the additional static analysis and symmetry breaking as can be seen in Table D.1 and Table D.2.

allows solving 21 constraints more in total. 20 of these constraints are a contradiction and 1 is satisfiable. Here, we deem clause learning to improve the performance for finding contradictions. Yet, ProB's constraint solver is able to solve 42 constraints that cannot be decided by the SMT solver, of which 28 constraints are satisfiable. The major reason for this difference is that the SMT solver often propagates theory constraints in an order that is inconvenient for ProB's constraint solver. This leads to exceeding the predefined timeout when grounding domains after finding a complete solution for a formula's Boolean abstraction without ever returning to the actual SMT solving routine. For instance, ProB's constraint solver is able to solve the benchmarks numbered 39 to 41 in several milliseconds while its SMT solver exceeds the predefined timeout. It seems that SMT solving is not beneficial if ProB's constraint solver is already able to solve a constraint in a short amount of time. We investigated the use of the SMT solver's additional theory solver for the integer difference logic as described in Section 6.6. Unfortunately, the problems in grounding domains are not caused by the integer difference logic but set theoretical constraints. In the future, we thus want to investigate the integration of an additional theory solver for first-order logic and, in particular, set theory in ProB's SMT solver. For instance, an interface to {log} [230] seems promising.

The constructive translation to SMT-LIB is superior to the axiomatic one for the selected benchmarks since it is able to solve 22 more constraints in about half the time as can be seen in Table D.2. When using the axiomatic translation, Z3 is able to solve

5 constraints that cannot be solved using the constructive translation, which are all satisfiable. 6 constraints that can only be solved using the constructive translation are satisfiable and 21 constraints are unsatisfiable. This shows that using lambda functions instead of quantified formulas improves both proving and disproving of translated B and Event-B constraints. Further, the parallel integration of both translations to SMT-LIB is again justified. The decomposition of constraints into independent components does not allow solving more constraints than the plain parallel integration. In particular, it increases the total runtime and even prevents solving 14 constraints. Here, it can be the case that Z3 failed to solve one component (unknown) leading to an overall failure while it is able to solve the full constraint. In total, the translation to SMT-LIB failed for 16 constraints due to missing translations. For instance, we are not able to efficiently encode the set of all natural numbers in SMT-LIB. The total runtime of the parallel integration of Z3 is considerably larger than the one of the integration that only uses the constructive translation to SMT-LIB. A reason is that we interrupt a solver instance of Z3 if another instance has found a solution and wait for it to return to the main procedure. Z3 only provides a soft interrupt that waits for specific program points in order to clean up the memory. Thus, interrupting a solver instance of Z3 does not result in an immediate interruption.

For ProB's constraint solver, neither using CSE nor using the CHR backend allows solving more constraints. Overall, it can be seen that ProB's native constraint solver is the best choice by default. Static symmetry breaking does not allow solving more constraints compared to ProB's default constraint solver and slightly increases the total runtime due to the computation of symmetry breaking predicates. Here, it should be noted that computing graph automorphisms for symmetry breaking is in the complexity class NP. We thus use a timeout of 10s for computing symmetry breaking predicates. In total, 185 constraints contain symmetries. The maximum amount of computed symmetry breaking predicates is 45, the median is 2, and the mean is 4.9. Generally, we deem symmetry breaking to be useful for finding contradictions and especially when computing all solutions of a constraint. When computing a single solution, breaking symmetries is not necessarily beneficial. For the selected benchmarks, it seems that symmetric constraints are not a difficulty for constraint solving using ProB.

ProB's SMT solver does not allow solving more constraints in total compared to its native constraint solver and its integration of Z3 for the selected benchmarks. The additional static analysis improves both the performance and coverage, which can be seen when comparing the total results of the SMT and Raw-SMT as well as Sym-SMT and Sym-Raw-SMT configurations in Table D.2. When comparing the total results of the SMT and Sym-SMT solver configurations, it can be seen that static symmetry breaking improves both the performance and coverage of SMT solving. Here, it should be noted that small differences in the theory solver can lead to finding other theory conflicts first, which can fundamentally change the search path of the SMT solver.

For the selected benchmarks, ProB's constraint solver is not able to decide for the satisfiability of 39 constraints due to the use of unfixed deferred sets while its SMT solver using static analysis and symmetry breaking is not able to do so for 34 constraints and the parallel integration of Z3 for 44 constraints.

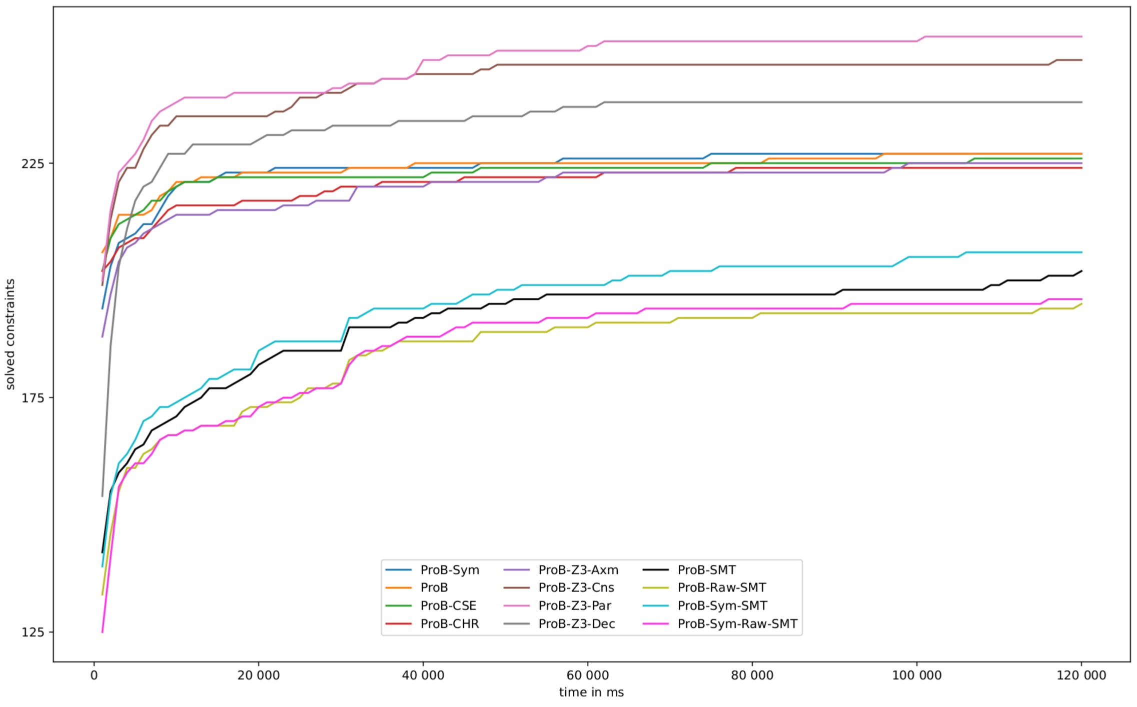


Figure 7.4.: A visualization of the BMC benchmark results presented in Table D.1 and Table D.2 showing the amount of constraints that can be solved by a constraint solver within a specific amount of time.

Figure 7.4 visualizes all considered constraint solvers' runtimes. Most constraint solvers are not able to solve fundamentally more constraints after around 100s. Only ProB's SMT solver configurations are able to solve several more constraints after around 115s, and might be able to solve more constraints when increasing the predefined timeout. Yet, the overall results already show benefits of each constraint solving backend and, in particular, of Z3. When comparing the graphs of the SMT solver configurations with and without the additional static syntax analysis, it can be seen that adding the inferred constraints improves the performance from the beginning. Further, it can be seen that ProB's constraint solver is able to solve many constraints faster than its SMT solver from the beginning. This again suggests that the order of theory constraints propagated by the SMT solver is often suboptimal for ProB's constraint solver.

7.6.2. Deadlock and Inductive Invariant Checking

Besides constraints from bounded model checking, we again use constraints for proving deadlock freedom and the inductiveness of invariants as done in Section 6.7.2. In total, we gathered 112 distinct constraints for deadlock checking, which is equal to the amount of used models. The constraints of all benchmarks have an average amount of 27 unique conjuncts or disjuncts and a median amount of 20. The largest constraint contains 123 unique conjuncts or disjuncts. The evaluated benchmarks can be seen in Table D.3 and

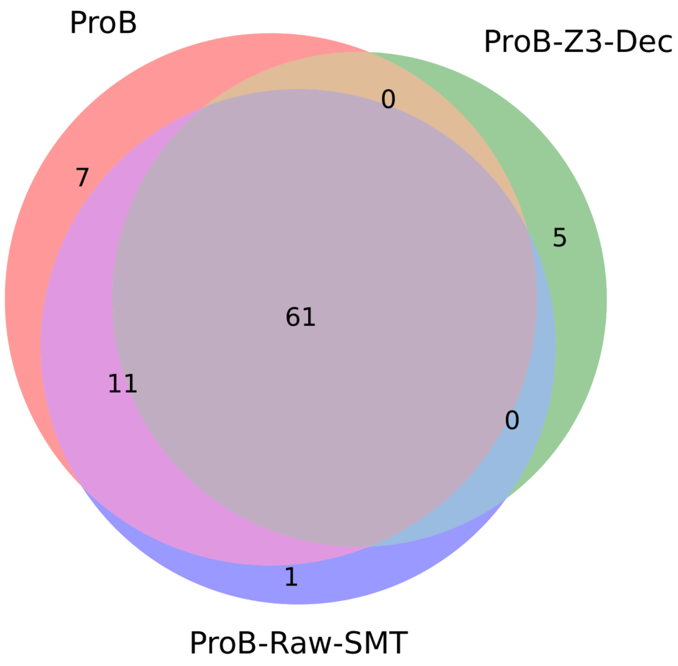


Figure 7.5.: A Venn diagram showing the amount of constraints from deadlock freedom proofs that can be solved by ProB's constraint solver, its parallel integration of Z3 that decomposes constraints into independent components, and its SMT solver without any additional syntax analysis as can be seen in Table D.3.

a corresponding Venn diagram in Figure 7.5.

ProB's constraint solver is able to solve the most amount of constraints for the selected benchmarks. Using CSE or the additional CHR backend does not improve the performance. ProB's SMT solver is able to solve more constraints than the integration of Z3 in total, and is able to solve one constraint that cannot be decided by ProB's constraint solver as can be seen in Figure 7.5. In total, ProB's SMT solver configurations have a runtime that is around one order of magnitude higher than the one of ProB's constraint solver. Again, we assume that the order of theory constraints propagated by the SAT solver is not ideal leading to many conflicts being found. The SMT solver then finally exceeds the predefined timeout while ProB is able to solve the constraints in a short amount of time, *e.g.*, for the benchmarks numbered 34 to 37, 52, 53, 86, 87, and 99. In particular, for the benchmarks numbered 35 to 37, the SMT solver configurations without the static analysis are able to solve the constraints in one second while the additional inferred constraints lead the SMT solver in a different direction resulting in exceeding the predefined timeout.

A total of 34 constraints contain symmetries, one of which contains 18 symmetric variables, which is the maximum amount of symmetric variables. The median amount of symmetric variables in these constraints is 3, and the mean is 3.9. Breaking symmetries neither improves the performance for ProB's constraint solver nor its SMT solver. Again, we deem symmetry breaking to show its full potential when searching for all

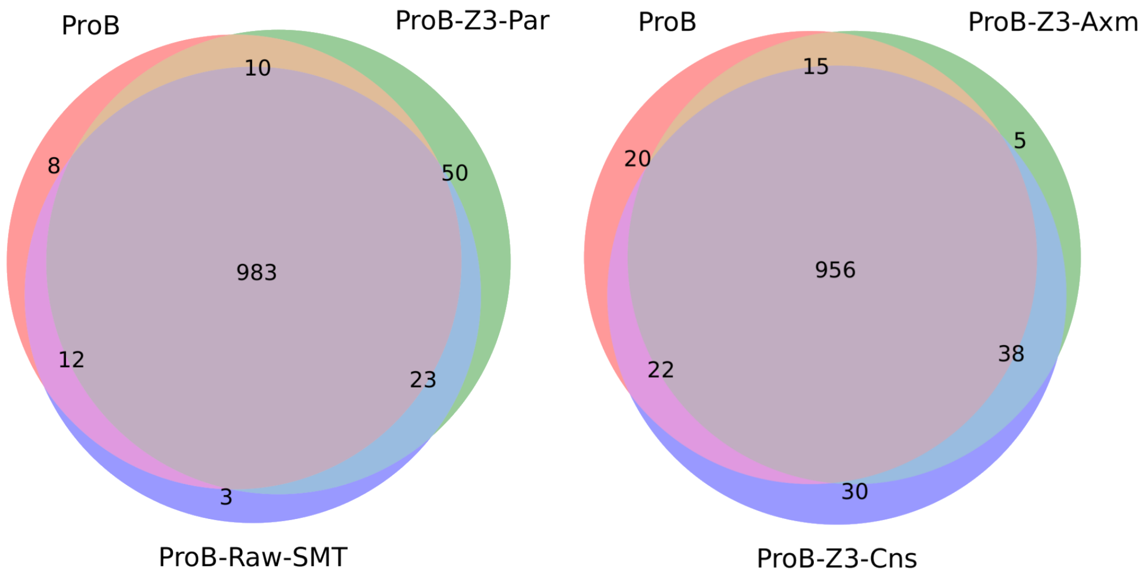


Figure 7.6.: Venn diagrams showing the amount of constraints from inductive invariant proofs that can be solved by ProB's constraint solver, its integrations of Z3, and its SMT solver without any additional static analysis as can be seen in Table D.4.

solutions of a constraint or when finding contradictions. Apparently, contradictions can already be found fast without breaking symmetries for the selected benchmarks.

For the integration of Z3, both translations to SMT-LIB show distinct benefits which again justifies the use of parallelization. The constructive translation to SMT-LIB is more likely to give up constraint solving (unknown), which can be seen when comparing the total runtimes of the axiomatic and constructive solver configurations in Table D.3. The decomposition of constraints into independent components decreases the total runtime by around 200s compared to the plain parallel integration of Z3. Here, we deem that several contradictions were found in one of the first components, which prevents solving the remaining components afterwards. In total, the translation to SMT-LIB failed for one constraint due to the use of unsupported operators when parallelizing both translations.

ProB's constraint solver and its SMT solver configurations are not able to decide for the satisfiability of 19 constraints due to the use of unfixed deferred sets while the parallel integration of Z3 is not able to do so for 21 constraints.

We further gathered 1302 constraints from 116 different models for inductive invariant checking. Here, the benchmarks have an average amount of 59 unique conjuncts or disjuncts, a median amount of 22, and a maximum amount of 353. The evaluated benchmarks can be seen in Table D.4.

For ProB's constraint solver, CSE and the additional CHR backend do not allow solving more constraints and slightly increase the total runtime for the selected benchmarks. The constructive translation to SMT-LIB is superior to the axiomatic one since it allows solving 32 more constraints in total. In particular, using lambda functions enables solving 52 constraints that cannot be decided when only resorting to quantifiers as can be seen in the Venn diagram on the right-hand side of Figure 7.6. Yet, the axiomatic translation also has benefits since it allows solving 20 constraints that cannot be decided when using lambda functions. The decomposition of constraints into independent components (ProB-Z3-Dec) does not improve the performance for the selected benchmarks. ProB's SMT solver without any additional static analysis (ProB-Raw-SMT) is superior to the different configurations of ProB's constraint solver since it allows solving 8 more constraints in total. Further, the SMT solver is able to decide for the satisfiability of 26 constraints that cannot be decided by ProB as can be seen in the Venn diagram on the left-hand side of Figure 7.6. 5 of these constraints are satisfiable and 21 are unsatisfiable. This shows that SMT solving is often better suited for finding contradictions compared to using CLP as is the case for ProB's constraint solver. However, ProB is able to decide for the satisfiability of 18 constraints that cannot be decided by its SMT solver. 6 of these constraints are satisfiable and 12 are unsatisfiable. In the case of the satisfiable constraints, the SMT solver seems to be guided in a direction that is inconvenient for the theory solver as is the case for the benchmarks from bounded model checking presented in Section 7.6.1, which finally leads to exceeding the predefined timeout. Again, we deem that an additional theory solver for first-order logic and set theory might improve this issue. The SMT solving framework enables an easy integration of different theory solvers alongside ProB's constraint solver as presented in Section 6.6. For the unsatisfiable constraints, ProB's constraint solver seems to be able to detect the contradiction when setting up all constraints at once while the SMT solver is not able to check all Boolean assignments within the predefined timeout. For the selected benchmarks, the additional static syntax analysis does not allow solving more constraints and slightly increases the total runtime, which can be seen when comparing the total results of the SMT and Raw-SMT configurations in Table D.4. Surprisingly, static symmetry breaking decreases the performance of SMT solving since it prevents solving several constraints and increases the total runtime. We deem that the additional theory constraints guide the theory solver in a direction that is not beneficial for solving the selected benchmarks. For ProB's constraint solver, static symmetry breaking also does not improve the performance. In total, 556 constraints contain symmetries. The maximum amount of found symmetry breaking predicates is 37, the median is 4, and the mean is 8. Further, ProB's constraint solver is not able to decide for the satisfiability of 79 constraints due to the use of unfixed deferred sets while its SMT solver is not able to do so for 74 constraints and the parallel integration of Z3 for 82 constraints.

In summary, most constraints can be decided by each constraint solving backend as can be seen in the Venn diagrams in Figure 7.5 and Figure 7.6. Yet, each constraint solver is able to solve constraints that cannot be decided by the other backends. This again confirms that using different constraint solvers is beneficial.

7.7. Soundness

One of the most important aspects of constraint solving is soundness. We have neither proven the soundness of ProB's integration of Z3 nor its SMT solver. Different kinds of tests were implemented to ensure certain behavior, which are described in the following. In this context, we would like to draw the reader's attention to a quote from Edsger Wybe Dijkstra: "Testing shows the presence, not the absence of bugs." [1, p.16]

We provide unit tests for many Prolog predicates used in ProB's integration of Z3 and its SMT solver. While unit tests are suitable to test single components, we deem integration tests to be most important for testing a constraint solver. In particular, we provide integration tests that send constraints to a constraint solver using its main interface predicate. We hereby want to answer the following questions for a specific constraint regarding a constraint solver's different possible outcomes:

1. contradiction: Is the constraint indeed unsatisfiable?
2. solution: Is the found solution correct?
3. Was an error exception thrown when solving constraints regardless of the result?

Error exceptions can be thrown by the Prolog interpreter itself or by our own error handling, *e.g.*, if reaching unexpected code. We also catch Z3's runtime exceptions and throw corresponding errors in Prolog.

We implemented a constraint solving routine in ProB for both the integration of Z3 and ProB's SMT solver where the results are checked with ProB's constraint solver according to the first two questions presented above. This routine can also be used via ProB's command-line interface either using the command `cdclt-double-check`, `z3-double-check`, `cdclt-free-double-check` or `z3-free-double-check`. The first two commands solve constraints considering the current machine's state while the other commands solve constraints without considering a state.

As of now, we have implemented 643 integration tests for ProB's integration of Z3 and 683 integration tests for ProB's SMT solver. The constraints stem from logic puzzles, manual tests, (minimal) examples reproducing prior bugs, and benchmarks, *e.g.*, as presented in Section 6.7, to check for consistency and a possible performance regression. The tests are located in ProB's test suite and are executed each night using Gitlab CI/CD[2]. Note that the constraint solvers' results for the benchmarks presented in Section 6.7 have been verified for consistency using ProB's constraint solver as far as it was able to do so within a predefined timeout. This verification was performed in our programming module for benchmarking. We did not include all of these benchmarks in ProB's test suite due to the long time that is necessary to solve the constraints.

ProB's test suite allows setting an option to fail if any error was thrown during a test case, which enables to investigate the third question from above.

To provide a more diverse set of test cases, we decided to use a fuzzer for SICStus Prolog that was tailored towards generating B and Event-B constraints [231]. The

[2]`https://docs.gitlab.com/ee/ci/`

constraints are generated as abstract syntax trees as understood by ProB's Prolog core and accepted by all of ProB's constraint solving backends [231]. The goal of fuzzing is to test a program by generating (semi-)randomized inputs and inspecting the behavior and results. If a fuzzer has found an input for which a test case fails, it shrinks this input to find a minimal reproducible example. We implemented 10 different fuzzing tests for ProB's SMT solver each generating 100 000 randomized B and Event-B constraints. In each test, the fuzzer is guided to generate different kinds of constraints, *e.g.*, using no strings, no sets or a large amount of disjunctions or conjunctions. Further, intentionally not well-defined B and Event-B constraints are generated to ensure that ProB's SMT solver and its integration of Z3 do not return any not well-defined result. We have found fuzzing to be a useful approach to testing. We were able to find several bugs some of which only occurred in rare special cases, which would probably not have been found that easily using manually created test cases. The fuzzing tests are also executed each night using Gitlab CI/CD.

During the development of ProB's SMT solver, we extended ProB's unsatisfiable core computation as described in Section 6.5.4 to optionally use Z3 or ProB's SMT solver for solving constraints. In this process, we noted that it is a useful method for testing a constraint solver to let it compute an unsatisfiable core of a formula and verify that the resulting formula is indeed a contradiction using a different constraint solver.

Part IV.

Conclusion

8. Translating Alloy and Extensions to Classical B

The Alloy language has gained popularity in the formal methods community. We noted that the Alloy Analyzer is able to solve complex relational constraints fast and provides sophisticated features for the analysis and visualization of formal models. Yet, we also recognized benefits of the B language, *e.g.*, when using integers, and ProB regarding tool support. The following research questions thus arose, which could be answered in this thesis.

