On Executing State-Based Specifications and Partial Order Reduction for High-Level Formalisms

Inaugural-Dissertation

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

vorgelegt von

Philipp Körner

aus Herne

Düsseldorf, Januar 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. Dr. Akram Idani
   Université Grenoble Alpes

Tag der mündlichen Prüfung: 14. Oktober 2022

Parts of this thesis have been published in the following peer-reviewed articles, conference proceedings and book chapters

- Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. Embedding High-Level Formal Specifications into Applications. In *Proceedings FM (International Symposium on Formal Methods)*, volume 11800 of *Lecture Notes in Computer Science*, pages 519–535. Springer, 2019

- Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. Integrating formal specifications into applications: the ProB Java API. *Formal Methods in System Design*, 57:160–187, 2020

- Sebastian Krings, Philipp Körner, Jannik Dunkelau, and Chris Rutenkolk. A Verified Low-Level Implementation of the Adaptive Exterior Light and Speed Control System. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 382–397. Springer, 2020

- Philipp Körner and Florian Mager. An Embedding of B in Clojure. In *Companion Proceedings MODELS (International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings)*, page 598–606. ACM, 2022

- Philipp Körner, Michael Leuschel, and Jannik Dunkelau. Towards a Shared Specification Repository. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2020

- Philipp Körner and Michael Leuschel. Towards Practical Partial Order Reduction for High-Level Formalisms. In *Proceedings VSTTE (International Conference on Verified Software: Theories, Tools, and Experiments) 2022*, volume 13800 of *Lecture Notes in Computer Science*. Springer, 2023. To appear.

Other peer-reviewed publications

Articles on Modelling and Verification

- Michael Butler, Philipp Körner, Sebastian Krings, Thierry Lecomte, Michael Leuschel, Luis-Fernando Mejia, and Laurent Voisin. The First Twenty-Five Years of Industrial Use of the B-Method. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 12327 of *Lecture Notes in Computer Science*, pages 189–209. Springer, 2020

- Fabian Vu, Dominik Hansen, Philipp Körner, and Michael Leuschel. A Multi-Target Code Generator for High-Level B. In *Proceedings iFM (International Conference on integrated Formal Methods)*, volume 11918 of *Lecture Notes in Computer Science*, pages 456–473. Springer, 2019

- Philipp Körner and Jens Bendisposto. Distributed Model Checking Using ProB. In *Proceedings NFM (NASA Formal Methods Symposium)*, volume 10811 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2018

- Jens Bendisposto, Philipp Körner, Michael Leuschel, Jeroen Meijer, Jaco van de Pol, Helen Treharne, and Jorden Whitefield. Symbolic Reachability Analysis of B through ProB and LTSmin. In *Proceedings iFM (International Conference on integrated Formal Methods)*, volume 9681 of *Lecture Notes in Computer Science*, pages 275–291. Springer, 2016

- Philipp Körner, Jeroen Meijer, and Michael Leuschel. State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin. In *Proceedings iFM (International Conference on integrated Formal Methods)*, volume 11023 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2018

- 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 *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z)*, volume 10817 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2018

- Dominik Hansen, Michael Leuschel, Philipp Körner, Sebastian Krings, Thomas Naulin, Nader Nayeri, David Schneider, and Frank Skowron. Validation and real-life demonstration of ETCS hybrid level 3 principles using a formal B model. *Software Tools for Technology Transfer*, 22, 2020

Articles on Teaching Formal Methods

- Antonio Cerone, Markus Roggenbach, James Davenport, Casey Denner, Marie Farrell, Magne Haveraaen, Faron Moller, Philipp Körner, Sebastian Krings, Peter Olveczky, Bernd-Holger Schlingloff, Nikolay Shilov, and Rustam Zhumagambetov. Rooting Formal Methods within Higher Education Curricula for Computer Science and Software Engineering – A White Paper. In *Proceedings FMFun (International Workshop on Formal Methods - Fun for Everybody) 2019*, volume 1301 of *CCIS*. Springer, 2021

- Sebastian Krings and Philipp Körner. Prototyping Games Using Formal Methods. In *Proceedings FMFun (International Workshop on Formal Methods - Fun for Everybody) 2019*, volume 1301 of *CCIS*. Springer, 2021

- Philipp Körner and Sebastian Krings. Increasing Student Self-Reliance and Engagement in Model-Checking Courses. In *Proceedings FMTea (Formal Methods Teaching)*, pages 60–74. Springer, 2021

Articles on Logic Programming

- 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

- Philipp Körner and Sebastian Krings. plspec - A Specification Language for Prolog Data. In *Proceedings Declare (International Workshop on Functional and Constraint Logic Programming) 2017*, volume 10997 of *LNAI*. Springer, 2018

- Isabel Wingen and Philipp Körner. Effectiveness of Annotation-Based Static Type Inference. In *Proceedings WFLP (International Workshop on Functional and Constraint Logic Programming) 2020*, volume 12560 of *Lecture Notes in Computer Science*, pages 74–93. Springer, 2021

- Philipp Körner, David Schneider, and Michael Leuschel. On the Performance of Bytecode Interpreters in Prolog. In *Proceedings WFLP (International Workshop on Functional and Constraint Logic Programming) 2020*, volume 12560 of *Lecture Notes in Computer Science*. Springer, 2021

- Falco Nogatz, Philipp Körner, and Sebastian Krings. Prolog Coding Guidelines: Status and Tool Support. In *Proceedings ICLP (International Conference on Logic Programming) (Technical Communications)*, volume 306 of *EPTCS*, 2019

- Alexandros Efremidis, Joshua Schmidt, Sebastian Krings, and Philipp Körner. Measuring Coverage of Prolog Programs Using Mutation Testing. In *Proceedings WFLP (International Workshop on Functional and Constraint Logic Programming) 2018*, volume 11285 of *Lecture Notes in Computer Science*. Springer, 2019

- Sebastian Krings, Michael Leuschel, Philipp Körner, Stefan Hallerstede, and Miran Hasanagić. Three Is a Crowd: SAT, SMT and CLP on a Chessboard. In *Proceedings PADL (International Symposium on Practical Aspects of Declarative Languages)*, volume 10702 of *Lecture Notes in Computer Science*. Springer, 2018

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 11. Juli 2022

Philipp Körner

When things go wrong, simplify.

Dan John

Abstract

This thesis is a selection of my co-authored manuscripts on state-based formal methods tools and applications. A focus lies on the B Method and the animator, constraint solver and model checker ProB.

The *first part* explores the opportunities that stem from executing state-based specifications. Three approaches are investigated:

Firstly, *embedding the tool* ProB into Java programs and interacting with it using a high-level API that exposes animation, constraint solving and model checking techniques: This technique enables a variety of applications, and has been successfully utilised in a timetable planning tool and a demonstrator in the railway domain.

Secondly, *treating imperative code* as a specification and attempting verification using the model checker CBMC: While technically feasible, verification *after* implementation comes with a variety of pitfalls and fixing located errors is quite cumbersome at this stage.

Finally, *embedding the B language* into Clojure in order to programmatically generate (parts of) and solve constraints or animate and model check constructed B machines: this approach treats specifications as plain data. Following the ideas of Lisp, this enables tools that analyse and transform specifications as well as the creation of domain-specific languages (DSLs).

The *second part* of this thesis re-visits an implementation of a state space reduction technique, partial order reduction (POR), in ProB. Anecdotally, we had little success with exploiting POR techniques for real-world models. Using a large collection of B machines, we put numbers to our impressions and find that, indeed, in the vast majority of cases, POR does not yield any reduction.

Motivated by a grand challenge we set ourselves, — a model of an interlocking system that *should* be susceptible to POR techniques, yet does not exhibit any reduction — we identify two idioms that hinder POR for higher-level specifications. The first idiom, usage of parameterised operations, often can be eliminated by unrolling a single operation into many, one for each possible parameter value. The second idiom, usage of high-level data structures such as sets or functions, often can be addressed by replacing sets with a bitvector encoding, or using constraint solvers to determine independence of operations.