RQ1: Which steps are necessary to automatically translate Alloy models into B? We presented an automated integration of Alloy 5 in ProB using the Alloy Analyzer's parser and typechecker as well as a custom implementation transforming parsed models into a Prolog representation. In Chapter 4, we presented a formal description of a translation from Alloy 5 and extensions to classical B. The translation rules are implemented in ProB's Prolog core using ProB's typechecker for typing resulting abstract syntax trees. While the presented translation rules cover the complete core language of Alloy 5, there might be models that are currently not supported by our implementation, *e.g.*, due to type errors in B. Such errors will of course be corrected when reported. The Alloy language is weakly typed and allows for a flexible application of operators, *e.g.*, using Alloy's relational join operator. This often requires many special cases for our translation rules to match B's typing. For instance, integers are singleton sets in Alloy and arguments of arithmetic operations are implicitly summarized, which makes it possible to add an integer and a set of integers. In B, this is not possible due to its strong typing.

Further, Alloy provides a universal type, noted `univ`, which has no direct counterpart in B. While it is possible to introduce a corresponding universal type in B using a deferred set, this prevents ProB to apply different techniques such as symmetry breaking since no distinct sets for different signatures are present. In consequence, we decided to only create parent types for specific signatures if necessary as described in Section 4.4.4.

Another flexible feature of Alloy is the keyword `this`, which is not available in B. For translating Alloy to B, we have to make sure to wrap B data into a singleton set or remove such before applying B operators including `this` depending on the underlying value.

Our work has shown that the implementation of an automated translation from Alloy 5 to B is possible but a tedious and error-prone task requiring many special cases due to the different kinds of typings.

In Section 5.3, we have shown that a translation from Alloy 6 to B is also possible.

The new concept of state changes can be translated using B machine variables, and LTL formulas as well as LTL model checking are supported by ProB. The declarative definition of state changes in Alloy 6 requires a preprocessing of Alloy predicates extracting the actual assignments of state variables to achieve an idiomatic translation. The reason is that B uses operational semantics explicitly assigning single variables. Alternatively, B's `ANY` substitution can be used, which does not require the extraction of assignments from Alloy predicates. This translation to B is easier to implement but less readable and performant than single assignments. A difference between Alloy 6 and B is that Alloy allows assigning any state while B only allows assigning the next state in a single machine operation. Therefore, Alloy predicates assigning any previous or future state other than the next state cannot be translated to B. Apart from that, we do not see any syntax elements or semantics that cannot be translated to B automatically. The Alloy Analyzer can again be used for parsing and typechecking Alloy 6 models. However, it should be noted that the recent version for Alloy 6 cannot be used for all Alloy 5 models since a few syntactic changes have been introduced. For instance, the single quote has no semantics in Alloy 5 but defines a mutable variable's next state in Alloy 6. To support both Alloy 5 and Alloy 6 in ProB, we thus have to integrate two different versions of the Alloy Analyzer's parser and typechecker.

In summary, the necessary steps we found for integrating Alloy into ProB include

– using the Alloy Analyzer's parser and typechecker (different versions for Alloy 5 and Alloy 6),

– transforming parsed models into a Prolog representation,

– in case of Alloy 6, extracting variable assignments from stateful Alloy predicates for single assignments, or resorting to a less idiomatic translation using B's `ANY` substitution,

– deciding for a translation of Alloy's universal type,

– providing syntax-directed translation rules for all language constructs in ProB's Prolog core, and

– using ProB's typechecker for verifying the translation's typing.

RQ2: How does ProB compete in checking Alloy models compared to the Alloy toolchain? The empirical evaluation in Section 4.7 has shown that ProB improves the performance of solving integer constraints compared to the Alloy 5 Analyzer by several orders of magnitude. A reason is that the Alloy Analyzer translates constraints to SAT, where integers are encoded as bits. This entails that arithmetic operations are also encoded in propositional logic, *e.g.*, introducing a full-adder. Consequently, the size of CNFs gets larger, and generating a CNF in the Alloy Analyzer can take longer than loading and solving the translated B formula in ProB.

We decided to prevent Alloy's `univ` type in B for efficiency. Instead, our implementation captures signatures that have a common base type and introduces artificial signatures for corresponding base types in the Alloy model as described in Section 4.4.4. However, apart from solving integer constraints, ProB does not improve the performance of constraint solving for Alloy models compared to the Alloy Analyzer as shown in Section 4.7 and Section 5.2. The different concept of modeling in Alloy compared to B renders constraint solving for translated models complex in ProB. Relational operators are heavily used in Alloy since n-ary finite relations are the only type of terms. The Alloy community seems to avoid the use of integers if possible since they are not well-supported by the Alloy Analyzer and, in particular, by SAT solvers. The experiments of translating Alloy 6 to B presented in Section 5.3 further emphasized this trend. An exemplary syntax-directed translation of an Alloy 6 model to B cannot be checked by ProB in a reasonable amount of time. Yet, this is also the case for the Alloy Analyzer's model checking backends. Here, we deem the suggested model to be not optimal for model checking since it defines many (symmetric) states. An idiomatic B model of the same puzzle that uses integers and a total function instead of many relational operators can be checked by ProB in a short amount of time. A corresponding model in Alloy 6 can still not be checked by the Alloy Analyzer's model checking backends while a syntax-directed translation to B can again be checked fast by ProB.

In summary, it is currently only possible to efficiently check a subset of Alloy models using ProB, but first experiments show benefits of using the ProB model checker for checking Alloy 6 models compared to the Alloy Analyzer's backends.

RQ3: What are significant strengths and weaknesses of SAT solving (Alloy Analyzer) compared to CLP (ProB)? One weakness of SAT solving involving background theories (aka eager SMT solving) is the integer arithmetic. A bit width has to be defined to translate integers to propositional logic and integer overflows might occur. In Section 4.8.1, we have shown that the integer arithmetic in Alloy is unsound when the overflow detection is turned off. Further, a translation to propositional logic is restricted to finite domains. CLP and ProB's constraint solver both support integers over unbounded domains and integer overflows do not occur.

For relational constraints over finite domains such as the transitive closure, a translation to propositional logic shows benefits compared to CLP. The Alloy Analyzer's backend to Kodkod is able to solve complex relational formulas in a short amount of time as shown in Section 5.2. Here, the concept of reducing and enumerating domains as applied in CLP can result in many choice points possibly inducing a lot of backtracking, especially when disproving formulas. ProB's constraint solver already implements a bit vector encoding of sets whose cardinality has to be computed, which was inspired by translations to propositional logic as applied by Kodkod. This encoding is applied lazily as soon as a set is known to be finite within the constraint solver. In the future, it should be investigated whether such eager SMT encodings can also lift ProB's performance for solving specific relational constraints. For instance, neither ProB's constraint solver nor its interface to Kodkod are able to prove the constraint $\text{closure}(\{1 \rightarrow 2, 2 \rightarrow 3\}) \notin 1..3 \leftrightarrow 1..3$.

Further, a manual combination of SAT solving and CLP in ProB resorting to Kodkod where both constraint solvers share constraints has shown benefits [156]. An automated integration of both constraint solvers seems promising to be investigated in the future. Our empirical evaluations have shown that it might be beneficial to automatically send all finite relational constraints to Kodkod, *e.g.*, a transitive closure, and let the remaining constraints be solved by ProB.

RQ4: Which main use cases can be singled out in which both languages complement each other? The benefits of SMT for solving Alloy models was already established prior to our work [213]. We further emphasized the need for sound integers in Alloy including other constraint solving backends than SAT solving (aka eager SMT solving), *e.g.*, ProB, and contributed to an ongoing discussion on natively integrating state changes in the Alloy language at that time. Afterward, different work on integrating SMT solvers in Alloy was presented [214, 232]. Further, the Alloy 6 language was presented, which natively provides a declarative concept of state changes as well as native BMC and complete model checking via external model checkers. In Section 5.3, we have shown that ProB's explicit-state and LTL model checkers can improve the performance of checking translated Alloy models compared to the Alloy Analyzer's backends. Yet, the style of modeling in Alloy often leads to non-idiomatic constraints in B causing a bad performance of model checking, which is also the case for the Alloy Analyzer's backends. Besides using ProB's model checker, we deem a native explicit-state model checker to be beneficial for verifying state-based Alloy 6 models.

The B language could be more flexible to improve its accessibility for novices, *e.g.*, by providing an operator similar to Alloy's relational join operator. In B, one has to remember many different syntactical symbols for the different types of functions. Alloy, on the other hand, uses quantities such as `some` or `one`, which can be more accessible in the beginning. However, in contrast to Alloy, the B language defines denotational semantics, which we believe prevents confusion in the long term. The Alloy Analyzer provides a vivid visualization of models called magic layout. Inspired by this, a similar layout for visualizing B and Event-B models was integrated in ProB's new graphical user interface based on the Java platform [233], which is especially useful for visualizing single states as is the case when using model finding. Yet, for visualizing state-based models, we deem an interactive animator as provided by ProB to be more useful, and suggest extending the Alloy Analyzer to integrate a similar animator in its graphical user interface.

9. SMT Solving for the Validation of B and Event-B Models

An integration of the Z3 SMT solver in ProB revealed benefits for findings contradictions in B formulas compared to ProB's constraint solver, especially over unbounded domains [48]. Yet, the translation uses many quantifiers for which Z3 often fails to find solutions. The following research questions thus arose, which could be answered in this thesis.

RQ5: How can the performance and coverage of ProB's integration of Z3 be improved? We found that Z3 supports lambda functions which can be used for many translations of B operators to SMT-LIB as understood by Z3 and presented formalized translation rules in Section 6.4. Different empirical evaluations in Section 6.7, Section 7.4, and Section 7.6 have shown that using lambda functions instead of quantified formulas improves the performance of Z3 for solving B and Event-B constraints, especially for model finding. We were also able to improve the coverage of the translation by supporting B's relational closure, iteration, and composition. These operators are not supported by the initial axiomatic translation to SMT-LIB since their translations using quantifiers are too involved. Unfortunately, Z3's support for lambda functions was unsound. This lead to Z3 disabling the use of lambda functions inside recursive definitions after a corresponding software bug was reported [186]. The translations presented in Section 6.4.1 are thus currently not supported by Z3 version 4.12.2 but remain valid in theory. Once Z3 provides full support for lambda functions, their implementation in ProB can be activated again.

The empirical results have further shown that the initial axiomatic translation from B to SMT-LIB also has benefits compared to the new constructive translation. We thus presented a new parallel integration of different configurations of Z3 in ProB using both translations to obtain the best performance. Further, we found that contradictions that are obvious in B might become complex to spot by Z3 after translating to SMT-LIB. For this, we decided to abstract B formulas to propositional logic as applied by lazy SMT solvers and check if a resulting formula is satisfiable using a small timeout (around 50 ms) as described in Section 6.7.1. If this is not the case, we have found a contradiction and do not have to translate a formula to SMT-LIB. Additionally, we found that a decomposition of constraints into independent components prior to the translation can improve the performance for finding contradictions, *e.g.*, as is the case for the benchmarks from inductive invariant proofs presented in Section 7.6.2. Yet, this depends on the order of components to be solved.

Last but not least, instantiating quantifiers that reason over finite domains prior to the translation to SMT-LIB can improve the performance of constraint solving for Z3 as shown in Section 7.5.

RQ6: Which steps are necessary to use ProB's constraint solver as a theory solver for SMT solving of B and Event-B constraints? In Section 6.5, we presented a direct implementation of SMT solving in ProB using sophisticated features such as early pruning, watched-literals, theory propagation, and restarts with phase saving. It is easy to use ProB's constraint solver as a theory solver with early pruning by using coroutines for constraint reification in Prolog. One pitfall is to send not well-defined predicates to ProB's constraint solver as is usually done in SMT solving. In low level Prolog code, the constraint solver does not report well-definedness errors. Thus, ProB's constraint solver neither confirms nor refutes the satisfiability of a formula if it is not well-defined. This can lead to spurious counterexamples being found by the SMT solver since possibly correct solutions of the SAT solver are refuted by the theory solver. We solved this issue by adding all necessary well-definedness conditions to each theory constraint that is reified with a Boolean variable as described in Section 6.5.4 and formalized in Section 7.3. Further, we introduce implications for each well-definedness condition in propositional logic to inform the SAT solver about B's well-definedness. This information allows for clause learning from well-definedness conditions and therewith improves the performance of SAT solving for Boolean abstractions of B and Event-B constraints.

Another peculiarity of ProB's constraint solver is that it implements theory propagation but does not provide explanations for derived formulas, which are required for SMT solving. We thus presented a method for lazily explaining theory propagations by computing an unsatisfiable core as described in Section 6.5.4.

Last but not least, an important aspect to consider when solving B and Event-B constraints are so-called unfixed deferred sets. There are different points in SMT solving where unfixed deferred sets can lead to spurious counterexamples being found whose occurrences need to be logged. We found that learning from such spurious counterexamples can lead to finding solutions in which case the use of unfixed deferred sets is irrelevant as described in Section 7.1. Yet, if refuting a formula after propagation one or more unfixed deferred sets, the SMT solver has to indicate that no decision could be made (unknown), which corresponds to the behavior of ProB's constraint solver.

RQ7: What are significant benefits of a direct implementation of SMT solving in ProB compared to using Z3? First and foremost, the direct implementation of SMT solving in ProB has the benefits of every proprietary software such as an increased maintainability and extensibility. Compared to the integration of Z3, we do not have to care for a high memory consumption, possible segmentation faults or assertion violations, which possibly lead to killing Z3's process.

Despite its benefits for solving many B constraints, the integration of Z3 fails to solve many constraints that are easy to solve for ProB. For instance, Z3 is not able to find a solution for $x \in \mathbb{P}(\mathbb{Z}) \wedge \mathrm{card}(x) > 10$. The main benefit of using ProB as a theory solver in SMT solving is that it is tailored towards solving B and Event-B constraints such as the set cardinality or relational composition. Further, the translation to SMT-LIB does not support the complete B language, which is the case for ProB's constraint and SMT solver. While the empirical results generally show that no constraint solver is the best for all types of constraints, ProB's SMT solver was able to solve several constraints better than ProB's constraint solver and its integration of Z3 as can be seen in Section 6.7.2 and Section 7.6. Further, a custom implementation of SMT solving enables the integration of an additional theory solver alongside ProB's constraint solver as shown in Section 6.6.

RQ8: What are significant strengths and weaknesses of CLP and SMT for solving B and Event-B constraints? SMT solving has proven to be especially useful for finding contradictions, *e.g.*, due to CDCL and backjumping. Further, using a Boolean abstraction of a FOL formula can prevent a possibly infeasible enumeration in the theory solver if a contradiction can already be detected by the SAT solver. CLP, on the other hand, is often better suited to find solutions for satisfiable constraints. For instance, we experienced issues in ProB's SMT solver where the SAT solver assigns variables in an order that is inconvenient for the theory solver resulting in exceeding the predefined timeout. Yet, when setting up all constraints at once, ProB's constraint solver is able to find a solution fast using CLP. In the future, this issue might be improved by using knowledge from the theory solver for the variable branching heuristic in the SAT solver. Weaknesses of CLP are the application of chronological backtracking and that it does not learn from conflicts.

In summary, it can be seen that strengths of CLP are weaknesses of SMT solving and vice versa. Both techniques thus appear to be a must-have for any portfolio of constraint solvers.

RQ9: How can the performance of ProB's constraint solver in disproving integer constraints over unbounded domains be improved? The empirical evaluations presented in this thesis have shown that ProB's constraint solver lacks performance for finding contradictions over unbounded integer domains. A reason is that CLP(FD) resorts to the plain enumeration of domains at some point, which is not feasible for unbounded domains. For instance, ProB is not able to disprove the formulas $x \in \mathbb{Z} \wedge x > y \wedge y > x$ as well as $x \in \mathbb{Z} \wedge y - x \leq 1 \wedge z - y \leq -3 \wedge x - z \leq 1$. We presented a graph-based constraint solver for the integer difference logic as well as an integration in ProB's SMT solver alongside ProB's constraint solver as described in Section 6.6. In Section 6.7.2, we have shown that the additional theory solver improves the performance of ProB's SMT solver. First, the IDL solver often enables faster proving and disproving of IDL constraints compared to CLP(FD), especially over unbounded integer domains, since it does not enumerate domains but computes shortest paths in a directed graph. Second, it automatically provides unsatisfiable cores without any additional overhead.

In the future, we can try to integrate the graph-based IDL solver into ProB's constraint solver for solving quantified formulas over unbounded integer domains. For instance, ProB is not able to prove the formula $\forall(x,y).(x \in \mathbb{Z} \land y \in \mathbb{Z} \Rightarrow \exists z.(x - z = y))$. Instead, the constraint solver recognizes that the domains cannot be enumerated exhaustively since they are unbounded and generates a virtual timeout (unknown). Here, it should be noted that the integration of the IDL solver in ProB's SMT solver does not affect solving quantified formulas since they are abstracted as a single Boolean variable. In particular, the theory solver thus receives complete quantified formulas as is the case for ProB's native constraint solver.

RQ10: What are significant features of Prolog that are specifically suitable or unsuitable for implementing an SMT solver? We have found Prolog to be a good, but not perfect, fit for implementing an SMT solver. First and foremost, Prolog is fast and able to perform millions of logical inferences per second due to the restriction to Horn clauses and the application of linear resolution for proof. Prolog natively implements backtracking and it is simple to extend its capabilities to backjumping for CDCL. For instance, one can use dynamic assertions that are checked for existence at each choice point deciding on whether a choice point is evaluated.

Modern Prolog implementations such as SICStus Prolog provide efficient data structures such as mutable dictionaries with amortized constant access on existing elements. Further, the language allows defining dynamic assertions and mutable state. These features are important for an efficient management of an SMT solver's state, which is often accessed and updated concurrently. Besides that, Prolog provides coroutines for a delayed propagation, which renders the implementation of features such as watched literals, early pruning, and theory propagation easy.

Additionally, Prolog allows for an intuitive and performant processing of abstract syntax trees due to the applied concepts of SLD-resolution and unification. For instance, Prolog implements definite clause grammars which allow defining concise and fast rules for rewriting propositional logic formulas to CNF.

Last but not least, we deem debugging to be easier in Prolog compared to low-level languages such as C. Prolog's debugger (`trace/0`) provides different features such as entering, skipping or redoing specific calls, and the debugger can be started at any time during the interpretation. Yet, one downside is that large abstract syntax trees in predicate calls often render them difficult to read.

Unfortunately, using Prolog also has some disadvantages for SMT solving. First, modifications of watched literals after propagating a SAT variable are undone when backtracking which is not necessary for SAT solving as was already pointed out by Howe and King [192]. In particular, all predicate calls are undone and checked for choice points during backtracking or backjumping. When using imperative programming languages, backjumping can be implemented faster since one just has to unassign all variables, clear the state, and change the current level of the search. Second, Prolog does not allow storing variable references globally. Therefore, an environment with variable references has to be used to set the references in learned clauses and theory conflicts during SMT solving. Searching for the corresponding variable references causes additional overhead. Since the amount of learned clauses during SAT and SMT solving can become large, this overhead is definitely notable. Last but not least, most low-level languages provide more sophisticated implementations for heaps than Prolog. For instance, it is not possible to efficiently update the priority of an element when using SICStus Prolog's default library for heaps. SAT solvers usually use a heap for managing their decision heuristic and often perform updates on existing elements. We therefore implemented an interface to the C++ Boost library in SICStus Prolog providing fibonaccy heaps. This C++ interface is backtrackable, *i.e.*, the effect of operations is undone when backtracking, by maintaining a counter for each element stating the time it was last updated.

All in all, we deem SAT solving to be more efficient in low-level languages such as C. Nevertheless, in the case of SMT solving with ProB's constraint solver, using a SAT solver implemented in C would require many interface calls between Prolog and C, which is not the case for a pure Prolog implementation.

10. Future Work

Besides the already presented future work, we want to mention two more possibilities to improve the performance of solving B and Event-B formulas in ProB.

Instead of using constraint solving, theorem proving might be beneficial for proving and disproving B and Event-B formulas. A reason is that theorem provers do not enumerate domains but use techniques such as the superposition calculus [234]. Here, a predefined set of rewriting rules is applied for simplifying formulas to finally disprove the initial goal, *i.e.*, receive a contradictory statement. For proving, the initial formula can be negated. For instance, this can potentially improve the performance for proving or disproving integer constraints and quantified formulas compared to ProB's constraint solver, especially over unbounded domains. Further, theorem provers do not have to take care of so-called unfixed deferred sets since their proofs are generally valid and do not have to make assumptions on the size of deferred sets. Yet, a disadvantage of theorem provers is that they do not provide explicit models, which is required for many tasks in ProB such as the animation of models or computing counterexamples for invalid properties. One possibility for integrating automated theorem provers in ProB is to translate constraints into the TPTP language [235] and use dedicated theorem provers.

In Section 6.7.2 and Section 7.6, it could be shown that CDCL(T) improves the performance for several constraints compared to plain saturation-based solving, especially for finding contradictions. Instead of learning constraints as is done in CDCL, one promising future work for improving the performance of constraint solving is to learn behavior in ProB's constraint solver. For instance, Zombori et al. [236] implemented reinforcement learning for guiding an automated theorem prover by learning from historic data. Chalumeau et al. [237] presented a constraint solver implementing reinforcement learning to learn branching decisions for certain types of constraints. ProB's constraint solver allows for a randomized enumeration of domains, which can improve the performance of constraint solving. By default, domains are enumerated linearly. Yet, there is no general domain enumeration order that is best for all types of constraints. It could thus be useful to learn the enumeration order of domains for certain constellations of constraints in ProB's constraint solver, especially in its CLP(FD) backend. Besides that, the application of certain rewriting or deduction rules in the constraint solver can be learned. By using a randomized enumeration order, it is possible to generate many simulations to implement search strategies such as a Monte-Carlo tree search.

ProB's SMT solver would also benefit from such an improvement since ProB's constraint solver is used as a theory solver. The SMT solver itself can possibly be further improved by learning clause selections in the SAT solver. For instance, Jakubuv and Urban have shown that learning clause guidance can improve the performance of constraint solving drastically [238].

Appendix

A. Alternative Alloy 6 Model of the Chameleon Puzzle

Listing A.1: Alloy 6 model of the „Chameleon Puzzle“ [7].

```
enum colors {blue,green,yellow}

one sig population {
  var chameleons: colors -> one Int
}

fact init {
  (population.chameleons)[blue] = 13 and
  (population.chameleons)[green] = 15 and
  (population.chameleons)[yellow] = 17
}

pred meet[c1: colors, c2: colors] {
  c1 != c2 and (population.chameleons)[c1] > 0 and (population.
      chameleons)[c2] > 0 and
  (let c3 = ((colors - c1) - c2) |
          population.chameleons' =
             (((population.chameleons
                ++ (c1->(minus[(population.chameleons)[c1],1])))
                ++ (c2->(minus[(population.chameleons)[c2],1])))
                ++ (c3->(plus[(population.chameleons)[c3],2]))
           ))
}

pred skip {
  population.chameleons' = population.chameleons
}

fact step {
  always ((some c1,c2: colors | meet[c1,c2]) or skip)
}

pred invariant {
  eventually ((population.chameleons)[blue] = 0 and (population.
      chameleons)[green] = 0)
}

run invariant for 7 Int, 361 steps
```

Listing A.2: A manual translation of the Alloy 6 model presented in Listing A.1 to classical B.

```
MACHINE chameleon_alloy_to_b

SETS Color = {yellow,green,blue}; Population = {P}

PROPERTIES
  card(Color) = 3 & card(Population) = 1 &
  {yellow}/\{green} = {} & {yellow}/\{blue} = {} &
  {green}/\{blue} = {}

VARIABLES
  chameleons_P

DEFINITIONS
  "CHOOSE.def";
  ASSERT_LTL  == "F{chameleons_P(blue) = 0 & chameleons_P(green) =
     0}"

INVARIANT
  chameleons_P : Color --> INTEGER &
  not (chameleons_P(blue) = 0 & chameleons_P(green) = 0)

INITIALISATION
  chameleons_P := {blue |-> 13, green |-> 15, yellow |-> 17}

OPERATIONS
  meet(c1,c2) =
    PRE c1 : Color & c2 : Color & c1 /= c2 & chameleons_P(c1) > 0 &
         chameleons_P(c2) > 0 THEN
         chameleons_P := chameleons_P <+
           (LET c3 BE  c3 = (Color - {c1,c2})
             IN {c1 |-> (chameleons_P(c1) - 1),
                   c2 |-> (chameleons_P(c2) - 1),
                   MU(c3) |-> chameleons_P(MU(c3)) + 2}
             END)
    END
END
```

B. Transformation Rules for Integer Difference Logic

In Figure B.1, we present the syntax-directed rules for rewriting a subset of integer constraints into IDL as implemented in ProB's graph-based constraint solver for IDL. The translation is represented by the function Ψ. Atomic IDL constraints are of the form $v_i - v_j \leq c$, where v_i and v_j are integer variables and c is a constant integer value. zero is the artifical variable introduced to rewrite formulas without a subtrahend to IDL as described in Section 6.6.1.

$\Psi(c \leq x - y) \equiv y - x \leq -c$

$\Psi(x - y \leq c) \equiv x - y \leq c$

$\Psi(x \leq c) \equiv x - \text{zero} \leq c$

$\Psi(-x \leq c) \equiv \text{zero} - x \leq c$

$\Psi(x \leq y) \equiv x - y \leq 0$

$\Psi(-x \leq y) \equiv y - x \leq 0$

$\Psi(c \leq y) \equiv \text{zero} - y \leq -c$

$\Psi(c \leq -y) \equiv y - \text{zero} \leq -c$

$\Psi(c1 \leq y - c2) \equiv \text{zero} - y \leq -c2 - c1$

$\Psi(c1 \leq -y - c2) \equiv y - \text{zero} \leq -c2 - c1$

$\Psi(x \leq y - c) \equiv x - y \leq -c$

$\Psi(x \leq y + c) \equiv x - y \leq c$

$\Psi(x - c \leq y) \equiv x - y \leq c$

$\Psi(x + c \leq y) \equiv x - y \leq -c$

$\Psi(x - c1 \leq c2) \equiv x - \text{zero} \leq c2 + c1$

$\Psi(x + c1 \leq c2) \equiv x - \text{zero} \leq c2 - c1$

$\Psi(x + c1 \leq y - c2) \equiv x - y \leq -c2 - c1$

$\Psi(x + c1 \leq y + c2) \equiv x - y \leq c2 - c1$

$\Psi(x - c1 \leq y - c2) \equiv x - y \leq -c2 + c1$

$\Psi(x - c1 \leq y + c2) \equiv x - y \leq c2 + c1$

$\Psi(c < x - y) \equiv y - x \leq -c - 1$

$\Psi(x - y < c) \equiv x - y \leq c - 1$

$\Psi(x < c) \equiv x - \text{zero} \leq c - 1$

$\Psi(-x < c) \equiv \text{zero} - x \leq c - 1$

$\Psi(x < y) \equiv x - y \leq -1$

$\Psi(-x < y) \equiv y - x \leq -1$

$\Psi(c < y) \equiv \text{zero} - y \leq -c - 1$

$\Psi(c < -y) \equiv y - \text{zero} \leq -c - 1$

$\Psi(c1 < y - c2) \equiv \text{zero} - y \leq -c2 - c1 - 1$

$\Psi(c1 \leq -y - c2) \equiv y - \text{zero} \leq -c2 - c1 - 1$

$\Psi(x < y - c) \equiv x - y \leq -c - 1$

$\Psi(x < y + c) \equiv x - y \leq c - 1$

$\Psi(x - c < y) \equiv x - y \leq c - 1$

$\Psi(x + c < y) \equiv x - y \leq -c - 1$

$\Psi(x - c1 < c2) \equiv x - \text{zero} \leq c2 + c1 - 1$

$\Psi(x + c1 < c2) \equiv x - \text{zero} \leq c2 - c1 - 1$

$\Psi(x + c1 < y - c2) \equiv x - y \leq -c2 - c1 - 1$

$\Psi(x + c1 < y + c2) \equiv x - y \leq c2 - c1 - 1$

$\Psi(x - c1 < y - c2) \equiv x - y \leq -c2 + c1 - 1$

$\Psi(x - c1 < y + c2) \equiv x - y \leq c2 + c1 - 1$

$\Psi(x - y = c) \equiv x - y \leq c \wedge y - x \leq -c$

$\Psi(x = y - c) \equiv \Psi(x - y = -c)$

$\Psi(x = y + c) \equiv \Psi(x - y = c)$

$\Psi(x - c1 = y - c2) \equiv \Psi(x - y = -c2 + c1)$

$\Psi(x - c1 = y + c2) \equiv \Psi(x - y = c1 + c2)$

$\Psi(x + c1 = y - c2) \equiv \Psi(x - y = -c2 - c1)$

$\Psi(x + c1 = y + c2) \equiv \Psi(x - y = c1 - c2)$

$\Psi(x = y) \equiv \Psi(x - y = 0)$

$\Psi(-x = -y) \equiv \Psi(y - x = 0)$

$\Psi(x - c = y) \equiv \Psi(x - y = c)$

$\Psi(x + c = y) \equiv \Psi(x - y = -c)$

$\Psi(c1 = y - c2) \equiv \Psi(y = c1 + c2)$

$\Psi(c1 = y + c2) \equiv \Psi(y = c1 - c2)$

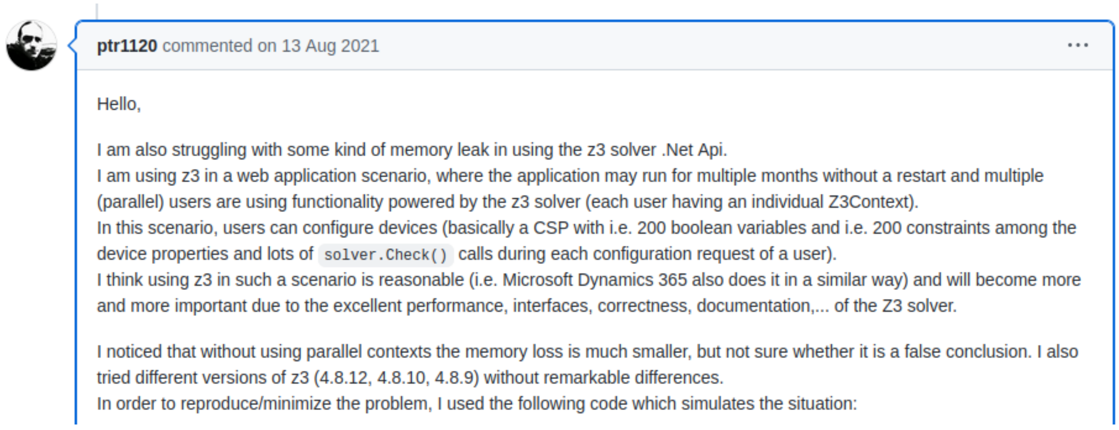
$\Psi(x - c1 = c2) \equiv \Psi(x = c2 + c1)$

$\Psi(x + c1 = c2) \equiv \Psi(x = c2 - c1)$

$\Psi(x = c) \equiv \Psi(x - \text{zero} = c)$

$\Psi(-x = c) \equiv \Psi(\text{zero} - x = c)$

Figure B.1.: Syntax-directed rules for rewriting a subset of integer constraints into IDL as implemented in ProB's graph-based constraint solver for IDL.



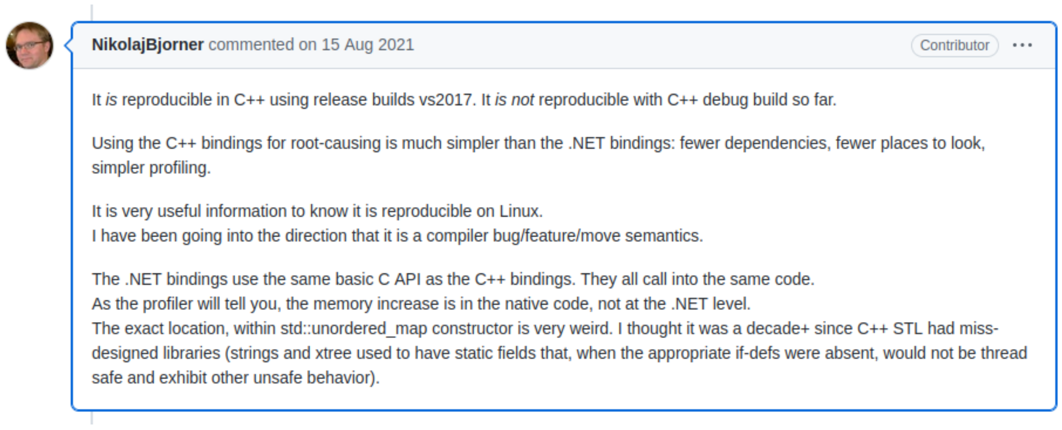
ptr1120 commented on 13 Aug 2021

Hello,

I am also struggling with some kind of memory leak in using the z3 solver .Net Api.
I am using z3 in a web application scenario, where the application may run for multiple months without a restart and multiple (parallel) users are using functionality powered by the z3 solver (each user having an individual Z3Context).
In this scenario, users can configure devices (basically a CSP with i.e. 200 boolean variables and i.e. 200 constraints among the device properties and lots of `solver.Check()` calls during each configuration request of a user).
I think using z3 in such a scenario is reasonable (i.e. Microsoft Dynamics 365 also does it in a similar way) and will become more and more important due to the excellent performance, interfaces, correctness, documentation,... of the Z3 solver.

I noticed that without using parallel contexts the memory loss is much smaller, but not sure whether it is a false conclusion. I also tried different versions of z3 (4.8.12, 4.8.10, 4.8.9) without remarkable differences.
In order to reproduce/minimize the problem, I used the following code which simulates the situation:

Figure C.1.: Excerpt from a comment on Github about a memory leak in Z3, especially when using many Z3 solvers in parallel [5].



NikolajBjorner commented on 15 Aug 2021 Contributor

It *is* reproducible in C++ using release builds vs2017. It *is not* reproducible with C++ debug build so far.

Using the C++ bindings for root-causing is much simpler than the .NET bindings: fewer dependencies, fewer places to look, simpler profiling.

It is very useful information to know it is reproducible on Linux.
I have been going into the direction that it is a compiler bug/feature/move semantics.

The .NET bindings use the same basic C API as the C++ bindings. They all call into the same code.
As the profiler will tell you, the memory increase is in the native code, not at the .NET level.
The exact location, within std::unordered_map constructor is very weird. I thought it was a decade+ since C++ STL had miss-designed libraries (strings and xtree used to have static fields that, when the appropriate if-defs were absent, would not be thread safe and exhibit other unsafe behavior).

Figure C.2.: An answer of Nikolaj Bjørner, one of Z3's core developers, on the issue regarding the memory leak in Z3 as can be seen in Figure C.1 [6].

C. Discussion on Github about Z3's Memory Consumption

A user of Z3 started a discussion on Github regarding a memory leak when disposing a Z3 solver context [224]. Another user further emphasized this issue when using many Z3 solvers in parallel as shown in Figure C.1. We sometimes experience the same issue and think that the description matches our use case. The answer of one of Z3's core developers shown in Figure C.2 is also of interest for our application.

Table D.1.: First part of detailed results of the BMC benchmarks used in Section 7.6.

		ProB-Z3								ProB			
No.	Name	(axiomatic)	(constructive)	(parallel)	(parallel & decomposed)	ProB	ProB-Sym	ProB-CSE	ProB-CHR	SMT	Raw-SMT	Sym-Raw-SMT	Sym-SMT
1	B/ABCD/Broker/TransactionsSimple	2 / 132 s	2 / 80 s	2 / 132 s	1 / 246 s	1 / 241 s	1 / 243 s	1 / 241 s	1 / 241 s	3 / 10 s	1 / 248 s	1 / 249 s	3 / 10 s
2	B/ABCD/TheSystem_small	2 / 144 s	2 / 12 s	2 / 146 s	2 / 131 s	1 / 241 s	2 / 177 s	2 / 162 s	1 / 241 s	1 / 244 s	1 / 244 s	1 / 244 s	1 / 244 s
3	B/ABCD/USER_CLASS	0 / 3 s	0 / 201 s	0 / 98 s	0 / 243 s	1 / 121 s	1 / 122 s	0 / 241 s	1 / 121 s	0 / 242 s	1 / 144 s	0 / 244 s	0 / 243 s
4	B/ABCD/bookstore	0 / 2 s	0 / 2 s	0 / 2 s	0 / 2 s	0 / 125 s	0 / 128 s	0 / 127 s	0 / 126 s	0 / 243 s	0 / 243 s	0 / 244 s	0 / 243 s
5	B/BZTT/GSM_revue	0 / 6 s	0 / 4 s	0 / 10 s	0 / 244 s	1 / 241 s	1 / 247 s	1 / 241 s	1 / 241 s	1 / 279 s	1 / 282 s	1 / 290 s	1 / 279 s
6	B/B_Day/Parking_R2	3 / 1 s	3 / 1 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	2 / 122 s	2 / 123 s	3 / 2 s
7	B/Benchmarks/Cruise_finite_k	3 / 5 s	3 / 4 s	3 / 5 s	3 / 9 s	1 / 241 s	1 / 251 s	1 / 242 s	1 / 241 s	1 / 289 s	1 / 291 s	1 / 301 s	1 / 291 s
8	B/Benchmarks/DSP0	2 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 2 s	3 / 1 s
9	B/Benchmarks/phonebook7_err	1 / 85 s	0 / 31 s	1 / 64 s	1 / 64 s	1 / 135 s	1 / 126 s	1 / 130 s	1 / 130 s	1 / 125 s	1 / 124 s	1 / 125 s	1 / 125 s
10	B/Benchmarks/tictac	1 / 10 s	3 / 40 s	3 / 44 s	3 / 10 s	2 / 130 s	2 / 124 s	2 / 130 s	2 / 137 s	2 / 102 s	2 / 121 s	2 / 123 s	2 / 101 s
11	B/CBC/Enabling/OpCallSelect	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 32 s	3 / 33 s	3 / 1 s
12	B/Demo/Bakery1err2	3 / 1 s	3 / 1 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	2 / 125 s	1 / 242 s	1 / 242 s	2 / 126 s
13	B/Demo/RussianPostalPuzzle2	1 / 242 s	3 / 8 s	3 / 10 s	3 / 10 s	1 / 241 s	1 / 242 s	1 / 241 s	1 / 241 s	1 / 244 s	1 / 250 s	1 / 249 s	1 / 244 s
14	B/Demo/Simpson/Simpson_Four_Slot	1 / 215 s	3 / 7 s	3 / 10 s	1 / 29 s	1 / 241 s	1 / 244 s	1 / 241 s	1 / 241 s	1 / 248 s	1 / 248 s	1 / 247 s	1 / 248 s
15	B/Demo/Simpson/Simpson_Four_Slot_CSP	0 / 244 s	1 / 13 s	1 / 244 s	0 / 59 s	1 / 241 s	1 / 243 s	1 / 241 s	1 / 241 s	1 / 244 s	1 / 245 s	1 / 246 s	2 / 187 s
16	B/Demo/Simpson/Simpson_Four_Slot_Ordered	1 / 243 s	2 / 12 s	2 / 124 s	1 / 38 s	2 / 121 s	2 / 122 s	2 / 121 s	2 / 121 s	1 / 246 s	1 / 248 s	1 / 249 s	1 / 247 s
17	B/Demo/Simpson/Simpson_Four_Slot_Symm	0 / 368 s	0 / 29 s	0 / 247 s	0 / 246 s	0 / 255 s	0 / 262 s	0 / 256 s	0 / 252 s	0 / 280 s	0 / 313 s	0 / 298 s	0 / 281 s
18	B/Demo/Simpson/Simpson_Four_Slot_TLC	1 / 186 s	3 / 5 s	3 / 7 s	1 / 19 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 241 s	2 / 240 s	1 / 243 s	1 / 243 s	1 / 243 s
19	B/Demo/phonebook5	2 / 2 s	0 / 1 s	2 / 2 s	1 / 3 s	2 / 82 s	2 / 57 s	2 / 75 s	2 / 78 s	1 / 122 s	1 / 122 s	1 / 123 s	1 / 123 s
20	B/Demo/scheduler_err	0 / 3 s	1 / 2 s	1 / 3 s	2 / 3 s	2 / 9 s	2 / 8 s	2 / 9 s	2 / 10 s	1 / 132 s	0 / 245 s	2 / 59 s	2 / 132 s
21	B/ErrorMachines/OneInvariantViolation	3 / 1 s	3 / 1 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 3 s	3 / 3 s	3 / 1 s
22	B/EventB/Bosch_mini1_m_3	3 / 2 s	3 / 2 s	3 / 2 s	3 / 3 s	1 / 241 s	1 / 242 s	0 / 361 s	1 / 241 s	1 / 273 s	1 / 271 s	1 / 272 s	1 / 273 s
23	B/EventB/Bosch_mini1v2_m_3	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	1 / 241 s	1 / 241 s	0 / 361 s	1 / 241 s	1 / 272 s	1 / 272 s	1 / 273 s	1 / 273 s
24	B/EventB/Bosch_mini2_m_3	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	0 / 361 s	0 / 361 s	0 / 361 s	0 / 361 s	1 / 271 s	1 / 271 s	1 / 272 s	1 / 271 s
25	B/EventB/ETH_Elevator/elevator10	0 / 266 s	3 / 39 s	3 / 48 s	0 / 268 s	0 / 361 s	0 / 369 s	0 / 363 s	0 / 361 s	0 / 364 s	0 / 363 s	0 / 364 s	0 / 364 s
26	B/EventB/ETH_Elevator/elevator5	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 123 s	2 / 123 s	2 / 121 s
27	B/EventB/ETH_Elevator/elevator6	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 122 s	2 / 124 s	2 / 125 s	2 / 122 s
28	B/EventB/ETH_Elevator/elevator7	3 / 2 s	3 / 2 s	3 / 2 s	3 / 3 s	3 / 9 s	3 / 12 s	3 / 9 s	3 / 36 s	2 / 126 s	2 / 135 s	2 / 138 s	2 / 127 s
29	B/EventB/ETH_Elevator/elevator8	3 / 2 s	3 / 2 s	3 / 2 s	3 / 3 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 129 s	2 / 140 s	2 / 140 s	2 / 128 s
30	B/EventB/EventB_Projekt/lift_solution	0 / 246 s	2 / 37 s	2 / 126 s	0 / 248 s	1 / 243 s	1 / 247 s	1 / 244 s	1 / 243 s	1 / 245 s	1 / 245 s	1 / 247 s	1 / 245 s
31	B/EventB/SiemensMiniPilot_Abrial_mch_0	3 / 2 s	3 / 2 s	3 / 2 s	3 / 5 s	2 / 122 s	2 / 123 s	2 / 122 s	2 / 121 s	3 / 3 s	3 / 3 s	3 / 4 s	3 / 3 s
32	B/Ivo/BenchmarksEnablingAnalysis/Abrial_LandingGear3/m0_mch	3 / 3 s	3 / 3 s	3 / 4 s	3 / 3 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
33	B/Ivo/BenchmarksEnablingAnalysis/Abrial_LandingGear3/m1_mch	3 / 8 s	3 / 8 s	3 / 7 s	3 / 7 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
34	B/Ivo/BenchmarksEnablingAnalysis/Abrial_LandingGear3/m4_mch	3 / 67 s	3 / 57 s	3 / 59 s	3 / 54 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 2 s	3 / 1 s
35	B/Ivo/BenchmarksPGE/Pathological/AllEnabled_Worst_Case_mch	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
36	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v1_mch	3 / 2 s	3 / 2 s	3 / 2 s	3 / 4 s	3 / 97 s	2 / 123 s	3 / 107 s	2 / 121 s	3 / 52 s	3 / 85 s	3 / 70 s	3 / 59 s
37	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v2_mch	3 / 2 s	3 / 2 s	3 / 2 s	3 / 4 s	2 / 121 s	2 / 123 s	2 / 121 s	2 / 121 s	2 / 170 s	2 / 148 s	2 / 158 s	2 / 156 s
38	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v3_mch	3 / 2 s	3 / 2 s	3 / 2 s	3 / 5 s	3 / 18 s	1 / 242 s	1 / 241 s	3 / 62 s	3 / 14 s	3 / 31 s	3 / 43 s	3 / 13 s
39	B/Ivo/BenchmarksPOR/Concurrent_Program_Development/conc_1_mch_finite	1 / 5 s	0 / 3 s	1 / 9 s	1 / 18 s	3 / 1 s	3 / 3 s	3 / 1 s	3 / 1 s	1 / 242 s	1 / 248 s	1 / 285 s	1 / 244 s
40	B/Ivo/BenchmarksPOR/Concurrent_Program_Development/conc_2_mch_finite	0 / 249 s	0 / 199 s	0 / 252 s	1 / 256 s	3 / 1 s	3 / 11 s	3 / 2 s	3 / 1 s	2 / 169 s	1 / 302 s	0 / 363 s	2 / 180 s
41	B/Ivo/BenchmarksPOR/Concurrent_Program_Development/conc_3_mch_finite	0 / 169 s	0 / 142 s	0 / 185 s	0 / 169 s	3 / 1 s	3 / 11 s	2 / 122 s	3 / 1 s	0 / 362 s	0 / 361 s	0 / 362 s	0 / 362 s
42	B/Ivo/BenchmarksPOR/Sieve/sieve_parallel_mch	0 / 133 s	1 / 13 s	1 / 11 s	1 / 16 s	2 / 122 s	2 / 125 s	2 / 124 s	2 / 123 s	1 / 245 s	1 / 245 s	1 / 247 s	1 / 245 s
43	B/Ivo/BenchmarksPOR/other/ConcurrentCounters	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 241 s	3 / 17 s	3 / 19 s	3 / 21 s	3 / 17 s
44	B/Ivo/NoDisablings_mx5	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	2 / 123 s	2 / 124 s	2 / 122 s	2 / 125 s	3 / 2 s	3 / 19 s	3 / 7 s	3 / 3 s
45	B/PerformanceTests/Generated/GeneratedMod100	3 / 125 s	3 / 143 s	3 / 134 s	2 / 147 s	1 / 247 s	1 / 291 s	1 / 281 s	1 / 247 s	3 / 15 s	3 / 15 s	3 / 15 s	3 / 15 s
46	B/PerformanceTests/PartialFunInverse	0 / 8 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 109 s	3 / 115 s	3 / 92 s	3 / 76 s
47	B/PragmasUnits/CaseStudies/Abrial_Hybrid/hybrid_flight/f_m0	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	2 / 42 s	3 / 2 s	3 / 2 s	2 / 42 s
48	B/PragmasUnits/CaseStudies/Abrial_Hybrid/hybrid_nuclear/C_m0	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	1 / 242 s	1 / 242 s	1 / 242 s	2 / 124 s
49	B/PragmasUnits/CaseStudies/Abrial_Hybrid/hybrid_nuclear/C_m0_internal	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	1 / 242 s	1 / 242 s	1 / 242 s	2 / 124 s
50	B/PragmasUnits/CaseStudies/Abrial_Hybrid/hybrid_nuclear/C_m0_internal_saved	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	1 / 242 s	1 / 242 s	1 / 242 s	2 / 124 s
51	B/PragmasUnits/InternalRepresentationTests/Case	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 22 s	3 / 5 s	3 / 5 s	3 / 21 s
52	B/PragmasUnits/InternalRepresentationTests/Case_internal	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 20 s	3 / 4 s	3 / 5 s	3 / 21 s
53	B/PragmasUnits/InternalRepresentationTests/Case_internal_saved	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 21 s	3 / 5 s	3 / 5 s	3 / 21 s
54	B/PragmasUnits/InternalRepresentationTests/Case_internal_saved_with_case	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 21 s	3 / 4 s	3 / 5 s	3 / 21 s
55	B/Puzzles/PartialSets	1 / 1 s	1 / 1 s	1 / 2 s	1 / 2 s	2 / 160 s	2 / 131 s	3 / 3 s	2 / 155 s	2 / 123 s	2 / 123 s	2 / 123 s	2 / 123 s
56	B/SchneiderBook/Chapter16/Mult	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
57	B/SchneiderBook/LabMaterial/records	2 / 2 s	2 / 3 s	2 / 3 s	2 / 3 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 14 s	2 / 8 s	2 / 9 s	2 / 11 s
58	B/Special/Dependency/SleepSetAlgorithm/SleepSets	1 / 244 s	1 / 243 s	1 / 244 s	1 / 246 s	1 / 246 s	1 / 248 s	1 / 246 s	1 / 246 s	1 / 242 s	1 / 243 s	1 / 244 s	1 / 243 s
59	B/Special/PO_ModelChecking/APBMR4	0 / 123 s	0 / 3 s	0 / 4 s	0 / 6 s	0 / 142 s	0 / 140 s	0 / 132 s	0 / 142 s	0 / 145 s	0 / 175 s	0 / 191 s	0 / 150 s
60	B/Special/PO_ModelChecking/Scheduler_Rodin_Provers	1 / 3 s	1 / 3 s	1 / 4 s	1 / 5 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 242 s	1 / 241 s	1 / 241 s
61	B/Special/PO_ModelChecking/Simple_PO_Check	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 4 s
62	B/Special/PO_ModelChecking/earley_3_original	0 / 377 s	0 / 9 s	0 / 387 s	0 / 378 s	3 / 1 s	3 / 2 s	3 / 2 s	3 / 1 s	2 / 186 s	3 / 157 s	2 / 145 s	2 / 192 s
63	B/SymbolicModelChecking/DisjunctionInProperties	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
64	B/SymbolicModelChecking/LightbotAbstract	3 / 42 s	3 / 29 s	3 / 30 s	3 / 41 s	2 / 125 s	2 / 122 s	2 / 127 s	2 / 130 s	1 / 257 s	1 / 257 s	1 / 278 s	1 / 260 s
65	B/SymbolicModelChecking/TimingExampleSimpler_v2_VariablesEvenMoreLimited	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	1 / 242 s	2 / 123 s	2 / 123 s	1 / 243 s
66	B/SymbolicModelChecking/TimingExampleSimpler_v2_VariablesLimited	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 28 s	1 / 243 s	1 / 245 s	1 / 244 s	1 / 243 s
67	B/SymmetryReduction/USB_4Endpoints	0 / 246 s	0 / 183 s	0 / 243 s	0 / 207 s	0 / 246 s	0 / 245 s	0 / 241 s	0 / 241 s	0 / 242 s	0 / 242 s	0 / 242 s	0 / 256 s
68	B/Tester/Any	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	2 / 127 s	2 / 126 s	2 / 125 s	2 / 127 s
69	B/Tickets/110/Library_mch	0 / 12 s	0 / 11 s	0 / 10 s	0 / 259 s	0 / 241 s	0 / 259 s	0 / 245 s	0 / 241 s	0 / 248 s	0 / 247 s	0 / 254 s	0 / 254 s
70	B/Tickets/Guiziou_ClearSy/Machine_AvecPrintf	0 / 1 s	0 / 1 s	0 / 2 s	0 / 5 s	2 / 126 s	2 / 122 s	2 / 125 s	2 / 126 s	2 / 122 s	2 / 122 s	2 / 123 s	2 / 122 s
							Solved constraints / Runtime s						