Zusammenfassung

Diese Arbeit enthält eine Auswahl an Manuskripten, an denen ich mitgewirkt habe, zum Thema Zustands-basierter formaler Methoden, insbesondere Werkzeuge und Anwendungen. Der Fokus liegt dabei auf der B-Methode und dem Animator, Constraint Solver und Model Checker ProB.

Der *erste Teil* befasst sich mit den Möglichkeiten, die sich daraus ergeben Zustands-basierte Spezifikationen auszuführen. Drei Voransgehensweisen werden untersucht:

Erstens, *das Werkzeug* ProB in Java Programmen *einzubinden* und damit über dessen Schnittstelle, die Animations-, Constraint Solvings- und Model Checking-Techniken bereitstellt, zu interagieren. Dieses Vorgehen ermöglicht eine Vielfalt an Anwendungen und wurde erfolgreich in einem Werkzeug zur Erstellung von Stundenplänen und einen Demonstrator im Schienenverkehr eingesetzt.

Zweitens, *imperativen Code* als eine Spezifikation zu behandeln und Verifikation mit dem Model Checker CBMC zu versuchen. Technisch ist es zwar möglich, die Verifikation *nach* der Implementierung zu erledigen; allerdings gibt es dabei eine Reihe Fallstricke und gefundene Fehler sind zu diesem Zeitpunkt nur noch schwer zu beheben.

Drittens, die B Sprache in Clojure *einzubetten* um programmatisch (Teile von) Constraints zu generieren und zu lösen oder auch so konstruierte B Maschinen zu animieren oder zu model checken. Dieses Verfahren behandelt Spezifikationen als einfache Daten. Dies folgt den Ideen von Lisp und erlaubt es Werkzeuge zu implementieren, die Spezifikationen analysieren und transformieren, sowie Domänen-spezifische Sprachen (DSLs) zu erstellen.

Der *zweite Teil* dieser Arbeit befasst sich mit einer Implementierung einer Technik zur Reduktion des Zustandsraumes in ProB, die sogenannte Partial Order Reduction (POR). In unseren bisherigen Experimenten hatten wir wenig Erfolg mit der Anwendung von POR Techniken, sobald es um realistische Modelle ging. Anhand einer großen Sammlung von B Maschinen testen wir unsere Eindrücke und können feststellen, dass in den allermeisten Fällen POR tatsächlich keine Reduktion bringt.

An einer Herausforderung die wir uns setzen — ein Modell eines Stellwerks das für POR Techniken bestens geeignet sein *sollte*, jedoch keine Reduktion vorzeigt — identifizieren wir zwei Idiome, die die Anwendung von POR für höhere Spezifikationenssprachen erschweren. Das erste Idiom davon ist die Verwendung von parameterisierten Operationen, die in vielen Fällen eliminiert werden kann, indem man eine Operation in mehrere umschreibt, und zwar eine für jeden möglichen Wert der Parameter. Dem zweite Idiom, die Verwendung von höheren Datenstrukturen wie Mengen oder Funktionen, kann häufig entgegengewirkt werden, indem man Mengen durch eine Kodierung eines Bitvektors ersetzt, oder indem man Constraint Solver verwendet um die Unabhängigkeit von Operationen festzustellen.

Acknowledgments

For more than ten years, I have been with the STUPS group in Düsseldorf. It is safe to say that I have enjoyed this time very much.

I owe my deepest gratitude to my supervisor Michael Leuschel, for sharing his knowledge, trust and support in many fruitful discussions with me; to Jens Bendisposto, who mentored me during my student years, taught me so much about programming and was always willing to lend an ear and give advice; and also to Sebastian Krings, who mentored me during my early phase as a researcher and co-authored many articles with me.

Another special thanks to Claudia Kiometzis is due, for keeping the administrative side of the university off my back and so many pleasant chats.

I also want to thank all the people I worked with, who taught me a thing or two and provoked new thoughts, in no particular order: Carl-Friedrich Bolz-Tereick, Ivo Dobrikov, David Geleßus, Jannik Dunkelau, Dominik Hansen, Sven Hager, Lukas Ladenberger, Mareike Mutz, Jessica Petrasch, Kristin Rutenkolk, David Schneider, Jonas Schneider, Joshua Schmidt, Fabian Vu, and John Witulski.

Computational infrastructure and support were provided by the Centre for Information and Media Technology at Heinrich Heine University Düsseldorf. Their work enabled the benchmarking tasks of chapters 5 and 6

Outside of the HHU, I must thank all my co-authors for all their work, their knowledge and new insights. Of course, I gratefully thank all anonymous reviewers, who offered constructive feedback or (fairly!) shot down my drafts, for their academic service.

To my family and friends, thank you for your support and patience.

Contents

Introduction

1. Introduction

Since the early days of computing, hardware got significantly more powerful: vast increases in computational power and available memory enabled the development of more complex software and software systems interacting with each other. The increase in opportunities comes at a cost: complex software systems are hard for developers to fully grasp and, so, programming errors occur.

In everyday life, software is almost omnipresent. Thus, there is a broad range in which software errors may manifest: they may lead to inconveniences ("I had to restart my PC and lost a few minutes' worth of work on a PowerPoint presentation"), disrupted infrastructure (e.g., most trains in the Netherlands were not running on April 3, 2022[1]), financial losses (e.g., the Santander UK bank duplicated transactions around Christmas 2021[2]), risk of human lives (e.g., systems in Austrian pharmacies gave a wrong dosage recommendation for medication[3]) or catastrophic failure leading to loss of human lives (e.g., two crashes of Boeing 737 MAX planes in 2018 and 2019, killing 346 people in total[4]).

Formal methods are an important means to assure software quality. Where software may threaten critical infrastructure or endanger human lives, the usage of formal methods, and in particular of model checking techniques and proof, are recommended or necessary to meet legal regulations.

1.1. State-Based Formal Methods

State-based formal methods are suitable to model and verify software: at the core, one reasons about the *state* of the software, i.e., the valuation of all variables at a point in time. Then, *paths*, i.e., sequences of states, can be used to reason about the behaviour of the model.

[1] http://web.archive.org/web/20220404093945/https://www.reuters.com/business/autos-transportation/most-dutch-rail-network-halted-by-technical-problem-2022-04-03/

[2] http://web.archive.org/web/20220103024411/https://techxplore.com/news/2021-12-santander-uk-mn-christmas-day.html

[3] http://web.archive.org/web/20211021004926/https://oesterreich.orf.at/stories/3126243/

[4] http://web.archive.org/web/20220421153916/https://www.businessinsider.com/european-canadian-regulators-to-do-own-review-of-boeing-jet-2019-3

In literature, software is typically modelled mathematically as a transition system[5] [BK08], defined as a 6-tuple $TS = (S, Act, \rightarrow, I, AP, L)$, which consists of:

- a set of states S, typically defined as the product type of all state variable types,
- a set of actions Act, which can be regarded as labels for computation steps,
- a transition relation $\rightarrow \subseteq S \times Act \times S$, that links a program state and an action to successor states,
- a set of initial states $I \subseteq S$,
- a set of atomic propositions AP, which can be regarded as predicates over state variables,
- the labelling function $L : S \rightarrow 2^{AP}$ that labels each state with a set of atomic propositions that are true in that state.

In most cases, one is interested in the set of states $R \subseteq S$ that are *reachable* from the initial states I. Then, the reachable part of the transition relation, $\rightarrow \cap (R \times Act \times R)$, is referred to as the *state space* induced by the transition system.

The transition relation $\rightarrow$ is often described in rules that have a form similar to:

Action: `if` *Guard* `then` *Substitution*

The action may only be executed if the guarding predicate evaluates to true in a state s, resulting a successor state s' following the variable substitutions. Accordingly, then $(s, Action, s') \in \rightarrow$ holds.

These rules allow a description of concurrency where several actors or processes can make progress. Each (deterministic) process has at most one available action. The model then accounts for nondeterminism of the system by interleaving all processes, so that several processes can make progress from a specific state. In other cases, actors or processes might also directly behave nondeterministic on an individual basis, with several available actions. Often, systems become hard to grasp once concurrent processes share (parts of their) state, as this further increases the amount of possible states and (too) many interactions have to be considered.