D. Detailed Results of Additional Constraint Solving Benchmarks

The detailed statistics of the additional benchmarks from BMC, deadlock freedom proofs, and inductive invariant checking used in Section 7.6 can be seen in Table D.1, Table D.2, Table D.3, and Table D.4. The paths stated in the benchmark names are relative to the folder `public_examples` in ProB's public specification repository [229].

Table D.2.: Second part of the BMC benchmarks presented in Table D.1. In total, 393 constraints were solved.

		ProB-Z3								ProB			
No.	Name	(axiomatic)	(constructive)	(parallel)	(parallel & decomposed)	ProB	ProB-Sym	ProB-CSE	ProB-CHR	SMT	Raw-SMT	Sym-Raw-SMT	Sym-SMT
71	B/Tickets/Guiziou_ClearSy/Machine_SansPrintf	2 / 121 s	1 / 121 s	2 / 157 s	2 / 123 s	2 / 122 s	2 / 122 s	2 / 122 s	2 / 122 s	2 / 151 s	2 / 152 s	2 / 152 s	2 / 151 s
72	B/Tickets/Hansen20_WhileInc/Counter	3 / 94 s	3 / 71 s	3 / 119 s	1 / 314 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
73	B/Tickets/Hansen20_WhileInc/Counter500	3 / 1 s	3 / 1 s	3 / 2 s	3 / 9 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
74	B/Tickets/Hansen20_WhileInc/Counter5000	3 / 14 s	3 / 15 s	3 / 17 s	3 / 45 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
75	B/Tickets/Treharne1/House_Tracker	0 / 99 s	0 / 33 s	0 / 63 s	0 / 86 s	0 / 138 s	0 / 146 s	0 / 141 s	0 / 143 s	0 / 188 s	0 / 188 s	0 / 148 s	0 / 188 s
76	CSPB/Bank3D	0 / 278 s	0 / 236 s	0 / 280 s	0 / 229 s	0 / 245 s	0 / 257 s	0 / 245 s	0 / 243 s	0 / 41 s	0 / 150 s	0 / 161 s	0 / 165 s
77	EventBPrologPackages/AbrialAccessControl/mac2_mch	0 / 243 s	0 / 5 s	0 / 244 s	0 / 294 s	3 / 10 s	3 / 6 s	3 / 1 s	3 / 9 s	1 / 249 s	1 / 258 s	1 / 262 s	1 / 250 s
78	EventBPrologPackages/AbrialAccessControl/mac4_mch	0 / 245 s	0 / 65 s	0 / 147 s	0 / 133 s	2 / 1 s	2 / 4 s	2 / 5 s	2 / 1 s	0 / 268 s	0 / 263 s	0 / 261 s	0 / 257 s
79	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m1_mch	3 / 2 s	3 / 2 s	3 / 2 s	3 / 3 s	1 / 241 s	1 / 242 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 242 s	1 / 241 s
80	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m2_mch	3 / 2 s	3 / 2 s	3 / 3 s	3 / 6 s	1 / 249 s	1 / 244 s	1 / 251 s	1 / 251 s	1 / 246 s	1 / 244 s	1 / 246 s	1 / 245 s
81	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m3_mch	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	1 / 241 s	1 / 243 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 242 s	1 / 241 s
82	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m4_mch	3 / 2 s	3 / 2 s	3 / 2 s	3 / 4 s	0 / 361 s	0 / 364 s	0 / 361 s	0 / 361 s	1 / 271 s	1 / 271 s	1 / 272 s	1 / 272 s
83	EventBPrologPackages/Abrial_BRP/DLK_Checking/ch6_brp_F01/b_4_mch	0 / 3 s	0 / 4 s	0 / 4 s	0 / 6 s	0 / 247 s	0 / 244 s	0 / 247 s	0 / 247 s	0 / 178 s	0 / 249 s	0 / 178 s	0 / 249 s
84	EventBPrologPackages/Abrial_BRP/DLK_Checking/ch6_brp_OK/b_5_mch	0 / 8 s	0 / 7 s	0 / 8 s	0 / 12 s	0 / 250 s	0 / 245 s	0 / 252 s	0 / 251 s	0 / 238 s	0 / 253 s	0 / 261 s	0 / 253 s
85	EventBPrologPackages/Abrial_Modes/mode_m3_mch	3 / 1 s	3 / 0 s	3 / 1 s	3 / 0 s	3 / 0 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 0 s	3 / 0 s	3 / 1 s	3 / 0 s
86	EventBPrologPackages/Abrial_Modes/mode_m4_mch	3 / 0 s	3 / 0 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 0 s
87	EventBPrologPackages/Abrial_Teaching/ch2_car/m2_lights_mch	3 / 0 s	3 / 0 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 0 s	3 / 1 s	3 / 1 s	3 / 0 s
88	EventBPrologPackages/Abrial_Teaching/ch2_car/m2_mch	3 / 2 s	3 / 2 s	3 / 2 s	3 / 4 s	2 / 121 s	2 / 123 s	2 / 121 s	2 / 123 s	1 / 206 s	2 / 72 s	2 / 74 s	1 / 206 s
89	EventBPrologPackages/Abrial_Teaching/ch2_car/m3_mch	3 / 3 s	3 / 3 s	3 / 3 s	3 / 10 s	3 / 9 s	3 / 11 s	3 / 10 s	2 / 122 s	1 / 139 s	1 / 247 s	1 / 248 s	1 / 143 s
90	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_2_standalone_mch	3 / 4 s	3 / 3 s	3 / 4 s	3 / 6 s	1 / 242 s	2 / 123 s	2 / 168 s	1 / 242 s	3 / 39 s	3 / 79 s	3 / 125 s	3 / 72 s
91	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_3_mch	1 / 10 s	1 / 12 s	1 / 11 s	1 / 19 s	0 / 124 s	0 / 129 s	0 / 124 s	0 / 128 s	1 / 149 s	1 / 158 s	1 / 163 s	1 / 149 s
92	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_3_standalone_mch	3 / 11 s	3 / 10 s	3 / 10 s	3 / 18 s	1 / 241 s	2 / 128 s	1 / 242 s	1 / 243 s	3 / 142 s	1 / 249 s	2 / 184 s	3 / 139 s
93	EventBPrologPackages/Abrial_Train_Ch17/train_0_mch	1 / 243 s	1 / 128 s	1 / 244 s	1 / 243 s	0 / 241 s	0 / 245 s	0 / 241 s	0 / 241 s	1 / 245 s	1 / 245 s	1 / 246 s	1 / 245 s
94	EventBPrologPackages/Abrial_Train_Ch17/train_1_prob_POR_mch	1 / 289 s	1 / 251 s	1 / 292 s	1 / 255 s	1 / 241 s	1 / 247 s	1 / 241 s	1 / 241 s	1 / 243 s	1 / 242 s	1 / 244 s	1 / 242 s
95	EventBPrologPackages/Abrial_Train_Ch17/train_1_prob_mch	1 / 289 s	1 / 250 s	1 / 296 s	1 / 254 s	1 / 241 s	1 / 248 s	1 / 241 s	1 / 241 s	1 / 244 s	1 / 257 s	1 / 263 s	1 / 247 s
96	EventBPrologPackages/Advance/CAN_Bus/CB3FSMM_mch_v2_wo_finite_inv	1 / 36 s	3 / 8 s	3 / 9 s	3 / 15 s	3 / 3 s	3 / 6 s	3 / 3 s	3 / 8 s	2 / 140 s	2 / 154 s	2 / 155 s	2 / 139 s
97	EventBPrologPackages/BinarySearch/binarySearchFail_impl_mch	1 / 148 s	1 / 254 s	1 / 256 s	1 / 259 s	0 / 121 s	0 / 134 s	1 / 3 s	0 / 121 s	0 / 361 s	0 / 361 s	0 / 362 s	0 / 361 s
98	EventBPrologPackages/BinarySearch/binarySearchFail_impl_mch_v2	0 / 65 s	1 / 154 s	1 / 179 s	1 / 256 s	0 / 121 s	0 / 134 s	1 / 3 s	0 / 122 s	0 / 361 s	0 / 361 s	0 / 362 s	0 / 361 s
99	EventBPrologPackages/BinarySearch/binary_search_mch	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s	1 / 241 s	1 / 242 s	1 / 241 s	1 / 241 s	2 / 131 s	2 / 167 s	2 / 166 s	2 / 132 s
100	EventBPrologPackages/BinarySearch/binary_search_prob_mch	3 / 3 s	3 / 3 s	3 / 3 s	3 / 7 s	3 / 2 s	3 / 22 s	3 / 15 s	3 / 4 s	3 / 15 s	3 / 22 s	3 / 24 s	3 / 16 s
101	EventBPrologPackages/BridgePuzzle/Bridge_mch	1 / 243 s	1 / 231 s	1 / 244 s	1 / 242 s	2 / 127 s	2 / 129 s	2 / 127 s	2 / 127 s	1 / 242 s	1 / 242 s	1 / 242 s	1 / 242 s
102	EventBPrologPackages/Dedale/LandingGears-5-fevrier-2015/LandingSystem_2_mch	3 / 112 s	3 / 42 s	3 / 80 s	3 / 25 s	3 / 4 s	3 / 18 s	3 / 2 s	3 / 3 s	1 / 252 s	1 / 252 s	2 / 193 s	2 / 232 s
103	EventBPrologPackages/Dedale/LandingGears-5-fevrier-2015/LandingSystem_3_mch	2 / 210 s	2 / 219 s	2 / 226 s	3 / 78 s	2 / 159 s	2 / 222 s	2 / 124 s	2 / 166 s	1 / 243 s	1 / 249 s	1 / 248 s	1 / 243 s
104	EventBPrologPackages/Deploy/s1_mch3_trains_mch	0 / 2 s	0 / 2 s	0 / 2 s	0 / 2 s	0 / 187 s	0 / 168 s	0 / 175 s	0 / 177 s	0 / 243 s	0 / 245 s	0 / 244 s	0 / 245 s
105	EventBPrologPackages/Deploy/s4_mch9_schedule_mch	0 / 7 s	0 / 6 s	0 / 7 s	0 / 9 s	0 / 123 s	0 / 135 s	0 / 123 s	0 / 124 s	0 / 383 s	0 / 375 s	0 / 372 s	0 / 372 s
106	EventBPrologPackages/EventB2Java/MIO_ref3_mch	0 / 243 s	0 / 37 s	0 / 243 s	0 / 243 s	0 / 241 s	0 / 242 s	0 / 241 s	0 / 241 s	0 / 245 s	0 / 245 s	0 / 246 s	0 / 246 s
107	EventBPrologPackages/EventB2Java/MIO_ref4_mch	0 / 244 s	0 / 19 s	0 / 244 s	0 / 245 s	0 / 241 s	0 / 244 s	0 / 241 s	0 / 241 s	0 / 140 s	0 / 242 s	0 / 242 s	0 / 204 s
108	EventBPrologPackages/EventB2Java/MIO_ref6_mch	0 / 255 s	0 / 75 s	0 / 254 s	0 / 258 s	0 / 241 s	0 / 256 s	0 / 242 s	0 / 241 s	0 / 150 s	0 / 70 s	0 / 244 s	0 / 154 s
109	EventBPrologPackages/HD_ABZ16/m5_mch	0 / 198 s	0 / 7 s	0 / 10 s	0 / 177 s	0 / 137 s	0 / 143 s	0 / 132 s	0 / 137 s	0 / 243 s	0 / 242 s	0 / 243 s	0 / 243 s
110	EventBPrologPackages/OLSR/M1_mch	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
111	EventBPrologPackages/ProofDirected/WD_PO_MC_Test_mch	3 / 1 s	3 / 1 s	3 / 1 s	3 / 4 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
112	EventBPrologPackages/ProofDirected/benchmarks/Cruise_finite1	3 / 8 s	3 / 7 s	3 / 9 s	3 / 18 s	1 / 250 s	1 / 250 s	1 / 258 s	1 / 251 s	1 / 263 s	1 / 265 s	1 / 280 s	1 / 262 s
113	EventBPrologPackages/ProofDirected/benchmarks/cars3	3 / 5 s	3 / 4 s	3 / 5 s	3 / 14 s	3 / 13 s	3 / 11 s	3 / 12 s	2 / 125 s	1 / 140 s	1 / 249 s	1 / 252 s	1 / 146 s
114	EventBPrologPackages/ProofDirected/benchmarks/earley_3	0 / 379 s	0 / 6 s	0 / 381 s	0 / 369 s	1 / 242 s	1 / 247 s	1 / 242 s	1 / 242 s	2 / 247 s	1 / 258 s	1 / 263 s	1 / 259 s
115	EventBPrologPackages/ProofDirected/benchmarks/earley_3_autoproof	0 / 374 s	0 / 6 s	0 / 376 s	0 / 367 s	1 / 242 s	1 / 246 s	1 / 242 s	1 / 242 s	2 / 243 s	1 / 262 s	1 / 259 s	1 / 261 s
116	EventBPrologPackages/ProofDirected/benchmarks/mondex_m2	0 / 245 s	0 / 244 s	0 / 245 s	0 / 248 s	1 / 122 s	1 / 125 s	1 / 122 s	1 / 139 s	0 / 252 s	0 / 249 s	0 / 252 s	0 / 254 s
117	EventBPrologPackages/ProofDirected/benchmarks/mondex_m3	0 / 246 s	0 / 245 s	0 / 246 s	0 / 250 s	1 / 123 s	1 / 124 s	1 / 124 s	1 / 131 s	0 / 250 s	0 / 253 s	0 / 255 s	0 / 249 s
118	EventBPrologPackages/SSF/Bepi_Soton/M0_mch	0 / 2 s	0 / 2 s	0 / 3 s	0 / 4 s	0 / 121 s	0 / 122 s	0 / 121 s	0 / 121 s	0 / 244 s	0 / 246 s	0 / 247 s	0 / 245 s
119	EventBPrologPackages/SSF/Bepi_Soton/M2_mch	2 / 3 s	3 / 3 s	3 / 3 s	2 / 7 s	2 / 121 s	2 / 122 s	2 / 121 s	2 / 121 s	2 / 144 s	2 / 201 s	2 / 164 s	2 / 134 s
120	EventBPrologPackages/SSF/DSAOCSSv002/StuffForClassicProB/ModeProtocolMachine_mch	0 / 375 s	0 / 247 s	0 / 375 s	0 / 370 s	0 / 244 s	0 / 245 s	0 / 244 s	0 / 244 s	0 / 244 s	0 / 244 s	0 / 244 s	0 / 244 s
121	EventBPrologPackages/Stefan/quicksort/qs9_mch	0 / 5 s	0 / 6 s	0 / 6 s	0 / 8 s	2 / 121 s	2 / 128 s	2 / 122 s	2 / 121 s	0 / 280 s	0 / 280 s	0 / 296 s	0 / 282 s
122	EventBPrologPackages/Swap/Swap1_err_mch	3 / 1 s	3 / 1 s	3 / 2 s	3 / 2 s	1 / 241 s	1 / 242 s	1 / 241 s	1 / 241 s	2 / 152 s	1 / 241 s	1 / 242 s	2 / 152 s
123	EventBPrologPackages/Swap/Swap1_mch	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 4 s	3 / 4 s	3 / 2 s
124	EventBPrologPackages/Tests/UnicodeGermanUmlauteAndJapanesCharacters2_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 5 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 2 s	3 / 2 s
125	EventBPrologPackages/Tickets/Cansell_RingLead/elect2_2_mch	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
126	EventBPrologPackages/Tokeneer/ref5_enrol_mch	2 / 176 s	2 / 170 s	2 / 181 s	2 / 173 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 4 s	3 / 4 s	3 / 3 s	3 / 4 s
127	EventBPrologPackages/TopologyDiscovery/rm_2	0 / 14 s	0 / 5 s	0 / 6 s	0 / 6 s	0 / 243 s	0 / 243 s	0 / 242 s	0 / 242 s	0 / 245 s	0 / 243 s	0 / 243 s	0 / 244 s
128	EventBPrologPackages/arinc653model-master/M1_Mach_PartProc_Trans_mch	0 / 254 s	0 / 31 s	0 / 254 s	0 / 249 s	0 / 242 s	0 / 250 s	0 / 241 s	0 / 241 s	0 / 272 s	0 / 265 s	0 / 268 s	0 / 272 s
129	EventBPrologPackages/arinc653model-master/M2_Mach_PartProc_Trans_with_Events_mch	0 / 274 s	0 / 60 s	0 / 275 s	0 / 271 s	0 / 244 s	0 / 258 s	0 / 244 s	0 / 245 s	0 / 368 s	0 / 310 s	0 / 314 s	0 / 363 s
130	TLC/InvariantViolation/countdown	0 / 1 s	0 / 1 s	0 / 1 s	0 / 2 s	1 / 242 s	1 / 243 s	1 / 243 s	1 / 244 s	1 / 241 s	1 / 241 s	1 / 241 s	1 / 241 s
131	TLC/NoError/CAN_BUS_tlc	1 / 24 s	3 / 10 s	3 / 14 s	3 / 16 s	3 / 11 s	3 / 10 s	3 / 8 s	3 / 30 s	1 / 260 s	1 / 241 s	1 / 242 s	1 / 242 s
132	TLC/NoError/CSM	3 / 3 s	3 / 3 s	3 / 3 s	3 / 7 s	1 / 241 s	1 / 245 s	1 / 241 s	1 / 241 s	2 / 136 s	2 / 155 s	2 / 154 s	3 / 70 s
133	TLC/NoError/safecap2549260349036854403	0 / 296 s	0 / 261 s	0 / 303 s	0 / 264 s	0 / 242 s	0 / 249 s	0 / 243 s	0 / 241 s	0 / 244 s	0 / 244 s	0 / 246 s	0 / 246 s
	Total	225 / 10 699 s	247 / 5238 s	**252 / 9340 s**	238 / 10 317 s	227 / 16 202 s	227 / 16 463 s	226 / 16 363 s	224 / 16 607 s	202 / 21 144 s	195 / 22 361 s	196 / 22 452 s	206 / 20 967 s
		Solved constraints / Runtime s											

Table D.3.: Detailed results of the benchmarks from deadlock freedom proofs used in Section 7.6. In total, 112 constraints were solved.

		ProB-Z3								ProB			
No.	Name	(axiomatic)	(constructive)	(parallel)	(parallel & decomposed)	ProB	ProB-Sym	ProB-CSE	ProB-CHR	SMT	Raw-SMT	Sym-Raw-SMT	Sym-SMT
1	B/ABCD/TheSystem_small	1 / 1 s	0 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
2	B/ABCD/USER_CLASS	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
3	B/ABCD/bookstore	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
4	B/B_Day/Parking_R2	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
5	B/Benchmarks/Cruise_finite_k	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
6	B/Benchmarks/DSP0	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
7	B/Benchmarks/phonebook7_err	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
8	B/Benchmarks/tictac	1 / 4 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
9	B/Demo/Bakery1err2	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
10	B/Demo/RussianPostalPuzzle2	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
11	B/Demo/Simpson/Simpson_Four_Slot	1 / 1 s	1 / 1 s	1 / 1 s	1 / 3 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
12	B/Demo/Simpson/Simpson_Four_Slot_Ordered	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
13	B/Demo/Simpson/Simpson_Four_Slot_Symm	0 / 1 s	0 / 1 s	0 / 1 s	0 / 123 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
14	B/Demo/Simpson/Simpson_Four_Slot_TLC	1 / 1 s	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
15	B/Demo/scheduler_err	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
16	B/EventB/Bosch_mini1_m_3	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
17	B/EventB/Bosch_mini1v2_m_3	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
18	B/EventB/Bosch_mini2_m_3	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 61 s	0 / 62 s	0 / 1 s
19	B/EventB/ETH_Elevator/elevator10	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
20	B/EventB/ETH_Elevator/elevator5	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
21	B/EventB/ETH_Elevator/elevator6	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
22	B/EventB/ETH_Elevator/elevator7	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
23	B/EventB/ETH_Elevator/elevator8	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
24	B/EventB/EventB_Projekt/lift_solution	1 / 1 s	1 / 1 s	1 / 2 s	1 / 3 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
25	B/Ivo/BenchmarksEnablingAnalysis/ABZ_Landing_Gear_Journal/Ref4_ControllerHandle_mch	0 / 20 s	1 / 13 s	1 / 20 s	1 / 2 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
26	B/Ivo/BenchmarksEnablingAnalysis/other/CAN_BUS	0 / 2 s	0 / 1 s	0 / 16 s	0 / 3 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
27	B/Ivo/BenchmarksPGE/Benchmarks/Cruise_finite1	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
28	B/Ivo/BenchmarksPGE/Pathological/AllEnabled_Worst_Case_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
29	B/Ivo/BenchmarksPGE/Pathological/ComplexGuards_Best_Case_mch	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 2 s	1 / 2 s	1 / 2 s	1 / 2 s
30	B/Ivo/BenchmarksPGE/other/CAN_BUS_tlc	0 / 1 s	0 / 67 s	0 / 68 s	0 / 67 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
31	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v1_mch	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
32	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v2_mch	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
33	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v3_mch	1 / 1 s	1 / 1 s	1 / 2 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
34	B/Ivo/BenchmarksPOR/Concurrent_Program_Development/conc_1_mch_finite	0 / 134 s	0 / 1 s	0 / 142 s	0 / 144 s	1 / 5 s	1 / 5 s	1 / 1 s	1 / 8 s	0 / 128 s	0 / 127 s	0 / 127 s	0 / 131 s
35	B/Ivo/BenchmarksPOR/Concurrent_Program_Development/conc_2_mch_finite	0 / 137 s	0 / 12 s	0 / 132 s	0 / 147 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 151 s	1 / 1 s	1 / 1 s	0 / 151 s
36	B/Ivo/BenchmarksPOR/Concurrent_Program_Development/conc_3_mch_finite	0 / 126 s	0 / 15 s	0 / 126 s	0 / 150 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 151 s	1 / 1 s	1 / 1 s	0 / 152 s
37	B/Ivo/BenchmarksPOR/Concurrent_Program_Development/conc_4_mch_finite	0 / 128 s	0 / 15 s	0 / 134 s	0 / 186 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 174 s	1 / 1 s	1 / 1 s	0 / 152 s
38	B/Ivo/BenchmarksPOR/Sieve/sieve_parallel_mch	0 / 122 s	0 / 128 s	0 / 127 s	0 / 130 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 2 s	1 / 2 s	1 / 2 s	1 / 2 s	1 / 2 s
39	B/Ivo/BenchmarksPOR/other/CSM	1 / 1 s	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
40	B/Ivo/BenchmarksPOR/other/ConcurrentCounters	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
41	B/Ivo/NoDisablings_mx5	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
42	B/Ivo/SkippingComplexGuardsEvaluation_mch_mx5000	1 / 7 s	1 / 5 s	1 / 9 s	1 / 5 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
43	B/PragmasUnits/CaseStudies/Abrial_Hybrid/hybrid_flight/f_m0	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
44	B/PragmasUnits/InternalRepresentationTests/Case	1 / 1 s	1 / 0 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 0 s	1 / 1 s	1 / 0 s
45	B/PragmasUnits/InternalRepresentationTests/Case_internal	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 0 s	1 / 1 s	1 / 0 s
46	B/PragmasUnits/InternalRepresentationTests/Case_internal_saved	1 / 0 s	1 / 1 s	1 / 1 s	1 / 0 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 0 s	1 / 1 s	1 / 0 s
47	B/PragmasUnits/InternalRepresentationTests/Case_internal_saved_with_case	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 0 s	1 / 1 s	1 / 1 s
48	B/Special/Dependency/SleepSetAlgorithm/SleepSets	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
49	B/Special/PO_ModelChecking/APBMR4	1 / 1 s	1 / 1 s	1 / 1 s	1 / 4 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
50	B/Special/PO_ModelChecking/Scheduler_Rodin_Provers	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
51	B/Special/PO_ModelChecking/Simple_PO_Check	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 0 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
52	B/Special/PO_ModelChecking/earley_3	0 / 126 s	0 / 1 s	0 / 133 s	0 / 123 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 151 s	0 / 151 s	0 / 152 s	0 / 151 s
53	B/Special/PO_ModelChecking/earley_3_original	0 / 130 s	0 / 1 s	0 / 29 s	0 / 122 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 154 s	0 / 152 s	0 / 152 s	0 / 154 s
54	B/SymbolicModelChecking/DisjunctionInProperties	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
55	B/SymbolicModelChecking/TimingExampleSimpler_v2_VariablesEvenMoreLimited	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
56	B/SymbolicModelChecking/TimingExampleSimpler_v2_VariablesLimited	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
57	B/Tickets/110/Library_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
58	B/Tickets/Guiziou_ClearSy/Machine_SansPrintf	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
59	B/Tickets/Hansen20_WhileInc/Counter	1 / 32 s	1 / 20 s	1 / 34 s	1 / 34 s	1 / 0 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
60	B/Tickets/Hansen20_WhileInc/Counter500	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 0 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
61	B/Tickets/Hansen20_WhileInc/Counter5000	1 / 3 s	1 / 3 s	1 / 3 s	1 / 6 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
62	CSPB/Bank3D	0 / 1 s	0 / 1 s	0 / 2 s	1 / 4 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
63	EventBPrologPackages/ABZ_Landing_Gear_Journal/Ref1_Valve_err	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
64	EventBPrologPackages/ABZ_Landing_Gear_Journal/Ref1_Valve_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
65	EventBPrologPackages/ABZ_Landing_Gear_Journal/Ref2_ControllerOutputs_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
66	EventBPrologPackages/ABZ_Landing_Gear_Journal/Ref3_ControllerSensors_mch	0 / 128 s	0 / 123 s	0 / 127 s	0 / 3 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
67	EventBPrologPackages/ABZ_Landing_Gear_Journal/Ref3_ControllerSensors_mch_eventb	0 / 129 s	0 / 122 s	0 / 131 s	0 / 3 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
68	EventBPrologPackages/ABZ_Landing_Gear_Journal/Ref5_Switch_mch	1 / 23 s	1 / 25 s	1 / 41 s	1 / 3 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
69	EventBPrologPackages/ABZ_Landing_Gear_Journal/Ref6_CockpitLights_mch	1 / 90 s	1 / 87 s	1 / 72 s	1 / 18 s	1 / 1 s	1 / 3 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s
70	EventBPrologPackages/AbrialAccessControl/mac2_mch	0 / 1 s	0 / 1 s	0 / 2 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
71	EventBPrologPackages/AbrialAccessControl/mac4_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 2 s	0 / 52 s	0 / 122 s	0 / 47 s	0 / 48 s	0 / 32 s	0 / 33 s	0 / 35 s	0 / 35 s
72	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m4_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
73	EventBPrologPackages/Abrial_BRP/DLK_Checking/ch6_brp_F01/b_4_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
74	EventBPrologPackages/Abrial_BRP/DLK_Checking/ch6_brp_OK/b_5_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 124 s	0 / 69 s	0 / 122 s	0 / 122 s	0 / 68 s
75	EventBPrologPackages/Abrial_Modes/mode_m3_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
76	EventBPrologPackages/Abrial_Modes/mode_m4_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
77	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_2_standalone_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
78	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_3_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 124 s	0 / 69 s	0 / 122 s	0 / 122 s	0 / 68 s
79	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_3_standalone_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 129 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
80	EventBPrologPackages/Abrial_Train_Ch17/Train1_Lukas_InitTRK	0 / 127 s	0 / 124 s	0 / 131 s	0 / 122 s	0 / 125 s	0 / 124 s	0 / 125 s	0 / 128 s	0 / 46 s	0 / 33 s	0 / 152 s	0 / 151 s
81	EventBPrologPackages/Abrial_Train_Ch17/Train1_Lukas_POR	0 / 126 s	0 / 123 s	0 / 128 s	0 / 121 s	0 / 121 s	0 / 121 s	0 / 121 s	0 / 121 s	0 / 153 s	0 / 35 s	0 / 152 s	0 / 153 s
82	EventBPrologPackages/Abrial_Train_Ch17/train_0_mch	0 / 113 s	0 / 1 s	0 / 124 s	0 / 123 s	0 / 7 s	0 / 7 s	0 / 1 s	0 / 8 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
83	EventBPrologPackages/Abrial_Train_Ch17/train_1_prob_POR_mch	0 / 127 s	0 / 123 s	0 / 127 s	0 / 122 s	0 / 121 s	0 / 121 s	0 / 121 s	0 / 122 s	0 / 151 s	0 / 152 s	0 / 153 s	0 / 152 s
84	EventBPrologPackages/Abrial_Train_Ch17/train_1_prob_mch	0 / 128 s	0 / 123 s	0 / 127 s	0 / 122 s	0 / 121 s	0 / 121 s	0 / 122 s	0 / 122 s	0 / 44 s	0 / 33 s	0 / 153 s	0 / 152 s
85	EventBPrologPackages/Advance/CAN_Bus/CB3FSMM_mch_v2_wo_finite_inv	1 / 1 s	0 / 77 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
86	EventBPrologPackages/BinarySearch/binarySearchFail_impl_mch	0 / 1 s	0 / 37 s	0 / 32 s	0 / 41 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 233 s	0 / 154 s	1 / 32 s	0 / 118 s
87	EventBPrologPackages/BinarySearch/binarySearchFail_impl_mch_v2	0 / 1 s	0 / 37 s	0 / 44 s	0 / 41 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 204 s	0 / 194 s	0 / 151 s	0 / 192 s
88	EventBPrologPackages/BinarySearch/binary_search_mch	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 31 s	1 / 31 s	1 / 32 s	1 / 31 s
89	EventBPrologPackages/BinarySearch/binary_search_prob_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
90	EventBPrologPackages/BridgePuzzle/Bridge_mch	0 / 1 s	1 / 1 s	1 / 1 s	0 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
91	EventBPrologPackages/Deploy/s1_mch3_trains_mch	0 / 1 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
92	EventBPrologPackages/Deploy/s4_mch9_schedule_mch	0 / 1 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 2 s	0 / 2 s
93	EventBPrologPackages/EventB2Java/MIO_ref3_mch	0 / 6 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
94	EventBPrologPackages/EventB2Java/MIO_ref4_mch	0 / 24 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
95	EventBPrologPackages/EventB2Java/MIO_ref6_mch	0 / 28 s	0 / 1 s	0 / 2 s	0 / 2 s	0 / 1 s	0 / 123 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 2 s	0 / 1 s
96	EventBPrologPackages/HD_ABZ16/m5_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
97	EventBPrologPackages/OLSR/M2_corrected	0 / 145 s	0 / 1 s	0 / 146 s	0 / 1 s	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
98	EventBPrologPackages/ProofDirected/WD_PO_MC_Test_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
99	EventBPrologPackages/ProofDirected/benchmarks/earley_3_autoproof	0 / 125 s	0 / 1 s	0 / 123 s	0 / 124 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 151 s	0 / 151 s	0 / 145 s	0 / 152 s
100	EventBPrologPackages/SSF/Bepi_Soton/M1_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
101	EventBPrologPackages/SSF/Bepi_Soton/M2_mch	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
102	EventBPrologPackages/SSF/DSAOCSSv002/StuffForClassicProB/ModeProtocolMachine_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 125 s	0 / 126 s	0 / 125 s	0 / 126 s	0 / 32 s	0 / 33 s	0 / 34 s	0 / 34 s
103	EventBPrologPackages/Stefan/quicksort/qs9_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 125 s	0 / 126 s	0 / 124 s	0 / 127 s	1 / 1 s	1 / 1 s	0 / 98 s	1 / 1 s
104	EventBPrologPackages/Swap/Swap1_err_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
105	EventBPrologPackages/Swap/Swap1_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
106	EventBPrologPackages/Tickets/Cansell_RingLead/elect2_2_mch	0 / 22 s	0 / 71 s	0 / 126 s	0 / 125 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 125 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
107	EventBPrologPackages/Tickets/ProjectionDiagram/m3_mch	1 / 1 s	1 / 1 s	1 / 2 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
108	EventBPrologPackages/Tokeneer/ref5_enrol_mch	0 / 1 s	0 / 1 s	0 / 2 s	0 / 2 s	1 / 1 s	1 / 9 s	1 / 1 s	1 / 1 s	0 / 121 s	0 / 121 s	0 / 121 s	0 / 121 s
109	EventBPrologPackages/TopologyDiscovery/rm_2	0 / 121 s	0 / 121 s	0 / 122 s	0 / 121 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
110	EventBPrologPackages/arinc653model-master/M1_Mach_PartProc_Trans_mch	0 / 1 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
111	EventBPrologPackages/arinc653model-master/M2_Mach_PartProc_Trans_with_Events_mch	0 / 1 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
112	TLC/InvariantViolation/countdown	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 122 s	0 / 124 s	0 / 217 s	0 / 123 s	0 / 31 s	0 / 31 s	0 / 32 s	0 / 31 s
	Total	64 / 2516 s	64 / 1560 s	66 / 2679 s	66 / 2433 s	**79 / 1023 s**	79 / 1233 s	79 / 1107 s	79 / 1534 s	70 / 2371 s	73 / 1828 s	73 / 2125 s	70 / 2441 s
								Solved constraints / Runtime s					

Table D.4.: Detailed results of the inductive invariant checking benchmarks used in Section 7.6. In total, 1302 constraints were solved.