Proof. One of the antecedents of proven software is the calculus that later became known as Hoare logic [Hoa69]. The basic idea is that a program can be enriched with assertions that can be proved. Later state-based formalisms consider a model of the software instead that shall be proven correct and then serves as a "blueprint" to derive software [Abr07]. Many powerful theorem provers and constraints solvers support this task, e.g., Coq [BC10], Isabelle [NWP02], Kodkod [TJ07] and Z3 [dMB08].

[5] For the verification of temporal properties, typically a variation named Kripke structure is used.

Animation. An animator is an important tool during modelling: While proof guarantees correctness wrt. to the specification, it is also important that one verifies that the specified behaviour is actually correct[6]. Thus, an animator allows *exploration* of the behaviour by interactively executing (and reverting) actions from arbitrary program states. This way, modelling and domain experts can verify that the machine "does the right thing", i.e., certain state changes are allowed as expected, certain traces can be executed and the calculated program states are correct. Part I of this thesis heavily relies on the animation capabilities.

Model Checking. A model checker is a tool that automatically verifies that certain properties, e.g., deadlock-freedom or invariant preservation, are fulfilled. To get an idea, a naïve model checker takes a transition system and simply enumerates all possible program states using a description of the transition relation $\rightarrow$, and verifies the property in each state individually. There are also more involved properties (e.g., temporal properties expressed in LTL [Pnu77] or CTL [BAPM83] that reason about execution traces rather than a single state), which require other verification algorithms [VW86, CES86].

However, an exhaustive approach is often not feasible due to the so-called *state space explosion* problem: typically, the state space grows exponentially in size with the amount of state variables and actions. Thus, more sophisticated techniques have been developed, (e.g., symbolic model checking that is based on predicates representing sets of states rather than state enumeration [BCM$^+$92]). One technique that attempts to reduce the enumerated part of the reachable states is called *partial order reduction* [Pel93] and is the focus of part II of this thesis.

There are many state-based formalisms. Examples appearing in this thesis include:

- B [Abr96] is a high-level formalism which is part of the B-method. As B, its successor Event-B [Abr10], and the supporting tool ProB [LB03] are the foundation of this thesis, they are considered in more detail below in section 1.2.
- Alloy [Jac03] is the name of both a language and an analyzer tool that can be used to describe complex structures, and is based on first-order relational logic. Typically, it is not regarded as a state-based formalism; however, extensions such as Electrum and a recent update (Alloy 6) add state-based behaviour. The model finding engine of Alloy is an efficient SAT solver named Kodkod [TJ07]. The αRby tool [MJ14] also embeds the specification language Alloy into the programming language Ruby. Alloy models can be translated into B machines [KSB$^+$18, KLS$^+$20].
- ASM (Abstract State Machine) [BS03] is a method for high-level specifications. Its concepts are very similar to those of B and Event-B, and can be translated to Event-B [LB16].

[6] In the same vein, consider Donald Knuth's infamous comment: "Beware of bugs in the above code; I have only proved it correct, not tried it."

- Promela is a low-level formalism that is similar to C code. In fact, C code can be embedded and used as part of the transition function. It is supported by the model checker SPIN [Hol97] that generates a C program that verifies the model.
- TLA$^+$ [Lam02] is a high-level untyped formalism sharing a similar abstraction level with B. Specifications can also be written in a programming language-like specification language named PlusCal [Lam09] and can then be translated to TLA$^+$. The most well-known supporting tool is the model checker TLC. A large subset of TLA$^+$ can be translated to B [HL12].
- VDM (Vienna Development Method) [Weg72, Jon90] is a formal method that also has its foundations in first-order logic. Its core language is the VDM-SL (specification language) which is supported by the Overture tool; a dialect VDM++ supports a more object-oriented style including classes and inheritance. The VDM core language and B share a similar abstraction level.

1.2. The B-Method

One rigorous methodology is the B-method [Abr96]. It was devised by Jean-Raymond Abrial in the 1980s and can be seen as an extension to the Z notation [ASM80]: while they share their foundations in first-order set theory, B was designed to offer a more structured notation and to support (i) refinement techniques, (ii) formal proof, and (iii) code generation.

Items (i) to (iii) suffice to give an overview of the methodology that is a *correct-by-construction* approach: First, an abstract, high-level specification is created that is obviously correct. For example, at this stage, a machine may "magically" obtain the result via a mathematical description which may include sets, relations, functions and sequences.

Then, this abstract specification is iteratively *refined*: more detail of how computation shall proceed is added to the machine. Each refinement step is linked to the previous refinement via *proof* obligations. These proof obligations ensure that the overall behaviour is not altered and still represents the calculations of the more abstract machine. If all proof obligations are correctly discharged by automated solvers, manual proof or — in some cases — careful manual review (e.g., after successful model checking), one can be sure that the current refinement still is correct.

Eventually, all high-level constructs are eliminated and an implementation level (named B0) is reached, where only structs, boolean and integer variables as well as enumerated sets occur. Then, a *code generator* can be applied and code can be derived that is correct *wrt. the original specification.*

Event-B [Abr10] is a notation with very similar methodology: it shares the high-level constructs and, roughly, the structure (except operations (i.e., actions) being re-named to events, separation of static and dynamic parts, etc.) with B. While B is intended to obtain correct software, Event-B is intended to verify the behaviour of complex *systems* including several actors, hardware components, etc. Thus, a notable difference is that

certain constructs — conditional statements and loops — were removed in order to streamline proofs. A detailed comparison including available tool support was recently published by Leuschel [Leu21].

1.2.1. Popular B Tools

The modelling process is supported — aside from aforementioned provers — by various tools. An overview can be found in an article I co-authored [BKK+20]. Below, the tools mentioned in the main parts of this thesis are introduced.

ProB

ProB [LB03] is an animator and model checker for B and Event-B. As mentioned above, several other state-based formalisms are supported via translation. It is written in SICStus Prolog [CM12b] and is built on top of SICStus' constraint logic programming solver for finite domains (CLP(FD)) in order to solve first-order logic constraints.

AtelierB

AtelierB [Cle16] is a proprietary IDE (integrated development environment) for the B-Method, supporting both B and Event-B, and is currently distributed by ClearSy. It comes with a variety of tools, including a proof obligation generator, provers, an automatic refinement tool (BART) [BM99] and code generators.

Rodin

Rodin [ABH+10] is an IDE developed for Event-B. Similar to AtelierB, it includes a proof obligation generator and supports a variety of provers. It is based on Eclipse and many additional tools (including ProB) are available via a plug-in mechanism.

1.3. Overview Over the Chapters

The main body of this thesis can roughly be grouped into two parts: first, integrating code with specifications via formal methods tools, and, second, partial order reduction. Figure 1.1 depicts the relationship between the individual chapters. In the following, a more detailed explanation is given.

Part I: Integrating Tooling & Applications

The first part of this thesis is concerned with integration of specifications into complex systems and tools.

While the core of ProB itself is written in Prolog, chapter 2 presents the Java API of ProB. As many developers are unfamiliar with Prolog, the Java API allows easier implementation of tools that integrate ProB. In particular, one can also execute B