		ProB-Z3								ProB			
No.	Name	(axiomatic)	(constructive)	(parallel)	(parallel & decomposed)	ProB	ProB-Sym	ProB-CSE	ProB-CHR	SMT	Raw-SMT	Sym-Raw-SMT	Sym-SMT
1	B/ABCD/TheSystem_small	10 / 4 s	9 / 3 s	10 / 5 s	9 / 4 s	10 / 2 s	10 / 2 s	10 / 1 s	10 / 2 s	10 / 11 s	10 / 11 s	10 / 11 s	10 / 11 s
2	B/ABCD/USER_CLASS	0 / 62 s	0 / 21 s	0 / 16 s	0 / 19 s	0 / 10 s	0 / 8 s	0 / 10 s	0 / 12 s	0 / 12 s	0 / 11 s	0 / 10 s	0 / 11 s
3	B/ABCD/bookstore	2 / 10 s	2 / 1 s	2 / 6 s	2 / 92 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 2 s	3 / 4 s	3 / 3 s	3 / 5 s	3 / 5 s
4	B/BZTT/GSM_revue	5 / 123 s	5 / 2 s	5 / 123 s	5 / 3 s	6 / 1 s	6 / 1 s	6 / 3 s	6 / 1 s	6 / 2 s	6 / 3 s	6 / 3 s	6 / 3 s
5	B/B_Day/Parking_R2	4 / 1 s	4 / 1 s	4 / 1 s	4 / 2 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	3 / 42 s	3 / 52 s	3 / 52 s	3 / 42 s
6	B/Benchmarks/CSM	13 / 2 s	13 / 2 s	13 / 3 s	13 / 19 s	13 / 1 s	13 / 1 s	13 / 1 s	13 / 1 s	13 / 1 s	13 / 1 s	4 / 1084 s	5 / 963 s
7	B/Benchmarks/Cruise_finite_k	26 / 10 s	26 / 10 s	26 / 11 s	26 / 17 s	26 / 1 s	26 / 2 s	26 / 1 s	26 / 1 s	26 / 3 s	26 / 4 s	26 / 5 s	26 / 4 s
8	B/Benchmarks/DSP0	4 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s
9	B/Benchmarks/phonebook7_err	2 / 1 s	1 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 3 s	2 / 2 s	2 / 3 s	2 / 2 s
10	B/Benchmarks/spec	4 / 2 s	4 / 2 s	4 / 2 s	5 / 2 s	5 / 12 s	5 / 14 s	5 / 11 s	5 / 11 s	6 / 33 s	6 / 28 s	6 / 44 s	6 / 42 s
11	B/Benchmarks/tictac	0 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	0 / 243 s	0 / 243 s	0 / 243 s	0 / 243 s	1 / 122 s	1 / 123 s	1 / 129 s	1 / 125 s
12	B/Demo/Bakery1err2	6 / 1 s	6 / 1 s	6 / 2 s	6 / 3 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s
13	B/Demo/RussianPostalPuzzle2	4 / 3 s	5 / 1 s	5 / 2 s	3 / 4 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s
14	B/Demo/Simpson/Simpson_Four_Slot	8 / 2 s	9 / 2 s	9 / 2 s	9 / 8 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s
15	B/Demo/Simpson/Simpson_Four_Slot_Ordered	7 / 1 s	8 / 1 s	9 / 2 s	9 / 6 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s
16	B/Demo/Simpson/Simpson_Four_Slot_Symm	1 / 246 s	1 / 3 s	1 / 5 s	1 / 4 s	0 / 27 s	0 / 30 s	0 / 27 s	0 / 32 s	1 / 8 s	1 / 10 s	1 / 10 s	1 / 6 s
17	B/Demo/Simpson/Simpson_Four_Slot_TLC	8 / 1 s	9 / 1 s	9 / 2 s	9 / 5 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s
18	B/Demo/phonebook5	1 / 1 s	0 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
19	B/Demo/scheduler_err	1 / 1 s	0 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
20	B/ErrorMachines/OneInvariantViolation	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
21	B/EventB/Bosch_mini1_m_3	12 / 2 s	12 / 2 s	12 / 3 s	12 / 6 s	10 / 243 s	10 / 243 s	10 / 243 s	10 / 242 s	12 / 121 s	12 / 121 s	12 / 122 s	12 / 121 s
22	B/EventB/Bosch_mini1v2_m_3	12 / 2 s	12 / 2 s	12 / 3 s	12 / 7 s	10 / 242 s	10 / 242 s	10 / 242 s	10 / 242 s	12 / 121 s	12 / 121 s	12 / 122 s	12 / 121 s
23	B/EventB/Bosch_mini2_m_3	7 / 1 s	7 / 1 s	7 / 1 s	7 / 4 s	5 / 241 s	5 / 241 s	6 / 121 s	6 / 121 s	7 / 91 s	7 / 91 s	7 / 91 s	7 / 91 s
24	B/EventB/ETH_Elevator/elevator10	16 / 13 s	34 / 12 s	34 / 15 s	34 / 10 s	14 / 2403 s	14 / 2404 s	14 / 2403 s	14 / 2403 s	15 / 1369 s	15 / 1192 s	13 / 1574 s	13 / 1551 s
25	B/EventB/ETH_Elevator/elevator5	10 / 2 s	10 / 2 s	10 / 2 s	10 / 7 s	9 / 121 s	9 / 121 s	9 / 121 s	9 / 121 s	10 / 31 s	10 / 31 s	10 / 31 s	10 / 31 s
26	B/EventB/ETH_Elevator/elevator6	13 / 2 s	13 / 2 s	13 / 3 s	13 / 5 s	12 / 121 s	12 / 121 s	12 / 121 s	12 / 121 s	13 / 91 s	12 / 121 s	12 / 121 s	13 / 91 s
27	B/EventB/ETH_Elevator/elevator7	16 / 4 s	16 / 4 s	16 / 5 s	16 / 4 s	15 / 121 s	15 / 121 s	15 / 121 s	15 / 121 s	16 / 91 s	15 / 121 s	15 / 121 s	16 / 91 s
28	B/EventB/ETH_Elevator/elevator8	16 / 6 s	16 / 5 s	16 / 6 s	16 / 4 s	14 / 241 s	14 / 241 s	14 / 241 s	14 / 241 s	15 / 212 s	15 / 212 s	15 / 212 s	15 / 212 s
29	B/EventB/EventB_Projekt/lift_solution	15 / 8 s	21 / 7 s	21 / 8 s	21 / 22 s	21 / 3 s	21 / 3 s	21 / 3 s	21 / 3 s	21 / 6 s	21 / 9 s	21 / 11 s	21 / 7 s
30	B/EventB/SiemensMiniPilot_Abrial_mch_0	8 / 2 s	8 / 2 s	8 / 3 s	8 / 5 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s
31	B/Ivo/BenchmarksEnablingAnalysis/Abrial_LandingGear3/m0_mch	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s
32	B/Ivo/BenchmarksEnablingAnalysis/Abrial_LandingGear3/m1_mch	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s
33	B/Ivo/BenchmarksEnablingAnalysis/Abrial_LandingGear3/m2_mch	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s	11 / 1 s
34	B/Ivo/BenchmarksEnablingAnalysis/Abrial_LandingGear3/m4_mch	26 / 3 s	26 / 3 s	26 / 4 s	26 / 3 s	26 / 1 s	26 / 3 s	26 / 1 s	26 / 1 s	26 / 1 s	26 / 1 s	26 / 1 s	26 / 1 s
35	B/Ivo/BenchmarksEnablingAnalysis/other/CAN_BUS	15 / 381 s	12 / 1 s	15 / 372 s	15 / 488 s	15 / 620 s	15 / 612 s	15 / 601 s	15 / 607 s	16 / 638 s	16 / 702 s	14 / 863 s	15 / 676 s
36	B/Ivo/BenchmarksPGE/Benchmarks/Cruise_finite1	26 / 10 s	26 / 9 s	26 / 11 s	26 / 16 s	26 / 1 s	26 / 2 s	26 / 1 s	26 / 1 s	26 / 4 s	26 / 4 s	26 / 5 s	26 / 4 s
37	B/Ivo/BenchmarksPGE/Pathological/AllEnabled_Worst_Case_mch	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s
38	B/Ivo/BenchmarksPGE/Pathological/ComplexGuards_Best_Case_mch	21 / 4 s	21 / 4 s	21 / 4 s	21 / 12 s	21 / 1 s	21 / 1 s	21 / 1 s	21 / 1 s	21 / 1 s	21 / 1 s	21 / 2 s	21 / 2 s
39	B/Ivo/BenchmarksPGE/abz_case_study/Ref4_ControllerHandle_mch	30 / 1 s	30 / 1 s	30 / 2 s	30 / 3 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 5 s	30 / 5 s
40	B/Ivo/BenchmarksPGE/other/CAN_BUS_tlc	17 / 7 s	21 / 2 s	21 / 3 s	21 / 12 s	18 / 400 s	18 / 404 s	16 / 609 s	18 / 474 s	16 / 601 s	20 / 422 s	15 / 781 s	16 / 661 s
41	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v1_mch	9 / 1 s	9 / 1 s	9 / 1 s	9 / 7 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s
42	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v2_mch	9 / 1 s	9 / 1 s	9 / 1 s	9 / 4 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s
43	B/Ivo/BenchmarksPOR/Concurrent/fact_m_v3_mch	8 / 1 s	8 / 1 s	8 / 1 s	8 / 5 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s	8 / 1 s
44	B/Ivo/BenchmarksPOR/other/ConcurrentCounters	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s
45	B/Ivo/NoDisablings_mx5	6 / 1 s	6 / 1 s	6 / 2 s	6 / 2 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s	6 / 1 s
46	B/Ivo/SkippingComplexGuardsEvaluation_mch_mx5000	16 / 5 s	16 / 4 s	16 / 7 s	16 / 8 s	16 / 1 s	16 / 1 s	16 / 1 s	16 / 1 s	16 / 1 s	16 / 1 s	7 / 1270 s	7 / 1261 s
47	B/PerformanceTests/Generated/GeneratedMod100	100 / 3 s	100 / 2 s	100 / 2 s	100 / 2 s	100 / 9 s	100 / 52 s	100 / 14 s	100 / 9 s	100 / 1 s	100 / 1 s	100 / 1 s	100 / 1 s
48	B/PerformanceTests/PartialFunInverse	1 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s
49	B/PragmasUnits/CaseStudies/Abrial_Hybrid/hybrid_flight/f_m0	5 / 1 s	5 / 1 s	5 / 1 s	5 / 2 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s
50	B/PragmasUnits/InternalRepresentationTests/Case	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
51	B/PragmasUnits/InternalRepresentationTests/Case_internal	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
52	B/PragmasUnits/InternalRepresentationTests/Case_internal_saved	3 / 1 s	3 / 1 s	3 / 1 s	3 / 3 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
53	B/PragmasUnits/InternalRepresentationTests/Case_internal_saved_with_case	3 / 1 s	3 / 1 s	3 / 1 s	3 / 2 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
54	B/Puzzles/PartialSets	3 / 1 s	3 / 1 s	3 / 1 s	2 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
55	B/SchneiderBook/Chapter16/Mult	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s
56	B/SchneiderBook/LabMaterial/records	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
57	B/Simple/TwoPurses	4 / 1 s	6 / 1 s	6 / 1 s	6 / 2 s	7 / 98 s	7 / 105 s	7 / 98 s	6 / 121 s	7 / 4 s	7 / 4 s	7 / 4 s	7 / 4 s
58	B/Special/Dependency/SleepSetAlgorithm/SleepSets	4 / 2 s	4 / 1 s	4 / 2 s	4 / 3 s	5 / 1 s	5 / 1 s	5 / 2 s	5 / 1 s	4 / 135 s	4 / 130 s	4 / 139 s	4 / 139 s
59	B/Special/PO_ModelChecking/APBMR4	0 / 4 s	0 / 4 s	0 / 4 s	8 / 20 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
60	B/Special/PO_ModelChecking/Scheduler_Rodin_Provers	4 / 2 s	4 / 2 s	4 / 3 s	4 / 3 s	6 / 18 s	6 / 18 s	6 / 21 s	6 / 17 s	6 / 7 s	6 / 7 s	6 / 7 s	6 / 7 s
61	B/Special/PO_ModelChecking/Simple_PO_Check	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
62	B/Special/PO_ModelChecking/earley_3_original	4 / 363 s	1 / 1 s	4 / 363 s	3 / 378 s	2 / 606 s	2 / 606 s	2 / 605 s	2 / 605 s	4 / 432 s	4 / 428 s	4 / 437 s	4 / 436 s
63	B/SymbolicModelChecking/DisjunctionInProperties	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s
64	B/SymbolicModelChecking/LightbotAbstract	7 / 1 s	7 / 1 s	7 / 1 s	7 / 3 s	7 / 21 s	7 / 27 s	7 / 20 s	7 / 34 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s
65	B/SymbolicModelChecking/TimingExampleSimpler_v2_VariablesEvenMoreLimited	4 / 1 s	4 / 1 s	4 / 1 s	4 / 2 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 41 s	4 / 41 s	4 / 42 s	4 / 41 s
66	B/SymbolicModelChecking/TimingExampleSimpler_v2_VariablesLimited	4 / 1 s	4 / 1 s	4 / 1 s	4 / 2 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 1 s	4 / 41 s	4 / 41 s	4 / 42 s	4 / 41 s
67	B/Tester/Any	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s
68	B/Tickets/Guiziou_ClearSy/Machine_SansPrintf	3 / 1 s	2 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s
69	B/Tickets/Treharne1/House_Tracker	0 / 377 s	0 / 421 s	0 / 439 s	0 / 425 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 121 s	2 / 151 s	2 / 151 s	2 / 151 s	2 / 151 s
70	CSPB/Bank3D	5 / 489 s	5 / 5 s	5 / 8 s	5 / 8 s	5 / 1204 s	5 / 1205 s	5 / 1204 s	5 / 1205 s	5 / 558 s	5 / 559 s	5 / 1279 s	5 / 1248 s
71	EventBPrologPackages/ABZ_Landing_Gear/Ref3_ControllerSensors_mch	24 / 0 s	24 / 1 s	24 / 1 s	24 / 1 s	24 / 1 s	24 / 1 s	24 / 1 s	24 / 1 s	24 / 1 s	24 / 1 s	24 / 1 s	24 / 1 s
72	EventBPrologPackages/ABZ_Landing_Gear/Ref5_Switch_mch	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s	30 / 1 s
73	EventBPrologPackages/AbrialAccessControl/mac2_mch	0 / 5 s	0 / 1 s	0 / 3 s	0 / 4 s	1 / 23 s	1 / 19 s	1 / 27 s	1 / 24 s	1 / 254 s	1 / 249 s	1 / 253 s	1 / 250 s
74	EventBPrologPackages/AbrialAccessControl/mac4_mch	0 / 10 s	0 / 3 s	0 / 7 s	0 / 3 s	0 / 1414 s	0 / 1416 s	0 / 1452 s	0 / 1418 s	0 / 963 s	0 / 961 s	0 / 1204 s	0 / 1206 s
75	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m1_mch	9 / 3 s	9 / 3 s	9 / 4 s	9 / 5 s	7 / 1 s	7 / 1 s	7 / 1 s	7 / 1 s	6 / 31 s	6 / 31 s	6 / 32 s	6 / 31 s
76	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m2_mch	12 / 4 s	12 / 4 s	12 / 4 s	12 / 6 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	8 / 32 s	8 / 32 s	8 / 34 s	8 / 35 s
77	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m3_mch	12 / 4 s	12 / 4 s	12 / 5 s	12 / 8 s	9 / 1 s	9 / 1 s	9 / 1 s	9 / 1 s	8 / 31 s	8 / 31 s	8 / 32 s	8 / 31 s
78	EventBPrologPackages/AbrialCrsCtl/CrsCtl_m4_mch	15 / 4 s	15 / 4 s	15 / 5 s	15 / 5 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	9 / 121 s	9 / 121 s	8 / 151 s	8 / 151 s
79	EventBPrologPackages/Abrial_BRP/DLK_Checking/ch6_brp_F01/b_4_mch	1 / 2 s	1 / 2 s	1 / 2 s	1 / 2 s	1 / 361 s	1 / 361 s	1 / 241 s	1 / 361 s	2 / 176 s	2 / 197 s	2 / 197 s	2 / 176 s
80	EventBPrologPackages/Abrial_BRP/DLK_Checking/ch6_brp_OK/b_5_mch	1 / 5 s	1 / 5 s	1 / 5 s	1 / 7 s	1 / 601 s	1 / 602 s	1 / 841 s	1 / 721 s	2 / 581 s	1 / 636 s	1 / 635 s	2 / 582 s
81	EventBPrologPackages/Abrial_Modes/mode_m3_mch	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 0 s	19 / 1 s	19 / 1 s	19 / 1 s
82	EventBPrologPackages/Abrial_Modes/mode_m4_mch	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s	19 / 1 s
83	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_2_standalone_mch	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s	10 / 1 s
84	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_3_mch	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s
85	EventBPrologPackages/Abrial_Teaching/ch4_brp/brp_3_standalone_mch	14 / 2 s	14 / 2 s	14 / 3 s	14 / 3 s	14 / 1 s	14 / 1 s	14 / 1 s	14 / 1 s	14 / 1 s	14 / 1 s	14 / 2 s	14 / 1 s
86	EventBPrologPackages/Advance/CAN_Bus/CB3FSMM_mch_v2_wo_finite_inv	18 / 1 s	20 / 1 s	20 / 1 s	20 / 3 s	18 / 121 s	18 / 121 s	18 / 121 s	18 / 121 s	19 / 31 s	19 / 31 s	19 / 151 s	19 / 151 s
87	EventBPrologPackages/BinarySearch/binarySearchFail_impl_mch	6 / 7 s	9 / 18 s	9 / 19 s	9 / 20 s	6 / 241 s	6 / 243 s	6 / 242 s	6 / 242 s	5 / 601 s	5 / 601 s	5 / 602 s	5 / 601 s
88	EventBPrologPackages/BinarySearch/binarySearchFail_impl_mch_v2	6 / 5 s	9 / 40 s	9 / 43 s	9 / 85 s	6 / 241 s	6 / 243 s	6 / 242 s	6 / 242 s	5 / 602 s	5 / 602 s	5 / 604 s	5 / 602 s
89	EventBPrologPackages/BinarySearch/binary_search_mch	10 / 2 s	12 / 2 s	12 / 2 s	12 / 3 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	11 / 212 s	12 / 122 s	12 / 99 s	12 / 92 s
90	EventBPrologPackages/BinarySearch/binary_search_prob_mch	12 / 4 s	12 / 2 s	12 / 3 s	12 / 9 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s	12 / 1 s
91	EventBPrologPackages/BridgePuzzle/Bridge_mch	2 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 1 s	3 / 5 s	3 / 5 s	3 / 4 s	3 / 5 s
92	EventBPrologPackages/Deploy/s1_mch3_trains_mch	3 / 2 s	3 / 2 s	3 / 2 s	2 / 2 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	3 / 2 s	3 / 2 s	3 / 2 s	3 / 2 s
93	EventBPrologPackages/EventB2Java/MIO_ref3_mch	2 / 6 s	2 / 3 s	2 / 4 s	2 / 4 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 2 s	2 / 1 s
94	EventBPrologPackages/EventB2Java/MIO_ref4_mch	7 / 7 s	7 / 2 s	7 / 4 s	7 / 3 s	7 / 2 s	7 / 3 s	7 / 3 s	7 / 3 s	7 / 4 s	7 / 4 s	7 / 5 s	7 / 4 s
95	EventBPrologPackages/EventB2Java/MIO_ref6_mch	11 / 9 s	11 / 2 s	11 / 6 s	11 / 5 s	11 / 390 s	11 / 393 s	11 / 404 s	11 / 390 s	11 / 166 s	11 / 165 s	11 / 287 s	11 / 285 s
96	EventBPrologPackages/HD_ABZ16/m5_mch	16 / 3 s	16 / 2 s	16 / 3 s	17 / 3 s	16 / 1 s	16 / 1 s	16 / 1 s	16 / 1 s	16 / 1 s	16 / 1 s	16 / 1 s	16 / 1 s
97	EventBPrologPackages/OLSR/M1_mch	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s
98	EventBPrologPackages/OLSR/M2_corrected	2 / 376 s	2 / 4 s	2 / 128 s	2 / 127 s	4 / 601 s	4 / 602 s	4 / 746 s	4 / 602 s	4 / 665 s	3 / 654 s	3 / 731 s	3 / 753 s
99	EventBPrologPackages/ProofDirected/WD_PO_MC_Test_mch	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s	1 / 1 s
100	EventBPrologPackages/ProofDirected/benchmarks/earley_3	3 / 367 s	1 / 2 s	3 / 371 s	2 / 364 s	2 / 481 s	2 / 482 s	2 / 483 s	2 / 481 s	3 / 575 s	3 / 575 s	3 / 576 s	3 / 576 s
101	EventBPrologPackages/ProofDirected/benchmarks/earley_3_autoproof	3 / 366 s	1 / 2 s	3 / 370 s	2 / 363 s	2 / 481 s	2 / 481 s	2 / 483 s	2 / 481 s	2 / 482 s	2 / 482 s	2 / 483 s	2 / 482 s
102	EventBPrologPackages/SSF/Bepi_Soton/M0_mch	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s	0 / 1 s
103	EventBPrologPackages/SSF/Bepi_Soton/M2_mch	5 / 3 s	2 / 3 s	5 / 3 s	5 / 2 s	5 / 3 s	5 / 5 s	5 / 2 s	5 / 3 s	5 / 130 s	5 / 146 s	5 / 221 s	5 / 195 s
104	EventBPrologPackages/Stefan/gcd/g6_mch_with_6small_values	1 / 2 s	1 / 2 s	1 / 3 s	1 / 364 s	4 / 1 s	4 / 1 s	4 / 2 s	4 / 1 s	1 / 461 s	1 / 465 s	1 / 477 s	1 / 473 s
105	EventBPrologPackages/Swap/Swap1_err_mch	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s
106	EventBPrologPackages/Swap/Swap1_mch	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s
107	EventBPrologPackages/Tickets/Cansell_RingLead/elect2_2_mch	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s	5 / 1 s
108	EventBPrologPackages/Tickets/ProjectionDiagram/m3_mch	10 / 1 s	10 / 1 s	10 / 1 s	10 / 2 s	5 / 241 s	5 / 241 s	5 / 241 s	5 / 241 s	7 / 61 s	7 / 61 s	7 / 61 s	7 / 61 s
109	EventBPrologPackages/Tokeneer/ref5_enrol_mch	37 / 1 s	37 / 1 s	37 / 1 s	37 / 1 s	37 / 1 s	37 / 2 s	37 / 1 s	37 / 1 s	37 / 1 s	37 / 1 s	37 / 1 s	37 / 1 s
110	EventBPrologPackages/TopologyDiscovery/rm_2	2 / 1 s	2 / 1 s	2 / 1 s	2 / 1 s	2 / 4 s	2 / 5 s	2 / 5 s	2 / 4 s	2 / 32 s	2 / 33 s	2 / 30 s	2 / 34 s
111	EventBPrologPackages/arinc653model-master/M1_Mach_PartProc_Trans_mch	0 / 308 s	0 / 6 s	0 / 7 s	0 / 4 s	0 / 1 s	0 / 2 s	0 / 1 s	0 / 1 s	0 / 14 s	0 / 15 s	0 / 17 s	0 / 12 s
112	EventBPrologPackages/arinc653model-master/M2_Mach_PartProc_Trans_with_Events_mch	1 / 470 s	1 / 10 s	1 / 16 s	1 / 7 s	1 / 2 s	1 / 3 s	1 / 2 s	1 / 2 s	1 / 46 s	1 / 45 s	1 / 45 s	1 / 47 s
113	TLC/Laws/SetLawsPow	1 / 675 s	1 / 505 s	1 / 619 s	0 / 735 s	1 / 657 s	1 / 649 s	1 / 626 s	1 / 654 s	0 / 733 s	0 / 736 s	0 / 734 s	0 / 733 s
114	TLC/Laws/SubsetLaws	1 / 435 s	3 / 4 s	3 / 244 s	1 / 488 s	4 / 407 s	4 / 401 s	5 / 360 s	4 / 421 s	1 / 484 s	0 / 602 s	0 / 601 s	0 / 601 s
115	TLC/NoError/USB_4Endpoints	30 / 77 s	30 / 2 s	30 / 256 s	30 / 152 s	30 / 23 s	30 / 28 s	30 / 2 s	30 / 28 s	30 / 16 s	30 / 16 s	30 / 16 s	30 / 16 s
116	TLC/NoError/safecap2549260349036854403	2 / 671 s	1 / 249 s	2 / 148 s	3 / 39 s	2 / 846 s	2 / 848 s	2 / 847 s	2 / 847 s	3 / 309 s	3 / 303 s	3 / 571 s	3 / 567 s
	Total	1014 / 6043 s	1046 / 1489 s	**1066 / 3794 s**	1066 / 4542 s	1013 / 14 340 s	1013 / 14 409 s	1013 / 14 649 s	1013 / 14 468 s	1021 / 12 861 s	1021 / 12 735 s	993 / 17 698 s	999 / 17 013 s

Solved constraints / Runtime s

Bibliography

[1] John N. Buxton and Brian Randell. Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee. `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF`, 1969. Rome, Italy (published April 1970). [Online; accessed 28-February-2023].

[2] Dominique Méry and Neeraj Kumar Singh. Formal Specification of Medical Systems by Proof-Based Refinement. *ACM Transactions on Embedded Computing Systems*, 12(1), 2013. doi: 10.1145/2406336.2406351.