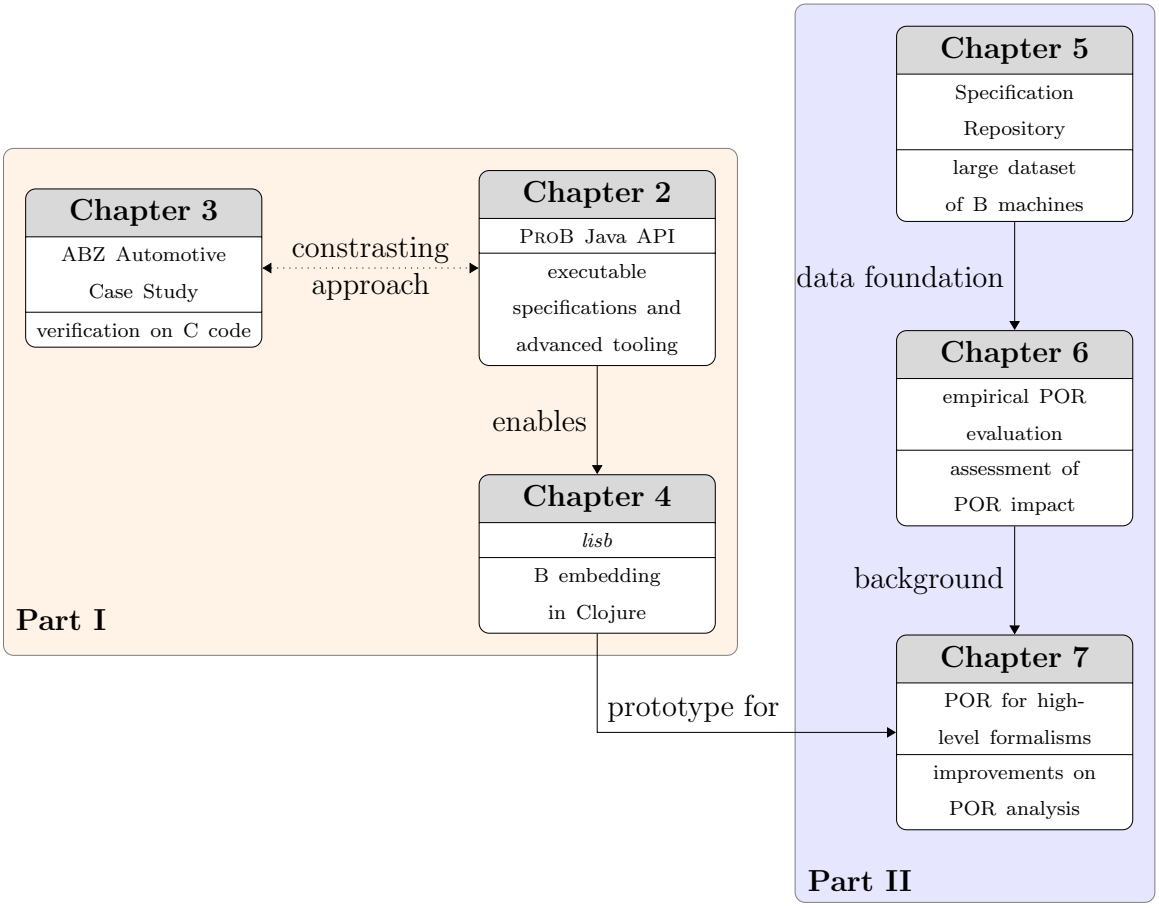


Figure 1.1.: Overview of the Relationship Between Chapters

specifications as part of a program, integrating high-level specifications early in the development cycle.

In stark contrast to this approach, chapter 3 implements a case study directly in C and only later attempts to verify properties. It challenges conventional wisdom that the application of formal methods is faster, less error-prone, etc., and surveys the tool support for verification of programs.

Returning to the Java eco-system, the library *lisb* is implemented on top of ProB's Java API and is presented in chapter 4. It serves as a foundation for tools that programmatically generate or transform specifications by treating them as plain data. An early version of *lisb* is already featured as a case study in chapter 2.

Part II: Partial Order Reduction

The second part of this thesis is on the current state of the partial order reduction (POR) technique in B, identifies obstacles and suggests improvements.

Chapter 5 concerns the large collections of B and Event-B specifications gathered during the development of ProB. The work of this chapter organises a repository of

benchmarks and makes information about the machines available, such as the number of reachable states, number of operations or ProB's runtime.

In chapter 6, the benchmark repository is used in order to grasp how effective the partial order reduction technique applied on B machines is in practice.

Since the results of this evaluation are rather negative, it motivated further investigations regarding the static analysis of POR. In chapter 7, two widespread patterns are identified that hinder successful application of the technique.

The bachelor's thesis of Jan Roßbach [Roß22] provides a link between the two parts: He implemented an automatic refinement tool using *lisb* that transforms the previously identified patterns hindering POR into constructs more suitable for static analysis. Indeed, the resulting machines are more susceptible to the existing implementation of partial order reduction in ProB. The developed tool even allowed a breakthrough for larger models, in particular the grand challenge presented in chapter 7.

In the following, the background motivating each chapter is given. Further, the individual research questions are stated and the design and methods to obtain answers are described.

1.4. Integrating formal specifications into applications: the ProB Java API

A typical formal methods workflow separates the development and proof of a specification from obtaining and embedding a final software product, may it be generated code or manual implementation, into a real cyber-physical system. Accordingly, formal methods tools usually are only intended for verification and do not offer an API to interact with them in a more fine-grained manner.

In contract, the ProB Java API exposes ProB's capabilities for constraint solving, animation and model checking. Individual B states are translated to Java objects and can be inspected programmatically. Further, one can evaluate predicates, examine which transitions are enabled and guide the animator by hand to successor states. Accordingly, it is the foundation of a new JavaFX frontend of ProB [BGJ+21] and several other tools (cf. section 2.5.3 and chapter 4).

Aside from developing formal methods tools, this API also gives rise to embedding formal specifications into applications: instead of generating code, one can animate the specification and execute state transitions based on external input. In chapter 2, we present different application patterns and discuss benefits and drawbacks. This includes (real-time) applications with user interaction such as Pac-Man, a chess engine driven by non-determinism of the specification and bounded model checking, and constraint solving for university time-table planning.

All these applications technically could have been directly implemented on top of the ProB core in Prolog. However, Prolog is arguably harder to grasp for software engineers, and its library eco-system is significantly limited compared to Java.

1.4.1. Research Questions

In the past, arguments pro and contra executability of specifications have been brought forth [HJ89, Fuc92, GH96]. The discussion seems to be settled and it is ascertained that (accidental) executability of specifications is not harmful as long as expressiveness of the formalism is not limited. However, in our experience, specifications that work well for animation are often less suitable for proof, and vice versa. Thus, we raise the question:

Research Question 1. *In what circumstances should (high-level) specifications be executed?*

A similar question is concerned with the application that may be embedding a formal specification or a formal methods tool. Whether such an approach is sensible depends on the kind of application. In certain situations, such as embedded systems or harsh real-time constraints, embedding (high-level) formal methods tools is less suitable or even impossible. We ask:

Research Question 2. *What kinds of applications can reasonably interact with a formal methods tool?*

A similar approach is that an abstract specification is refined until code can be generated from the specification. Typically, this requires many refinement steps, as set operations, functions, etc. have to be eliminated, until the B0 level is reached. A recent tool — B2Program by Vu et al. [VHKL19] — also allows code generation from specifications that are more high-level than B0. Then, one could use the generated code instead of embedding formal methods tools. It is worth discussing:

Research Question 3. *What are benefits and drawbacks wrt. generated code?*

1.4.2. Design and Methods

In the article, we first present the ProB Java API as well as several applications implemented on top. This includes:

Pac-Man as an example for real-time applications: The rules and state of the Pac-Man game are encoded as an Event-B machine. The movement and the corresponding state changes can be considered as a "turn-based" game: During each tick, control of the Pac-Man is first given to the user via keyboard inputs, which trigger the corresponding events in the machine. Then, each ghost is moved by a simple AI. The challenge of this case-study is that animation and visualisation has to occur fast enough, so that the application feels like a real game.

Chess as an example for non-determinism and AI applications: Similar to the Pac-Man game, the board state and legal moves are encoded as a B machine. However, there are no real-time constraints for chess engines and it is acceptable for the view to freeze for several seconds. Here, a small AI (minimax with alpha-beta pruning) drives the exploration of the state space after the user's turn. It then picks the move it deems best for the other player.

ProB Logic Calculator that brings the B (as well as the TLA$^+$) language to the web: A small web server takes the input of a text box, tries to parse it as an expression or predicate and evaluates it using ProB. The result is then rendered back on the web page. It serves as an example how easy it is to embed ProB, as the entire code consists of about 220 lines of Clojure.

lisb is another application in Clojure: an earlier version compared to the one in chapter 4 is presented. It briefly describes the capabilities of embedding parts of the B language in Clojure and using ProB as a backend. The main insight is that this version of *lisb* enables (small) DSLs for constraint generation from external data sources. Such constraints can be directly evaluated using ProB and, in turn, resulting values can be used to drive applications or serve as inputs for new constraints.