[3] Donald Ervin Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition.* Addison-Wesley, 1997.

[4] Chao Wang, Franjo Ivančić, Malay Ganai, and Aarti Gupta. Deciding Separation Logic Formulae by SAT and Incremental Negative Cycle Elimination. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 322–336. Springer, 2005. doi: 10.1007/11591191_23.

[5] Peter Bruch. Memory leak in Z3 Context using .NET API. `https://github.com/Z3Prover/z3/issues/5043/#issuecomment-898591449`, 2021. User comment on Github issue. [Online; accessed 2-Dec-2022].

[6] Nikolaj Bjørner. Memory leak in Z3 Context using .NET API. `https://github.com/Z3Prover/z3/issues/5043/#issuecomment-898967287`, 2021. User comment on Github issue. [Online; accessed 2-Dec-2022].

[7] USSR Academy of Sciences and USSR Academy of Pedagogical Sciences. Kvant Magazine 1985. `http://kvant.mccme.ru/1985/index.htm`, 1985. [Online; accessed 14-June-2022].

[8] Peter Kriens. Formal Specification of the Chameleon Puzzle in Alloy 6. `https://alloytools.discourse.group/t/problem-in-understanding-mutable-state-in-alloy-6/268/4`, 2022. User comment in Discourse forum. [Online; accessed 15-June-2022].

[9] John Allan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965. doi: 10.1145/321250.321253.

[10] John Allan Robinson. Computational Logic: The Unification Computation. *Machine Intelligence*, 6:63–72, 1971.

[11] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960. doi: 10.1145/321033.321034.

[12] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962. doi: 10.1145/368273.368557.

[13] Bernie Cohen. A Brief History of Formal Methods. *Formal Aspects of Computing*, 1995.

[14] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A Successful Application of B in a Large Project. In *World Congress on Formal Methods*, 1999.

[15] Didier Essamé and Daniel Dollé. B in Large-Scale Projects: The Canarsie Line CBTC Experience. In Jacques Julliand and Olga Kouchnarenko, editors, *Proceedings B (International Conference of B Users)*, pages 252–254. Springer, 2006. doi: 10.1007/11955757_21.

[16] Dominik Hansen, Michael Leuschel, David Schneider, Sebastian Krings, Philipp Körner, Thomas Naulin, Nader Nayeri, and Frank Skowron. Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 292–306. Springer, 2018. doi: 10.1007/978-3-319-91271-4_20.

[17] Tommaso Dreossi, Shromona Ghosh, Alberto Sangiovanni-Vincentelli, and Sanjit A. Seshia. A Formalization of Robustness for Deep Neural Networks, 2019.

[18] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems, 2020.

[19] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Star-Based Reachability Analysis of Deep Neural Networks. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, pages 670–686. Springer, 2019. doi: 10.1007/978-3-030-30942-8_39.

[20] Haitham Khedr, James Ferlez, and Yasser Shoukry. Effective Formal Verification of Neural Networks using the Geometry of Linear Regions. *ArXiv*, abs/2006.10864, 2020.

[21] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. VERIFAI: A Toolkit

for the Design and Analysis of Artificial Intelligence-Based Systems. *Computing Research Repository*, abs/1902.04245, 2019. doi: 10.1007/978-3-030-25540-4_25.

[22] Aaditya Prakash Chouhan and Gourinath Banda. Formal Verification of Heuristic Autonomous Intersection Management Using Statistical Model Checking. *Sensors*, 20(16), 2020. doi: 10.3390/s20164506.

[23] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. Towards Verified Artificial Intelligence, 2016.

[24] Alexander Bagnall and Gordon Stewart. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. *Proceedings AAAI (Conference on Artificial Intelligence)*, 33(01):2662–2669, 2019. doi: 10.1609/aaai.v33i01.33012662.

[25] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, 1996. doi: 10.1017/CBO9780511624162.

[26] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 1st edition, 2010. doi: 10.1017/CBO9781139195881.

[27] Michael Leuschel, Jens Bendisposto, Ivaylo Dobrikov, Sebastian Krings, and Daniel Plagge. From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In Jean-Louis Boulanger, editor, *Formal Methods Applied to Complex Systems: Implementation of the B Method*, chapter 14, pages 427–446. Wiley ISTE, 2014. doi: 10.1002/9781119002727.ch14.

[28] Michael Leuschel and Michael Butler. ProB: An Automated Analysis Toolset for the B Method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008. doi: 10.1007/s10009-007-0063-9.

[29] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *Proceedings FME (International Symposium of Formal Methods Europe)*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003. doi: 10.1007/978-3-540-45236-2_46.

[30] Mats Carlsson, Greger Ottosson, and Björn Carlson. An Open-Ended Finite Domain Constraint Solver. In *Proceedings PLILP (International Symposium on Programming Language Implementation and Logic Programming)*, pages 191–206. Springer, 1997. doi: 10.1007/BFb0033845.

[31] Thierry Lecomte. Applying a Formal Method in Industry: A 15-Year Trajectory. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, pages 26–34. Springer, 2009. doi: 10.1007/978-3-642-04570-7_3.

[32] Denis Sabatier. Using Formal Proof and B Method at System Level for Industrial Projects. In *Proceedings RSSRail (International Conference on Reliability, Safety,*

and Security of Railway Systems), pages 20–31. Springer, 2016. doi: 10.1007/978-3-319-33951-1_2.

[33] Michael Butler, Dana Dghaym, Tomas Fischer, Thai Son Hoang, Klaus Reichl, Colin Snook, and Peter Tummeltshammer. Formal Modelling Techniques for Efficient Development of Railway Control Products. In *International Conference on Reliability, Safety and Security of Railway Systems*, pages 71–86. Springer, 2017. doi: 10.1007/978-3-319-68499-4_5.

[34] Mathieu Comptier, David Déharbe, Julien Molinero Perez, Louis Mussat, Thibaut Pierre, and Denis Sabatier. Safety analysis of a CBTC system: a rigorous approach with Event-B. In *Proceedings RSSRail (International Conference on Reliability, Safety, and Security of Railway Systems)*, pages 148–159. Springer, 2017. doi: 10.1007/978-3-319-68499-4_10.

[35] Mathieu Comptier, Michael Leuschel, Luis-Fernando Mejia, Julien Molinero Perez, and Mareike Mutz. Property-based modelling and validation of a CBTC zone controller in Event-B. In *Proceedings RSSRail (International Conference on Reliability, Safety, and Security of Railway Systems)*, pages 202–212. Springer, 2019. doi: 10.1007/978-3-030-18744-6_13.

[36] Randolf Berglehner, Abdul Rasheeq, and Ibtihel Cherif. An Approach to Improve SysML Railway Specification Using UML-B and Event-B. *Poster presented at RSSRail*, 2019.

[37] Thierry Lecomte. Programming the CLEARSY Safety Platform with B. In Alexander Raschke, Dominique Méry, and Frank Houdek, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 124–138. Springer, 2020. doi: 10.1007/978-3-030-48077-6_9.

[38] CENELEC EN 50128:2011. CENELEC EN 50128: Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. Standard, European Committee for Electrotechnical Standardization, 2011.

[39] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002. doi: 10.1145/505145.505149.

[40] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis.* MIT Press, 2006.

[41] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[42] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[43] Sebastian Krings, Michael Leuschel, Joshua Schmidt, David Schneider, and Marc Frappier. Translating Alloy and Extensions to Classical B. *Science of Computer Programming*, 188, 2020. doi: 10.1016/j.scico.2019.102378.

[44] Sebastian Krings, Joshua Schmidt, Carola Brings, Marc Frappier, and Michael Leuschel. A Translation from Alloy to B. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 10817 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2018. doi: 10.1007/978-3-319-91271-4_6.

[45] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Proceedings TACAS (International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007. doi: 10.1007/978-3-540-71209-1_49.

[46] Daniel Plagge and Michael Leuschel. Validating B, Z and TLA$^+$ using ProB and Kodkod. In *Proceedings FM (International Symposium on Formal Methods)*, volume 7436 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2012.

[47] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings TACAS (International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[48] Sebastian Krings and Michael Leuschel. SMT Solvers for Validation of B and Event-B Models. In Erika Ábrahám and Marieke Huisman, editors, *Proceedings IFM (International Conference on Integrated Formal Methods)*, pages 361–375. Springer, 2016. doi: 10.1007/978-3-319-33693-0_23.

[49] Joshua Schmidt and Michael Leuschel. Improving SMT Solver Integrations for the Validation of B and Event-B Models. In Alberto Lluch Lafuente and Anastasia Mavridou, editors, *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 12863 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2021. doi: 10.1007/978-3-030-85248-1_7.

[50] Joshua Schmidt and Michael Leuschel. SMT Solving for the Validation of B and Event-B Models. *International Journal on Software Tools for Technology Transfer*, 24(6):1043–1077, 2022. doi: 10.1007/s10009-022-00682-y.

[51] George Boole. *The Mathematical Analysis of Logic.* Philosophical Library, 1847.

[52] John Venn. On the Diagrammatic and Mechanical Representation of Propositions and Reasonings. *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, 10(58):1–18, 1880. doi: 10.1080/14786448008626877.

[53] William Stanley Jevons, Robert Adamson, and Harriet A. Jevons. *Pure Logic and Other Minor Works.* ATLA monograph preservation program. Macmillan, 1890.

[54] George Edward Hughes and Max John Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996.

[55] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[56] Bernhard Pfahringer. *Disjunctive Normal Form*, pages 371–372. Springer, 2017. doi: 10.1007/978-1-4899-7687-1_223.

[57] Steven Prestwich. CNF Encodings. *Frontiers in Artificial Intelligence and Applications*, 185, 2009. doi: 10.3233/978-1-58603-929-5-75.

[58] Irving M. Copi, Carl Cohen, and Kenneth McMahon. *Introduction to Logic*. Taylor & Francis, 2016.

[59] Augustus De Morgan. On the Structure of the Syllogism and on the Application of the Theory of Probabilities to Questions of Argument and Authority. In Cambridge Philosophical Society, editor, *Transactions of the Cambridge Philosophical Society*, volume 8, pages 379 – 408. Cambridge University Press, 1849.

[60] Grigori Samuilowitsch Tseitin. On the Complexity of Derivation in Propositional Calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, 1983. doi: 10.1007/978-3-642-81955-1_28.

[61] Thierry Boy de la Tour. An Optimality Result for Clause form Translation. *Journal of Symbolic Computation*, 14(4):283–301, 1992. doi: 10.1016/0747-7171(92)90009-S.

[62] Elmar Eder. *Relative Complexities of First Order Calculi*. Artificial intelligence = Künstliche Intelligenz. Vieweg, 1992. doi: 10.1007/978-3-322-84222-0.

[63] David A. Plaisted and Steven Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. doi: 10.1016/S0747-7171(86)80028-1.

[64] Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On Generating Small Clause Normal Forms. In *Proceedings CADE (International Conference on Automated Deduction)*, volume 1421 of *Lecture Notes in Computer Science*, pages 397–411. Springer, 1998. doi: 10.1007/BFb0054274.

[65] Paul Jackson and Daniel Sheridan. Clause Form Conversions for Boolean Circuits. In *Proceedings SAT (International Conference on Theory and Applications of Satisfiability Testing)*, SAT'04, pages 183–198. Springer, 2004. doi: 10.1007/11527695_15.

[66] Thierry Boy De La Tour. *Optimisation par renommage dans la méthode de résolution.* Theses, Institut National Polytechnique de Grenoble - INPG, 1991.

[67] Elliott Mendelson. *Introduction to Mathematical Logic.* Discrete Mathematics and Its Applications. CRC Press, 6th edition, 2015.

[68] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 2nd edition, 1996.

[69] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3rd edition, 2010.

[70] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[71] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. doi: 10.2307/2371045.

[72] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[73] Kurt Gödel. *Über die Vollständigkeit des Logikkalküls.* PhD thesis, Universität Wien, 1929.

[74] Alain Colmerauer, Henry Kanoui, Robert Pasero, and Philippe Roussel. Un système de communication homme-machine en franc ais. Rapport pr eliminaire de fin de contrat IRIA. *Tech. rep., Faculte des Sciences de Luminy, Universite Aix-Marseille II*, 1973.

[75] Alain Colmerauer and Philippe Roussel. *The Birth of Prolog*, pages 331–367. Association for Computing Machinery, 1996. doi: 10.1145/155360.155362.

[76] ISO/IEC 13211-1. ISO/IEC 13211-1: Information technology – Programming languages – Prolog – Part 1: General core. Standard, International Organization for Standardization, 1995.

[77] ISO/IEC 13211-2. ISO/IEC 13211-2: Information technology – Programming languages – Prolog – Part 2: Modules. Standard, International Organization for Standardization, 2000.

[78] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. Fifty Years of Prolog and Beyond. *Theory and Practice of Logic Programming*, pages 1–83, 2022. doi: 10.1017/S1471068422000102.

[79] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!*, volume 7 of *Texts in Computing.* College Publications, 2006.

[80] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[81] Mats Carlsson and Per Mildner. SICStus Prolog - The First 25 Years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012. doi: 10.1017/S1471068411000482.

[82] Robert A. Kowalski and Donald Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2(3):227–260, 1971. doi: 10.1007/978-3-642-81955-1_32.

[83] Robert A. Kowalski. Predicate Logic as Programming Language. In Jack L. Rosenfeld, editor, *Proceedings IFIP (International Federation for Information Processing)*, pages 569–574. North-Holland, 1974.

[84] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 4th edition, 2011.

[85] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings STOC (Annual ACM SIGACT Symposium on Theory of Computing)*, STOC '71, pages 151–158. Association for Computing Machinery, 1971. doi: 10.1145/800157.805047.

[86] Leonid A. Levin. Universal Sequential Search Problems. *Problems of Information Transmission*, 9(3), 1973.

[87] João P. Marques Silva and Karem A. Sakallah. GRASP - a New Search Algorithm for Satisfiability. In *Proceedings ICCAD (International Conference on Computer-Aided Design)*, pages 220–227. IEEE Computer Society, 1997.

[88] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[89] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings DAC (Annual Design Automation Conference)*, pages 530–535. ACM, 2001. doi: 10.1145/378239.379017.

[90] Armin Biere. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Proceedings SAT (International Conference on Theory and Applications of Satisfiability Testing)*, pages 28–33. Springer, 2008. doi: 10.1007/978-3-540-79719-7_4.

[91] Gilles Audemard and Laurent Simon. Refining Restarts Strategies for SAT and UNSAT. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012. doi: 10.1007/978-3-642-33558-7_11.

[92] Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Proceedings SAT (International Conference on Theory and Applications of Satisfiability Testing)*, pages 294–299. Springer, 2007. doi: 10.1007/978-3-540-72788-0_28.

[93] Jan Johannsen. The Complexity of Pure Literal Elimination. *Journal of Automated Reasoning*, 35:89–95, 2005. doi: 10.1007/978-1-4020-5571-3_6.

[94] Randal E. Bryant, Steven German, and Miroslav N. Velev. Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic. *ACM Transactions on Computational Logic*, 2(1):93–134, 2001. doi: 10.1145/371282.371364.

[95] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Proceedings CAV (International Conference on Computer Aided Verification)*, pages 78–92. Springer, 2002. doi: 10.1007/3-540-45657-0_7.

[96] Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant. Deciding Separation Formulas with SAT. In *Proceedings CAV (International Conference on Computer Aided Verification)*, pages 209–222. Springer, 2002. doi: 10.1007/3-540-45657-0_16.

[97] Ofer Strichman. On Solving Presburger and Linear Arithmetic with SAT. In *Proceedings FMCAD (International Conference on Formal Methods in Computer-Aided Design)*, pages 160–170. Springer, 2002. doi: 10.1007/3-540-36126-X_10.

[98] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006. doi: 10.1145/1217856.1217859.

[99] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 36–50. Springer, 2005. doi: 10.1007/978-3-540-32275-7_3.

[100] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-825.

[101] Clark W. Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, 2018. doi: 10.1007/978-3-319-10575-8_11.

[102] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003. doi: 10.1017/CBO9780511615320.

[103] Kim Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998. doi: 10.7551/mitpress/5625.001.0001.

[104] Joxan Jaffar and Spiro Michaylov. Methodology and Implementation of a CLP System. In *Proceedings ICLP (International Conference on Logic Programming)*, pages 196–218. MIT Press, 1987.

[105] Kim Marriott, Peter J. Stuckey, and Mark Wallace. Constraint Logic Programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 409–452. Elsevier, 2006. doi: 10.1016/S1574-6526(06)80016-7.

[106] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*, 1978. Springer. doi: 10.1007/3-540-08766-4.

[107] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[108] Edmund M Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999. doi: 10.1002/9780470050118.ecse247.

[109] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977. doi: 10.1109/TSE.1977.229904.

[110] Kenneth L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon, 1992. CMU Tech Rpt. CMU-CS-92-131.

[111] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L.J. Hwang. Symbolic Model Checking: 1020 States and Beyond. *Information and Computation*, 98(2):142–170, 1992. doi: 10.1016/0890-5401(92)90017-A.

[112] Kenneth L. McMillan. *Symbolic Model Checking*, pages 25–60. Springer, 1993. doi: 10.1007/978-1-4615-3190-6_3.

[113] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer, 1999.

[114] Sebastian Krings. *Towards Infinite-State Symbolic Model Checking for B and Event-B*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, Germany, 2017.

[115] Armin Biere. Bounded Model Checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 739–764. IOS Press, 2021. doi: 10.3233/FAIA201002.

[116] Yakir Vizel and Arie Gurfinkel. Interpolating Property Directed Reachability. In *Proceedings CAV (International Conference on Computer Aided Verification)*, pages 260–276. Springer, 2014. doi: 10.1007/978-3-319-08867-9_17.

[117] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In Warren A. Hunt and Steven D. Johnson, editors, *Proceedings FMCAD (International Conference on Formal Methods in Computer-Aided Design)*, pages 127–144. Springer, 2000. doi: 10.1007/3-540-40922-X_8.

[118] Alessandro Cimatti and Alberto Griggio. Software Model Checking via IC3. In P. Madhusudan and Sanjit A. Seshia, editors, *Proceedings CAV (International Conference on Computer Aided Verification)*, pages 277–293. Springer, 2012. doi: 10.1007/978-3-642-31424-7_23.

[119] Adolf Fraenkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86(3):230–237, 1922. doi: 10.1007/BF01457986.

[120] Abraham Adolf Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of Set Theory*, volume 67. Elsevier, 1973.

[121] ClearSy. *Atelier B, User and Reference Manuals*, 2009. Available at `http://www.atelierb.eu/`.

[122] Richard Bonichon, David Déharbe, Thierry Lecomte, and Valerio Medeiros Jr. LLVM-based code generation for B. In *Proceedings SBMF (Brazilian Symposium on Formal Methods)*, volume 8941 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014. doi: 10.1007/978-3-319-15075-8_1.

[123] Jean-Christophe Voisinet. JBTools: an experimental platform for the formal B method. *Proceedings PPPJ (International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools)*, pages 137–139, 2002.

[124] Dominique Méry and Neeraj Kumar Singh. Automatic Code Generation from Event-B Models. In *Proceedings SoICT (International Symposium on Information and Communication Technology)*, pages 179–188. ACM ICPS, 2011. doi: 10.1145/2069216.2069252.

[125] Dominique Méry and Neeraj Kumar Singh. A Generic Framework: From Modeling to Code. *Innovations in Systems and Software Engineering*, 7(4):227–235, 2011. doi: 10.1007/s11334-011-0165-0.

[126] Fabian Vu, Dominik Hansen, Philipp Körner, and Michael Leuschel. A Multi-target Code Generator for High-Level B. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 11918 of *Lecture Notes in Computer Science*, pages 456–473. Springer, 2019. doi: 10.1007/978-3-030-34968-4_25.

[127] Michael Leuschel. Spot the Difference: A Detailed Comparison Between B and Event-B. In Alexander Raschke, Elvinia Riccobene, and Klaus-Dieter Schewe, editors, *Logic, Computation and Rigorous Methods - Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday*, volume 12750 of *Lecture Notes in Computer Science*, pages 147–172. Springer, 2021. doi: 10.1007/978-3-030-76020-5_9.

[128] Dominik Hansen, David Schneider, and Michael Leuschel. Using B and ProB for Data Validation Projects. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 167–182. Springer, 2016. doi: 10.1007/978-3-319-33600-8_10.

[129] David Schneider, Michael Leuschel, and Tobias Witt. Model-Based Problem Solving for University Timetable Validation and Improvement. *Formal Aspects of Computing*, 30(5):545–569, 2018. doi: 10.1007/s00165-018-0461-7.

[130] David Schneider. *Constraint Modelling and Data Validation Using Formal Specification Languages*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, Germany, 2017.

[131] ISO/IEC 13568. Information technology - Z formal specification notation - Syntax, type system and semantics. Standard, International Organization for Standardization, 2002.

[132] Dominik Hansen and Michael Leuschel. Translating TLA+ to B for Validation with ProB. In *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 7321 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2012. doi: 10.1007/978-3-642-30729-4_3.

[133] Dominik Hansen and Michael Leuschel. Translating B to TLA$^+$ for Validation with TLC. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 8477 of *Lecture Notes in Computer Science*, pages 40–55, 2014.

[134] Ivaylo Dobrikov and Michael Leuschel. Optimising the ProB Model Checker for B Using Partial Order Reduction. *Formal Aspects of Computing*, 28(2):295–323, 2016. doi: 10.1007/s00165-015-0351-1.

[135] Philipp Körner and Michael Leuschel. Towards Practical Partial Order Reduction for High-Level Formalisms. In Akash Lal and Stefano Tonetta, editors, *Proceedings VSTTE (Working Conference on Verified Software: Theories, Tools, and Experiments)*, volume 13800 of *Lecture Notes in Computer Science*, pages 72–91. Springer, 2023. doi: 10.1007/978-3-031-25803-9_5.

[136] Michael Leuschel. Operation Caching and State Compression for Model Checking of High-Level Models - How to Have Your Cake and Eat It. In Maurice H. ter Beek and Rosemary Monahan, editors, *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 13274 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2022. doi: 10.1007/978-3-031-07727-2_8.

[137] Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *The Journal of Logic Programming*, 37(1):95–138, 1998. doi: 10.1016/S0743-1066(98)10005-5.

[138] Christian Holzbaur and Thom W. Frühwirth. Compiling Constraint Handling Rules into Prolog with Attributed Variables. In *Proceedings PPDP (International Symposium on Principles and Practice of Declarative Programming)*, 1999. doi: 10.1007/10704567_7.

[139] Tom Schrijvers, Jan Wielemaker, and Bart Demoen. Constraint Handling Rules for SWI-Prolog. In *Proceedings WLP (Workshop on (Constraint) Logic Programming)*. Universität Ulm, Germany, 2005.

[140] Sebastian Krings, Joshua Schmidt, Patrick Skowronek, Jannik Dunkelau, and Dierk Ehmke. Towards Constraint Logic Programming over Strings for Test Data Generation. In *Proceedings WLP (Workshop on (Constraint) Logic Programming)*, volume 12057 of *Lecture Notes in Computer Science*, pages 139–159. Springer, 2020. doi: 10.1007/978-3-030-46714-2_10.

[141] Carola Brings. Translating Alloy to B. Bachelor thesis, Heinrich-Heine-Universität Düsseldorf, Germany, 2017.

[142] Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors. *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings*, volume 10817 of *Lecture Notes in Computer Science*, 2018. Springer. doi: 10.1007/978-3-319-91271-4.

[143] Alberto Lluch-Lafuente and Anastasia Mavridou, editors. *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings*, volume 12863 of *Lecture Notes in Computer Science*, 2021. Springer. doi: 10.1007/978-3-030-85248-1.

[144] Joshua Schmidt, Sebastian Krings, and Michael Leuschel. Interactive Model Repair by Synthesis. In *Proceedings ABZ (International Conference on Rigorous State*

Based Methods), volume 9675 of *Lecture Notes in Computer Science*. Springer, 2016. doi: 10.1007/978-3-319-33600-8_25.

[145] Joshua Schmidt, Sebastian Krings, and Michael Leuschel. Repair and Generation of Formal Models Using Synthesis. In *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 11023 of *Lecture Notes in Computer Science*. Springer, 2018. doi: 10.1007/978-3-319-98938-9_20.

[146] Alexandros Efremidis, Joshua Schmidt, Sebastian Krings, and Philipp Körner. Measuring Coverage of Prolog Programs Using Mutation Testing. In Josep Silva, editor, *Proceedings WFLP (International Workshop on Functional and (Constraint) Logic Programming)*, volume 11285 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2018. doi: 10.1007/978-3-030-16202-3_3.

[147] Jannik Dunkelau, Sebastian Krings, and Joshua Schmidt. Automated Backend Selection for ProB Using Deep Learning. In Julia M. Badger and Kristin Yvonne Rozier, editors, *Proceedings NFM (International Symposium on NASA Formal Methods)*, pages 130–147. Springer, 2019. doi: 10.1007/978-3-030-20652-9_9.

[148] Sebastian Krings, Philipp Körner, and Joshua Schmidt. Experience Report on an Inquiry-Based Course on Model Checking. In Veronika Thurner, Oliver Radfelder, and Karin Vosseberg, editors, *Tagungsband des 16. Workshops "Software Engineering im Unterricht der Hochschulen"*, volume 2358 of *Proceedings CEUR*, pages 87–98. CEUR-WS.org, 2019.

[149] Jannik Dunkelau, Joshua Schmidt, and Michael Leuschel. Analysing ProB's Constraint Solving Backends. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*. Springer, 2020. doi: 10.1007/978-3-030-48077-6_8.

[150] Sebastian Krings and Michael Leuschel. Constraint Logic Programming over Infinite Domains with an Application to Proof. In *Proceedings WLP (Workshop on (Constraint) Logic Programming)*, volume 234 of *EPTCS*. Electronic Proceedings in Theoretical Computer Science, 2016. doi: 10.4204/EPTCS.234.6.

[151] André Sülflow, Ulrich Kühne, Robert Wille, Daniel Große, and Rolf Drechsler. Evaluation of SAT-like Proof Techniques for Formal Verification of Word-Level Circuits. In *Proceedings WRTLT (IEEE Workshop on RTL and High Level Testing)*. IEEE Computer Society Press, 2007.

[152] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. In *Proceedings ISSTA (International Symposium on Software Testing and Analysis)*, ISSTA '96, pages 239–249. ACM, 1996. doi: 10.1145/226295.226322.

[153] Sebastian Krings and Michael Leuschel. Proof Assisted Bounded and Unbounded Symbolic Model Checking of Software and System Models. *Science of Computer Programming*, 158:41–63, 2018. doi: 10.1016/j.scico.2017.08.013.

[154] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. *Formal Methods in System Design*, 2017. doi: 10.1007/s10703-016-0267-2.

[155] Raymond T. Boute. The Euclidean Definition of the Functions Div and Mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, 1992. doi: 10.1145/128861.128862.

[156] Sebastian Krings, Michael Leuschel, Philipp Körner, Stefan Hallerstede, and Miran Hasanagic. Three Is a Crowd: SAT, SMT and CLP on a Chessboard. In Francesco Calimeri, Kevin W. Hamlen, and Nicola Leone, editors, *Practical Aspects of Declarative Languages - 20th International Symposium, PADL 2018, Los Angeles, CA, USA, January 8-9, 2018, Proceedings*, volume 10702 of *Lecture Notes in Computer Science*, pages 63–79. Springer, 2018. doi: 10.1007/978-3-319-73305-0_5.

[157] Larry Wos, Ross Overbeck, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice Hall, 1984.

[158] Michael Leuschel and David Schneider. Towards B as a High-Level Constraint Modelling Language. In Yamine Ait Ameur and Klaus-Dieter Schewe, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 8477 of *Lecture Notes in Computer Science*, pages 101–116. Springer, 2014. doi: 10.1007/978-3-662-43652-3_8.

[159] Aleksandar Milicevic and Daniel Jackson. Preventing Arithmetic Overflows in Alloy. *Science of Computer Programming*, 94:203–216, 2014. doi: 10.1007/978-3-642-30885-7_8.

[160] Emina Torlak, Mana Taghdiri, Greg Dennis, and Joseph P. Near. Applications and Extensions of Alloy: Past, Present and Future. *Mathematical Structures in Computer Science*, 23(4):915–933, 2013. doi: 10.1017/S0960129512000291.

[161] Aboubakr Achraf El Ghazi and Mana Taghdiri. Analyzing Alloy Formulas using an SMT Solver: A Case Study. *Computing Research Repository*, abs/1505.00672, 2015.

[162] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark W. Barrett. Relational Constraint Solving in SMT. In *Proceedings CADE (International Conference on Automated Deduction)*, volume 10395 of *Lecture Notes in Computer Science*, pages 148–165, 2017. doi: 10.1007/978-3-319-63046-5_10.

[163] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. A Proof Assistant for Alloy Specifications. In *Proceedings TACAS (International*

Conference on Tools and Algorithms for the Construction and Analysis of Systems), volume 7214 of *Lecture Notes in Computer Science*, pages 422–436, 2012. doi: 10.1007/978-3-642-28756-5_29.

[164] Leonid Mikhailov and Michael J. Butler. An Approach to Combining B and Alloy. In *Proceedings ZB (International Conference of B and Z Users)*, volume 2272 of *Lecture Notes in Computer Science*, pages 140–161. Springer, 2002. doi: 10.1007/3-540-45648-1_8.

[165] Paulo J. Matos and João Marques-Silva. Model Checking Event-B by Encoding into Alloy. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 5238 of *Lecture Notes in Computer Science*, page 346, 2008. doi: 10.1007/978-3-540-87603-8_34.

[166] Petra Malik, Lindsay Groves, and Clare Lenihan. Translating Z to Alloy. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 5977 of *Lecture Notes in Computer Science*, pages 377–390, 2010. doi: 10.1007/978-3-642-11811-1_28.

[167] Daniel Plagge and Michael Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer, 2007. doi: 10.1007/978-3-540-73210-5_25.

[168] Nuno Macedo and Alcino Cunha. Alloy Meets TLA+: An Exploratory Study. *Computing Research Repository*, abs/1603.03599, 2016.

[169] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *Proceedings FSE (Symposium on the Foundations of Software Engineering)*, FSE 2016, pages 373–383. ACM, 2016. doi: 10.1145/2950290.2950318.

[170] Joseph P. Near and Daniel Jackson. An Imperative Extension to Alloy. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 5977 of *Lecture Notes in Computer Science*, pages 118–131, 2010. doi: 10.1007/978-3-642-11811-1_10.

[171] Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, and Nazareno Aguirre. DynAlloy: Upgrading Alloy with Actions. In *Proceedings ICSE (International Conference on Software Engineering)*, pages 442–451, 2005. doi: 10.1145/1062455.1062535.

[172] Marcelo F. Frias, Carlos López Pombo, Juan P. Galeotti, and Nazareno Aguirre. Efficient Analysis of DynAlloy Specifications. *ACM Transactions on Software Engineering and Methodology*, 17(1):4:1–4:34, 2007. doi: 10.1145/1314493.1314497.

[173] Alcino Cunha. Bounded Model Checking of Temporal Formulas with Alloy. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, volume 8477 of *Lecture Notes in Computer Science*, pages 303–308, 2014. doi: 10.1007/978-3-662-43652-3_29.

[174] Antoine Requet. BART: A Tool for Automatic Refinement. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 345–345. Springer, 2008. doi: 10.1007/978-3-540-87603-8_33.

[175] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, and Alexander Romanovsky. Patterns for Refinement Automation. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Proceedings FMCO (International Symposium on Formal Methods for Components and Objects)*, pages 70–88. Springer, 2010. doi: 10.1007/978-3-642-17071-3_4.

[176] Peter Kriens, Burkhardt Renz, David Karapetyan, and Daniel Jackson. Alloy Models. `https://github.com/AlloyTools/models`, 2022. Licensed under the Apache License 2.0. [Online; accessed 17-November-2022].

[177] Joshua Schmidt. Wrong unsat when using lambda functions. `https://github.com/Z3Prover/z3/issues/6748`, 2023. Github issue. [Online; accessed 9-Jun-2023].

[178] Chris Newcombe. Why Amazon Chose TLA + . In Yamine Ait Ameur and Klaus-Dieter Schewe, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 25–39. Springer, 2014.

[179] Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. *The Electrum Analyzer: Model Checking Relational First-Order Temporal Specifications*, pages 884–887. Association for Computing Machinery, 2018. doi: 10.1145/3238147.3240475.

[180] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings CAV (International Conference on Computer Aided Verification)*, pages 359–364. Springer, 2002.

[181] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In Armin Biere and Roderick Bloem, editors, *Proceedings CAV (International Conference on Computer Aided Verification)*, pages 334–342. Springer, 2014.

[182] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings SMT (International Workshop on Satisfiability Modulo Theories)*, 2010.

[183] Leonardo de Moura and Nikolaj Bjørner. Generalized, Efficient Array Decision Procedures. In *Proceedings FMCAD (International Conference on Formal Methods in Computer-Aided Design)*, pages 45–52, 2009. doi: 10.1109/FMCAD.2009.5351142.

[184] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://SMT-LIB.org`, 2016. [Online; accessed 19-Sep-2023].

[185] Jean-Raymond Abrial and Louis Mussat. On Using Conditional Definitions in Formal Theories. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *Proceedings ZB (International Conference of B and Z Users)*, Lecture Notes in Computer Science 2272, pages 242–269. Springer, 2002. doi: 10.1007/3-540-45648-1_13.

[186] Basile Clement. Invalid unsat with recursive functions and arrays. `https://github.com/Z3Prover/z3/issues/5813`, 2022. Github issue. [Online; accessed 2-Dec-2022].

[187] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting Symmetry in SMT Problems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proceedings CADE (International Conference on Automated Deduction)*, pages 222–236. Springer, 2011. doi: 10.1007/978-3-642-22438-6_18.

[188] Carlos Areces, David Déharbe, Pascal Fontaine, and Orbe Ezequiel. SyMT: Finding Symmetries in SMT Formulas. In *Proceedings SMT (International Workshop on Satisfiability Modulo Theories)*, 2013.

[189] Eugene M. Luks. Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time. *Computer and System Sciences*, 25(1):42–65, 1982. doi: 10.1016/0022-0000(82)90009-5.

[190] Tommi Junttila and Petteri Kaski. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In *Proceedings ALENEX (Symposium on Algorithm Engineering and Experiments)*, pages 135–149. SIAM, 2007. doi: 10.1137/1.9781611972870.13.

[191] Michael Leuschel. Fast and Effective Well-Definedness Checking. In Brijesh Dongol and Elena Troubitsyna, editors, *Proceedings IFM (International Conference on Integrated Formal Methods)*, pages 63–81. Springer, 2020. doi: 10.1007/978-3-030-63461-2_4.

[192] Jacob M. Howe and Andy King. A Pearl on SAT Solving in Prolog. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, pages 165–174. Springer, 2010. doi: 10.1007/978-3-642-12251-4_13.

[193] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings ICCAD (International Conference on Computer-Aided Design)*, pages 279–285. IEEE Press, 2001.

[194] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings IJCAI (International Joint Conference on Artificial Intelligence)*, pages 399–404. Morgan Kaufmann Publishers Inc., 2009.

[195] Armin Biere and Andreas Fröhlich. Evaluating CDCL Variable Scoring Schemes. In Marijn Heule and Sean Weaver, editors, *Proceedings SAT (International Conference on Theory and Applications of Satisfiability Testing)*, pages 405–422. Springer, 2015. doi: 10.1007/978-3-319-24318-4_29.

[196] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In Andrei Voronkov, editor, *Proceedings CADE (International Conference on Automated Deduction)*, pages 195–210. Springer, 2002. doi: 10.1007/3-540-45620-1_17.

[197] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-Based Procedures for Temporal Reasoning. In Susanne Biundo and Maria Fox, editors, *Recent Advances in AI Planning*, pages 97–108. Springer, 2000.

[198] Jesper Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference Decision Diagrams. In Jörg Flum and Mario Rodriguez-Artalejo, editors, *Computer Science Logic*, pages 111–125. Springer, 1999. doi: 10.1007/3-540-48168-0_9.

[199] Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock Difference Diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999. doi: 10.7146/brics.v5i46.19491.

[200] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16: 87–90, 1958. doi: 10.1090/qam/102435.

[201] Lester Randolph Ford. *Network Flow Theory*. Number 923 in Network Flow Theory. Rand Corporation, 1956.

[202] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA^{+} Model Checking Made Symbolic. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019. doi: 10.1145/3360549.

[203] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT Competition 2015-2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):221–259, 2019. doi: 10.3233/SAT190123.

[204] Atif Mashkoor. The Hemodialysis Machine Case Study. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 329–343. Springer, 2016. doi: 10.1007/978-3-319-33600-8_29.

[205] Thai Son Hoang, Colin Snook, Lukas Ladenberger, and Michael Butler. Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-Simulation. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 360–375. Springer, 2016. doi: 10.1007/978-3-319-33600-8_31.

[206] Frédéric Boniol and Virginie Wiels. The Landing Gear System Case Study. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, pages 1–18. Springer, 2014. doi: 10.1007/978-3-319-07512-9_1.

[207] Dominik Hansen, Lukas Ladenberger, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. Validation of the ABZ Landing Gear System Using ProB. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, pages 66–79. Springer, 2014. doi: 10.1007/s10009-015-0395-9.

[208] David Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 217–230. Springer, 2010. doi: 10.1007/978-3-642-11811-1_17.

[209] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. SMT Solvers for Rodin. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 194–207. Springer, 2012. doi: 10.1007/978-3-642-30885-7_14.

[210] David Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 78(3):310–326, 2013. doi: 10.1016/j.scico.2011.03.007. Abstract State Machines, Alloy, B and Z - Selected Papers from ABZ 2010.

[211] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In *Proceedings CHARME (Advanced Research Working Conference on Correct Hardware Design and Verification Methods)*, pages 54–66, 1999. doi: 10.1007/3-540-48153-2_6.

[212] Ewan Davidson, Özgür Akgün, Joan Espasa, and Peter Nightingale. Effective Encodings of Constraint Programming Models to SMT. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, pages 143–159. Springer, 2020. doi: 10.1007/978-3-030-58475-7_9.

[213] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational Reasoning via SMT Solving. In Michael Butler and Wolfram Schulte, editors, *Proceedings FM (International Symposium on Formal Methods)*, pages 133–148. Springer, 2011. doi: 10.1007/978-3-642-21437-0_12.

[214] Ali Abbassi, Nancy A. Day, and Derek Rayside. Astra Version 1.0: Evaluating Translations from Alloy to SMT-LIB. *Computing Research Repository*, abs/1906.05881, 2019.

[215] Tjark Weber. SMT Solvers: New Oracles for the HOL Theorem Prover. *International Journal on Software Tools for Technology Transfer*, 13(5):419–429, 2011. doi: 10.1007/s10009-011-0188-8.

[216] Hadrien Bride, Olga Kouchnarenko, Fabien Peureux, and Guillaume Voiron. Workflow Nets Verification: SMT or CLP? In Maurice H. ter Beek, Stefania Gnesi, and Alexander Knapp, editors, *Proceedings FMICS-AVoCS (International Workshop on Formal Methods for Industrial Critical Systems and Automated Verification of Critical Systems)*, pages 39–55. Springer, 2016. doi: 10.1007/978-3-319-45943-1_3.

[217] Makai Mann, Amalee Wilson, Cesare Tinelli, and Clark W. Barrett. Smt-Switch: a solver-agnostic C++ API for SMT Solving. *Computing Research Repository*, abs/2007.01374, 2020. doi: 10.1007/978-3-030-80223-3_26.

[218] Thierry Boy de la Tour. An Optimality Result for Clause Form Translation. *Journal of Symbolic Computation*, 14(4):283–301, 1992. doi: 10.1016/0747-7171(92) 90009-S.

[219] Dejan Jovanović, Clark Barrett, and Leonardo de Moura. The Design and Implementation of the Model Constructing Satisfiability Calculus. In *Proceedings FMCAD (International Conference on Formal Methods in Computer-Aided Design)*, pages 173–180. FMCAD Inc., 2013. doi: 10.1109/FMCAD.2013.7027033. Portland, Oregon.

[220] Andrew Healy, Rosemary Monahan, and James F. Power. Predicting SMT Solver Performance for Software Verification. In Catherine Dubois, Paolo Masci, and Dominique Méry, editors, *Proceedings F-IDE (Workshop on Formal Integrated*

Development Environment), volume 240 of *EPTCS*, pages 20–37, 2016. doi: 10.4204/EPTCS.240.2.

[221] Joshua Schmidt. Internalization of exists is not supported. `https://github.com/Z3Prover/z3/issues/6330`, 2022. Github issue. [Online; accessed 28-December-2022].

[222] Joshua Schmidt. Z3 consumes a lot of memory until the process gets killed. `https://github.com/Z3Prover/z3/issues/5571`, 2022. Github issue. [Online; accessed 29-December-2022].

[223] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Notices*, 42(6):89–100, 2007. doi: 10.1145/1273442.1250746.

[224] Gennadii Saltyshchak. Memory leak in Z3 Context using .NET API. `https://github.com/Z3Prover/z3/issues/5043/`, 2021. Github issue. [Online; accessed 2-Dec-2022].

[225] Pieter Hintjens. 0MQ - The Guide. `http://zguide.zeromq.org/page:all`, 2011.

[226] Michael Leuschel. Fast and Effective Well-Definedness Checking. In *Proceedings IFM (International Conference on Integrated Formal Methods)*, pages 63–81, 2020. doi: 10.1007/978-3-030-63461-2_4.

[227] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Proceedings CADE (International Conference on Automated Deduction)*, pages 183–198. Springer, 2007.

[228] Rosalie Defourné. Encoding TLA+ Proof Obligations Safely For SMT. In *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, page 88–106. Springer, 2023. doi: 10.1007/978-3-031-33163-3_7.

[229] Philipp Körner, Michael Leuschel, and Jannik Dunkelau. Towards a Shared Specification Repository. In Alexander Raschke, Dominique Méry, and Frank Houdek, editors, *Proceedings ABZ (International Conference on Rigorous State Based Methods)*, pages 266–271. Springer, 2020. doi: 10.1007/978-3-030-48077-6_22.

[230] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931, sep 2000. doi: 10.1145/365151.365169.

[231] Joshua Schmidt. Fuzzing von Prolog Programmen. Bachelor thesis, Heinrich-Heine-Universität Düsseldorf, Germany, 2015.

[232] Tariq, Khadija. Linking Alloy with SMT-based Finite Model Finding. Master's thesis, University of Waterloo, 2021.

[233] Jens Bendisposto, David Geleßus, Yumiko Jansing, Michael Leuschel, Antonia Pütz, Fabian Vu, and Michelle Werth. ProB2-UI: A Java-Based User Interface for ProB. In Alberto Lluch Lafuente and Anastasia Mavridou, editors, *Proceedings FMICS (Formal Methods for Industrial Critical Systems)*, pages 193–201. Springer, 2021.

[234] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994. doi: 10.1093/logcom/4.3.217.

[235] G. Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022. doi: 10.1093/jigpal/jzac068.

[236] Zsolt Zombori, Josef Urban, and Chad E. Brown. Prolog Technology Reinforcement Learning Prover. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 489–507. Springer, 2020. doi: 10.1007/978-3-030-51054-1_33.

[237] Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. SeaPearl: A Constraint Programming Solver Guided by Reinforcement Learning. In Peter J. Stuckey, editor, *Proceedings CPAIOR (International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research)*, pages 392–409. Springer, 2021. doi: 10.1007/978-3-030-78230-6_25.

[238] Jan Jakubuv and Josef Urban. Hammering Mizar by Learning Clause Guidance (Short Paper). In John Harrison, John O'Leary, and Andrew Tolmach, editors, *Proceedings ITP (International Conference on Interactive Theorem Proving)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:8. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi: 10.4230/LIPIcs.ITP.2019.34.