PlüS [Sch17, SLW18] is an application for university timetable planning. In the background, there is a B machine that holds the current state of the timetable. The B machines has operations so that the user may check for conflicting courses and to move individual courses in order to resolve these issues. Using the state of the B machine, one can also generate PDF files containing the recommended timetables for each combination of major and minor subject.

ETCS HL3 Concept [HLS$^+$18, HLK$^+$20] is an example of an industrial application: Roughly, trains are controlled not only based on physical track sections (which can be distinguished via axle counters or track circuits), but even on virtual subsections of the same track section. The safety-critical core component, that manages these virtual subsections, is an embedded B model. For a demonstrator, real-world hardware components interacted with the ProB model.

Based on our experience with such diverse applications, we assess what kind of applications may benefit from embedding (high-level) formal specifications.

1.5. A Verified Low-Level Implementation and Visualization of the Adaptive Exterior Light and Speed Control System

The article in chapter 3 describes an experiment that we conducted as an answer to the ABZ 2020 case study. In stark contrast to the article before, we discard the usage of formal methods during development entirely. In what may be more common practice in industry, we started a test-driven implementation in C directly from the specification. We used the opportunity to also experiment with formal methods tools that claim to be able to verify existing programs.

This approach puts conventional wisdom of the formal methods community to the test: typical claims are that employing formal methods during development is faster, better, cheaper and can eliminate nearly entire classes of errors [Hal90, BH95, ED07]. Thus, our implementation provides a baseline that other case study contributions, which actually employ formal methods, should surpass.

1.5.1. Research Questions

While employing formal methods during development is often claimed to be advantageous, there is little data on successful formal methods projects. Even less data is available on projects that have been implemented by two different teams, one using an approach based on formal methods based and the other developing the software in a "traditional" way. We hoped to add to the of evidence that formal methods *are* beneficial. A provocative formulation of the leading question may thus be:

Research Question 4. *How does verification after a non-formal, test-driven workflow ("correcting-the-construction") compare to applying formal methods from the get-go "correct-by-construction"?*

One motivation of the article concerns code generation (e.g., [Vu18]): the correctness of code generators usually is not proven. Hence, optimally, one would verify the desired properties on the emitted code. The question we thus asked ourselves is:

Research Question 5. *What classes of properties of C code are verifiable by existing tools?*

1.5.2. Design and Methods

Initially, we implemented the ABZ 2020 case study in C. We employed a test-driven approach, first testing each individual functional requirement based on the textual description in the specification document [HR20] before implementing it. The order in which we implemented the requirements was based on what was necessary to fulfil the provided validation sequences.

After we did our best to ensure correctness using these traditional means, we selected several requirements and added assertions for CBMC, a bounded model checker for C, to the code.

Chapter 3 is an extended version of the article for the ABZ. In particular, we use the opportunity to compare our approach with the other case study contributions.

1.6. Treating Specifications as Data

> " ... there is nothing wrong with saying: programs process data. Because data is information. Information systems... this should be what we are doing, right? We are the stewards of the world's information. And information is just data. It is not a complex thing. It is not an elaborate thing. It is a simple thing, until we programmers start touching it."
>
> *Rich Hickey — Clojure, Made Simple*

The idea for the first version of *lisb* was proposed by David Schneider [Sch17, Chapter 7] during his case study on data validation of university curricula [SLW18]. Briefly

summarised, the goal was to verify that all combinations of major and minor subjects at the faculties of Arts & Humanities and Business Administration & Economics at Heinrich Heine University Düsseldorf can be studied in the legal standard time ("Regelstudienzeit"). One idea was to generate conforming timetables from scratch using a constraint-based approach [SLW15]. In order to address several shortcomings with the B language and to interact with (partial) solutions that the constraint solver provides, *lisb* was created. The main goal was to transform the course information obtained from the electronic course catalogue into constraints that can be programmatically manipulated, combined and extended.

However, in order to capture more informal constraints, e.g., preferred timeslots of lecturers, it was decided that a state-based approach, where operations move courses from one slot to another and verify candidate solutions, is more practical. This decision led to a slumber period for *lisb*.

This initial prototype of *lisb* covered a subset of B that contains the mathematical notation of predicates and expressions, making it possible to generate complex constraints. Such an approach would have been useful, e.g., for a SAT encoding of the crowded chessboard puzzles [KLK+18]. In order to calculate the number of true boolean variables, bit-wise adders had to be generated. Due to a lack of such a technique, a string was iteratively concatenated in to order to obtain an input file for the solver.

A noteworthy addition is that *lisb* now captures the entire B language, so that entire B (state) machines can be generated. One motivation was to extract and verify models from smart contracts. This allows us to explore the benefits of Lisp — often summarised as *code is data and data is code* — in the context of formal specifications. By treating specifications as plain data, one can take them apart, develop or generate separate components independently and re-combine the results. This facilitates the following tasks, which currently are not possible using plain B:

- programmatic generation of specifications,
- transformation of existing machines,
- minor language extensions such as introduction of new operators,
- development of domain-specific languages,
- re-use of common expressions and predicates.

1.6.1. Research Questions

The DEFINITIONS system of B has several issues as discussed by Leuschel [Leu21]: Some tools insert definitions by text replacement, whereas others insert a piece of AST instead. This may lead to different interpretations due to operator precedence, as, in the latter case, an implicit pair of parentheses is added around the definition call. Further, definitions allow accidental capturing of variables.

Clojure, on the other hand, offers a clean and powerful macro system that — if used correctly — does not have these issues. Such a macro system may be a solution to the question:

Research Question 6. *What kind of issues of B's* DEFINITIONS *can be addressed using a Lisp-style macro system?*

B has no standard way to interact with external data sources. During early development of *lisb*, ProB added support to read and write XML files via external functions (implemented in Prolog and accessible in B) [HSL16]. A more programmatic way is more extensible wrt. new data formats. Especially in the context of data validation, where data might also be required to be pre-processed before its correctness can be verified, we ask:

Research Question 7. *What is a favourable way to integrate external data sources with* ProB*'s constraint solver?*

Many existing B tools use the ProB binary (written in Prolog) directly, or, more recently, the Java API (cf. chapter 2). A language embedding that allows a programming language to construct (parts of) specifications can as well be used for implementation of more advanced tools. This includes DSL tools that generate a B specification, as well as machine-to-machine transformations. An interesting question gauges whether it is preferable to use a tool such as *lisb* and the worth of meta-programming facilities in a specification language:

Research Question 8. *How does meta-programming of B models elevate DSL and tool development?*

1.6.2. Design and Methods

In order to answer these research questions, we have experimented with *lisb* over the course of several years. Early versions included examples based on well-known constraint programming problems, e.g., finding solutions for Sudoku or the n-queens problem. We also generated constraints that solve the crowded chessboard problem discussed in [KLK+18].

More recently, as we widened the scope of *lisb* to deal with entire B machines, we considered tools that generate B machines. In particular, we discuss a *lisb* implementation of the algorithm description language by Clark [CBH+16, Cla16]. Further, the refinement tool by Jan Roßbach [Roß22] gives insights regarding machine transformations.

1.7. Towards a Shared Specification Repository

> "Questions were disorder awaiting organization. The more you understood, the more the world aligned. The more chaos made sense, as all things should."
>
> *Brandon Sanderson — Rhythm of War*

My bachelor's thesis was on distributed model checking via ProB [Kör14, KB18], and my master's thesis fully integrated a B frontend using ProB into LTSmin [Kör17, KML18]. For both research articles, one would expect a proper evaluation of the developed tool. Concerning distributed model checking, one is interested in machines large enough so that at some point during model checking, there are enough unvisited states such that queues of 600 workers can be filled; yet, the state space should be finite. Different techniques of LTSmin may cater to different features of machines. When asking for interesting machines for benchmarks, I was met with a shrug.

A specification repository as suggested in the article in chapter 5 can fix this issue. Instead of having thousands of text files specifying something, we add *data about* the machines. The collected data can be used for regression testing, or testing new B tools (e.g., [Leu20]).

In many formal methods communities, there is no standardised set of benchmarks. In particular, in the B community, machines are often selected arbitrarily with no clear rationale in order to measure impact of novel algorithms and tools. This may be due to lack of realistic machines and can easily introduce a selection bias in the result. In plenty of instances, benchmarks are also not available, or were made available at one time but have not been archived properly. In contract, large sets have been established [BST10, HS00] for SMT and SAT solving, which are built by their communities.

The goals of the article were to provoke reflection on scientific standards, especially on reproducibility and benchmark selection, within the B community, and to ease locating interesting machines for new researchers.

1.7.1. Research Questions

Once a collection of machines grows to a critical mass, it is hard to retrieve specifications with certain features. For example, selecting machines with at least 100 000 states, with at most ten operations or those that can be model checked with 4 GB of RAM, is impossible without additional information. Additionally, the relevant features may differ based on the application: different tools might, for example, parse different dialects of B, machines for data validation significantly differ from those specifying state-based behaviour, etc. Hence, the following question:

Research Question 9. *How can information about specifications be organised in an open, extensible way?*

Once a set of benchmarks is selected, another question is how to make it accessible. It could be described by a set of filters or a text file containing all files. Optimally, the

raw data shall be included as well. In order to achieve reproducible research, we raise the question:

Research Question 10. *How should sets of benchmarks be handled?*

In some cases, new versions of specifications are created after they are published. This may be due to unforeseen errors, bug fixes or conflicting features. We thus ask:

Research Question 11. *How should changes to specifications be incorporated into the repository?*

In the dataset that initialised the repository, we included the public examples that were collected during the development of ProB. It contains benchmarks for different algorithms, examples for several constructs and idioms, test cases, machines from publicly reported issues, teaching material and much more. Thus, the next question is concerned with what a "typical" specification is and how much variety there is:

Research Question 12. *What kind of specifications are available in* ProB*'s public examples?*

1.7.2. Design and Methods

All machines contained in the public ProB examples are loaded and model checked. Some arbitrary timeout is necessary in order to deal with infinite state spaces. For now, we limited the runtime to 30 minutes per machine in order to collect the data.

The results are stored as machine-processable `.edn` (extensible data notation) files. This includes, for example, whether ProB is able to load the machine (as some test cases for the parser are included), the number of variables, operations, states and transitions, whether the invariant holds or whether there is a deadlock. It is extensible wrt. specific information gathered by different algorithms, tools and tool configurations, e.g., runtime, reduced state space size or memory consumption.

1.8. Empirical Evaluation of POR for B

Partial order reduction (POR) is a state space reduction technique for model checking. It exploits the independence of operations $\mathrm{op}_1, \ldots, \mathrm{op}_n$; i.e., if $\mathrm{op}_1, \ldots, \mathrm{op}_n$ are enabled in any given state s, then all permutations of $\mathrm{op}_1, \ldots, \mathrm{op}_n$ must induce a valid execution fragment starting in s and every permutation must yield the same state s'. Then, in the best case, only a single ordering of these operations must be explored.

Ultimately, the optimal reduction depends on the considered property. Checking for deadlock-freedom usually yields better reduction than invariant verification. Often, the state invariant is a predicate containing most if not all state variables. Any operation that may negate the invariant usually is explored regardless of independence. Only so-called stutter events, which are known to preserve the invariant, are not explored. Proving an operation to be a stutter event often is far from trivial. Thus, because of the

additional requirement for invariant checking that is hard to fulfil, one would expect the best reduction for checking deadlock-freedom. Further, verification of a specific part of the invariant that includes fewer identifiers may also yield better reduction than checking the entire invariant. In some cases, it can even be worthwhile to verify each conjunct independently.

For many (lower-level) formalisms, POR is able to reduce the number of explored states and transitions by several orders of magnitudes, resulting in shorter runtime and lower memory consumption. Consequently, even large models can be verified with modest computational power.

One approach to POR, the ample set approach of Peled [Pel93] was implemented in ProB by Dobrikov [Dob17]. While the technique certainly is promising, application to real models did not fulfil our expectations for B. In fact, almost no "interesting" (e.g., large-scale or industrial) B model was susceptible to the implemented POR techniques for neither deadlock nor invariant checking.

1.8.1. Research Question

In the brief chapter 6, we considered the following research question:

Research Question 13. *How well does the current implementation of POR in* ProB *perform?*

1.8.2. Design and Methods

In order to gauge the impact of the current implementation of partial order reduction in ProB, we make use of the specification repository presented in chapter 5. Each machine is model checked two times for 30 minutes without applying POR for different properties: the first run attempts to verify deadlock-freedom, the second run shall verify invariant preservation. Afterwards, the same two runs are executed *with* POR. For all runs, we examine the number of reached states.

Considering different properties may give us insights on strengths and weaknesses of POR for B. We take into account that, due to re-ordering of operations using POR, the number of states reported by ProB may differ if the property is violated (as other parts of the state space are explored first). We also account for machines that time out with POR and do not time out without POR, and vice versa.

1.9. Towards Practical Partial Order Reduction for High-Level Formalisms

The results of the empirical evaluation in chapter 6 made clear that POR works very well for only a few models. As expected, most reduction occurs during verification of deadlock-freedom. However, the vast majority of machines (> 80 %) was not susceptible to the POR technique implemented in ProB.

However, there is a number of models of which we would expect good reduction. The article in chapter 7 identifies some typical features of B machines that hinder POR and suggests new techniques for analysis. In the analysis, we focus on checking deadlock-freedom only, as it is the foundation on which invariant checking is built. Additionally, invariant checking requires operations to be recognised as stutter events, which rarely is the case.

1.9.1. Research Questions

The included manuscript investigates the following question:

There are several theories that could explain the results of chapter 6: first, as discussed by Leuschel [Leu08], B has a high level of abstraction. A single state transition in B can represent thousands or even millions of state changes in lower-level formalisms. For example, in B, one can sort an arbitrary large sequence with a single expression, whereas in Promela, one would have to manually implement a sorting algorithm. It may well be that, for abstract specifications, potential for reduction is "lost" simply by avoiding these intermediate steps. In turn, reduction may occur during refinement of a specification towards an implementation level.

Second, due to the expressiveness of B, there are many ways to formulate the same constraints. In practice, B machines may follow idioms coined by Abrial [Abr96, Abr10], or may be written to suit specific tools such as provers or animators. Such a specification style may have influence on the effectiveness of POR.

Third, an issue may be that independence often simply is not determined during static analysis. One could assume that solver backends time out during static analysis.

Research Question 14. *Why is the application of POR techniques in* ProB *unsuccessful in most cases?*

1.9.2. Design and Methods

Equipped with the knowledge of chapter 6, we select small B machines that feature some degree of independence, and should thus be susceptible to POR techniques. Based on those examples, we identify patterns where POR is not successful and propose *(i)* unrolling of parameterised operations, *(ii)* a bitvector encoding of sets for more fine-grained, fast syntactic analysis, and *(iii)* slower constrained-based checks to obtain additional information as solutions.

Afterwards, we aim to transfer the gained insights to a grand challenge, an academic model of an interlocking system by Abrial [Abr10, Chapter 17], which escaped our POR techniques for many years. This model has been discussed in more detail by Leuschel [LBH14]: it shares many features with industrial models that cannot be disclosed. By forcing a certain operation to be taken as soon as possible, one can reduce the state space by two orders of magnitudes — this behaviour is exactly what we would expect POR to accomplish. Indeed, with a prototypical implementation of the aforementioned transformations and constraint-based analysis, this reduction is achievable on the original model.

Part I.

Integrating Formal Methods Tooling and Applications

2. Integrating Formal Specifications into Applications — The ProB Java API

Abstract

The common formal methods workflow consists of formalising a model followed by applying model checking and proof techniques. Once an appropriate level of certainty is reached, code generators are used in order to gain executable code.

In this paper, we propose a different approach: instead of generating code from formal models, it is also possible to embed a model checker or animator into applications in order to use the formal models themselves at runtime. We present a Java API to the ProB animator and model checker. We describe several case studies that use this API as enabling technology to interact with a formal specification at runtime.

2.1. Introduction

When designing safety-critical software, the use of formal methods is highly recommended [CEN11] to ensure correctness. This is often done by combining (manual and automatic) proof with model checking.

Once a formal model has been found to be correct, it is usually required to translate the model into a traditional, imperative programming language. Then, low-level formalisms are usually close enough that code can be generated easily. When using high-level formalisms though, the model has to be gradually refined to an implementation level so that it only uses a restricted version of the specification language, disallowing high-level constructs which require, e.g., constraint solving techniques or unconstrained memory for execution. The alternative to code generation is manual implementation, which is known to be error-prone.

In this paper, we investigate another approach: we assume that a high-level specification is written to be *executable*, in the sense that a tool like an animator or model checker is able to compute all state transitions. Can we then implement a program interfacing with, e.g., a model checker that also simulates the environment and executes the model by choosing a traversing transition?

This paper is a mixture of a position, tool and application paper, and is structured as follows: in the remainder of this section, we briefly introduce two high-level specification languages, B and Event-B, as well as ProB, an animator and model checker for these

languages. Afterwards, we present the enabling technology, the ProB Java API, which allows for fine-grained interaction with ProB in section 2.2. Following, we evaluate our approach by implementing and discussing several new case studies based on the ProB Java API in section 2.3, summarising its use in existing industrial applications and insights gained from implementation work. In section 2.4, we distinguish an embedding of a formal specification in software from user-driven animation and revisit arguments concerning executability of specifications in the context of our approach and the B language. Then, related work in form of similar tools and other applications built by third parties on top of the ProB Java API are considered in section 2.5. Next, we give an outlook about what kind of applications of the presented approach we may see in the future in section 2.6, before drawing our conclusions in section 2.7.

This article is based on our contribution to the 3rd world congress on formal methods (FM'19) [KBD+19]. It extends the original paper in the following ways:

- We give a more in-depth overview of the capabilities of the ProB Java API, and present some code snippets that show basic usage of its API in a Java application.
- We discuss two additional, previously unpublished case studies that show further use cases of the technology.
- The discussion in section 2.5 is set into a more B-specific context, since the logical foundation is neither intended nor possible to execute entirely.
- We include a more thorough comparison with new experiments regarding code generation as presented in [VHKL19].
- We give a summary of known third-party tools that already use ProB Java API.
- We discuss the integration potential of executable specifications with AI as well as possible future use cases.

2.1.1. B, Event-B and ProB

Both B [ALN+91] and its successor Event-B [Abr10] are state-based specification languages that allow for high levels of abstraction. They are based on Zermelo-Fraenkel set theory with the axiom of choice [Fra22, FBHL73], using sets for data modelling. Further, they make use of generalised substitution for state modifications, and refinement calculus [Bac81, BW12] to describe models at different levels of abstraction [CM12a].

The highest level of abstraction includes, besides set theory, formulation of quantified formulae over arbitrary domains, functional composition and lambda expressions, as well as non-deterministic assignments[1].

In the following, we describe several projects that make use of ProB [LB03], an animator and model checker for both B and Event-B. Its core is developed mainly in SICStus Prolog [CWA+88], with some parts being implemented in C and Java, and makes

[1] Cf. `https://www3.hhu.de/stups/prob/index.php/Summary_of_B_Syntax`

use of co-routines and SICStus' CLP(FD) library [COC97]. Besides B, ProB offers support for several other formalisms as well, including TLA$^+$ [Lam02] (via translation to B [HL12]), Z [SA92, PL07], CSP [Hoa78, BL05] and more. Hence, the approach discussed in this article is immediately applicable to languages other than B and Event-B.

2.2. ProB Java API

As ProB is written in Prolog, which admittedly is neither the most popular nor the easiest language to pick up, it is hard for formal method experts to adapt or extend the default validation capabilities of ProB.

Thus, a main design goal of the ProB Java API was to offer convenient access to the core features of ProB. It allows data exchange in both directions, allowing one to provide inputs to ProB and obtaining the outputs without having to parse streams or log files.

The ProB Java API also allows a fine-grained interaction with ProB, not just one-shot scenarios. Indeed, ProB provides a command-line interface, which can and has been used to develop several tools on top of it (e.g., DTVT [LBL12], OLAF, CAVAL or SafeCap [ILR13] for data validation or BTestBox [dAOMDM19] for test-case generation). However, these tools usually require just a one-shot interaction with ProB: ProB is asked to validate a formal model and ProB's outputs are translated to feedback provided to the user. Many tools require a more fine-grained interaction, with repeated calls to ProB depending on the results of earlier calls.

The source code of ProB Java API is available on GitHub [Proa]. For developers who want to build tools on top of the ProB Java API, we create releases of the tool as jar files than can be consumed using one of the build tools for the JVM such as Maven or Gradle. The artifacts are stored on Maven Central [Prod].

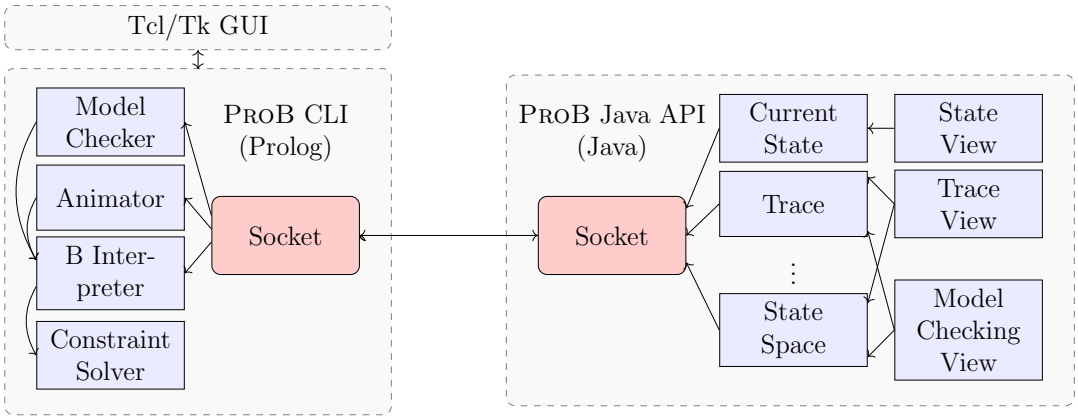


Figure 2.1.: Overview of the ProB Ecosystem

A general overview of the ProB Java API is given in fig. 2.1. For each B model that is interacted with, an instance of the ProB CLI (command line interface) which actually

loads the model is started in socket-mode. This means that the ProB CLI listens on a socket for commands to execute whitelisted Prolog code. The whitelist offers fine-grained access to ProB's constraint solving, animation and model checking capabilities as well as ProB's preferences and machine components.

Each command on the whitelist has a corresponding implementation in the ProB Java API. This offers an API that is fairly low-level and intended for ProB and ProB Java API developers. It is complemented by a high-level API that is built on top and abstracts away from ProB's internals in Prolog. The high-level API allows easy animation of the model, exploration of the state space, solving custom constraints over the variables in the state space, or registration of listeners subscribed to custom formulae which are notified once a new state is reached.

The State Space acts as the central interface to the ProB CLI. It is a representation of the underlying labelled transition system. Exploring the state space by executing operations adds transitions and newly encountered states. It allows animation of the model, evaluation of predicates in arbitrary states, extraction of states that match a given predicate, and, in general, execution of arbitrary ProB Java API commands.

The Model is an in-memory version of the loaded B machine. The ProB Java API offers convenient access to the contents of the specification. This includes invariants, variables, operations and their preconditions, etc. Upon that, it is possible to expand on loaded machines by adding further invariants or operations, resulting in a dynamically altered version with stricter semantics [CBH+16].

The Trace keeps track of the path throughout the state space starting from the initialisation of the machine. Traces behave like a browser history in the sense that they are append-only, but it is possible to "go back in time" and start a new fork. Executing an operation during animation automatically appends the successor state to the currently active trace.

The State objects are linked to their corresponding state space. They store outgoing transitions as well as map abstractions of variables and formulas to abstractions of values. For example, it is possible to retrieve the value of a given state variable but also to add expressions and predicates which are automatically evaluated in every state and are kept track of.

Value Translation is required to give a meaningful representation to the values of state variables. By default, ProB provides a string representation of each value to the ProB Java API. However, they can be translated into Java data structures as well: For example, B integers are translated into BigIntegers, B sets correspond to Java sets and sequences to Java lists. Naturally, this translation does not work for infinite sets. To avoid duplication of the entire state space in ProB and the ProB Java API, only up

```java
public static void main(String[] args) throws Exception {
    String filename = args[0];
    int steps = Integer.parseInt(args[1]);

    Api api = Main.getInjector().getInstance(Api.class);

    // load model, initialize state space
    StateSpace stateSpace = api.b_load(args[0]);
    Trace trace = new Trace(stateSpace);

    // execute specified amount of transitions
    for (int i = 0; i < steps; i++) {
        trace = trace.anyOperation(null);
    }

    State state = trace.getCurrentState();

    // print current state
    state.getVariableValues(EXPAND).forEach(
        (k, v) -> {System.out.println(k+"=>"+v);}
    );

    stateSpace.kill();
}
```

Listing 2.1: ProB Java API Usage Example

to 100 states are cached in Java. If a non-cached state is required, it is retrieved via a handle (a unique state ID) from the ProB CLI.

Trace Synchronisation is a tool that is provided by the ProB Java API. It allows coupling of multiple traces, even on different B models. One example is that a refined machine is synchronised with a more abstract version upon the shared operations, in order to ensure that it is a valid refinement. Another example is synchronisation of two entirely different machines that are two components in a system.

In the following, we want to demonstrate how the ProB Java API can be used within a Java application. The program loads a B model, performs a number of random steps and prints the value of each variable in the final state. The example is simplified to a minimum, i.e., we do not handle errors like missing files or syntax errors in the B model. Also, deadlocks will be ignored, i.e., if no operations can be executed, the animator will remain in the same state. Listing 2.1 shows the animation code, the full code is available on GitHub [Proc].

The entry point to the ProB Java API is the Api class. We use the Guice dependency injection framework [Goo], this means we can retrieve a fully configured Api class using a so-called Injector. The Api class has methods to load formal models for several formalisms, e.g. B, Event-B, Z, CSP, TLA$^+$ and a few more. All methods return a StateSpace object, that is as described the central interface to interact with the Prolog core. We can then create a Trace object and start to execute operations, in this example we perform some random operations. We could pass a filter (e.g. a list of operation names from which we want to choose the operations), but here we pass null which means, that we do not care which operations are used. Finally, we inspect the resulting state of the execution. Here, we print the values for each variable, but we could also inspect the invariant or evaluate arbitrary expressions or predicates.

2.3. Examples

In this section, we describe different use cases based on several examples. The first couple of examples we discuss are student projects implementing two well-known games: Pac-Man and Chess in section 2.3.1 and section 2.3.2, respectively. Furthermore, we present an application in section 2.3.3 that does not represent the typical software development cycle, but rather shows how to use the API in a more creative way: it implements a web application that checks whether predicates (written in B or TLA$^+$) are valid and provides counter-examples when not. Afterwards, in section 2.3.4, we show an experiment regarding domain specific languages on top of B. Finally, the approach found use in two more complex projects, namely a timetable planner for university courses, and a safety critical, industrial application for the ETCS Hybrid Level 3 concept, considered in sections 2.3.5 and 2.3.6, respectively.

For the four software prototypes, we use the state that is translated into Java data structures in order to provide an (interactive) visualisation. The other examples provide feedback to the user via the web-based front-end or uses a read-eval-print-loop (REPL) for user interaction.

Links to all code examples that are publicly available are given in section 2.8.

2.3.1. Real-Time Animation: Pac-Man

Our first example application is based on a formal model of Pac-Man.

The formal model itself is written in Event-B. It specifies all valid positions on the board that the Pac-Man and the ghosts can be in. There are state transitions that describe valid moves, though in the model itself ghosts are allowed to turn around. The model also manages the duration and targets of super pills (so that ghosts may be eaten, but only once per pill), and encounters of the Pac-Man with pills and ghosts. Finally, it keeps tracks of the Pac-Man's lives and deadlocks the game once none of Pac-Man's lives are left. It is possible to play a turn-based version of Pac-Man in the animator.

Note that the model is non-deterministic in the sense that there are multiple available operations, one for each direction the Pac-Man and each ghost may move.

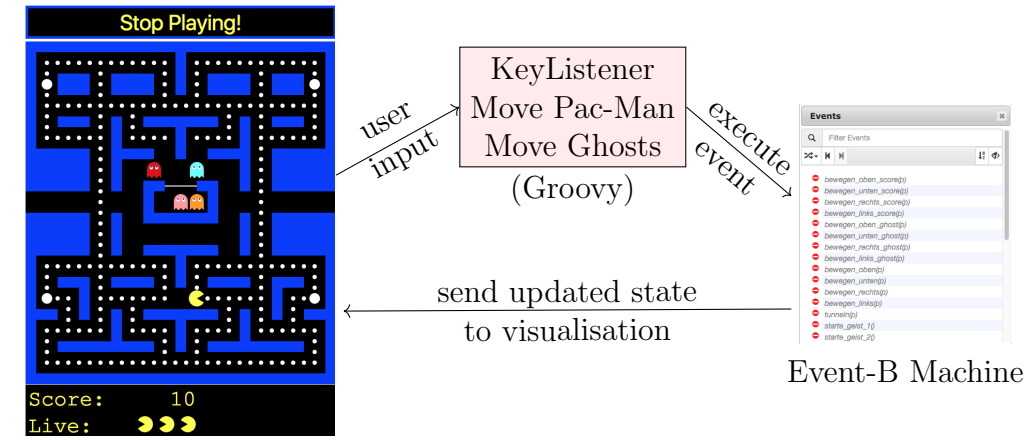


Figure 2.2.: Architecture of a Pac-Man Game Based on a Formal Model

Additional to the model, we implemented an interface via ProB Java API that allows to play the game via traditional controls instead of executing transitions by clicking in the operation view. On the press of an arrow key, the following actions happen:

- In the current state, it is evaluated whether the Pac-Man may move into that direction and the operation to move him is executed if allowed. Operations that result in eating a pill are preferred. This yields a new Trace object.
- For each ghost, it is evaluated whether enough time has passed to leave the monster pen. If so, the transition to move the ghost in a direction mandated by a heuristic is executed. New Trace objects are generated after moving each ghost and the movement operation is appended.
- It is verified whether the Pac-Man or some of the ghosts have to jump to the other side of the board via the tunnel. If the operation is enabled, it is executed.
- If available, operations that catch a ghost or the Pac-Man are executed.
- The GUI inspects the current state of the Trace and updates based on the new state values. The positions of the ghosts and the Pac-Man, the remaining pills, the score and the amount of remaining lives are extracted from the animation state.

For this kind of application, as the calculation of the next-state function is very fast, we did not encounter any performance issues when executing the model. We found that, even though the visualisation is in Java, depending on the operating system and JDK implementation, the game can run smoothly or just below acceptable performance[2]. Yet, we find it especially note-worthy that it is indeed possible to create real-time applications

[2]On a Mac, it runs smoothly. On more powerful Linux PCs, it runs with stutters. We suspect that the socket communication is slower depending on the OS.

that depend on user input based on formal models, as at least five events per tick are executed, one to move the Pac-Man and four to move the ghosts. Plain animation in ProB could not capture this, instead it would turn Pac-Man into a turn-based game.

Main Contribution: Real-Time Animation

The Pac-Man case study shows that our approach is feasible for real-time applications as long as the computation of successor states is not too complex. The application is able to timely react to user input, directly embedding the formal model in the application does not lead to a noticeable performance decline.

Lessons Learned: Non-Determinism

The case study made obvious that it is hard to get the amount of non-determinism right. The formal model itself has to incorporate certain aspects non-deterministically, e.g. we have to take into account every key the player might press. Each time the state of the underlying model changes, ProB has to compute which events are enabled and the successor states they lead to.

However, as the player will only pick a single move out of all the possible ones, most of the events are never really executed and the computation work is discarded anyway. Due to the way ProB and the ProB Java API-based animation interact, we cannot simply let the user move first and then find out if the selected movement is actually valid as we would have to roll back changes to the state space. Simultaneously, the model has to be as deterministic as possible to allow automatic execution. As at least the ghosts are to be moved automatically, the computer controlled aspects of Pac-Man could be modelled deterministically in order to avoid ambiguity and to avoid having to implement how to decide between different options. Yet, we decided that the AI outside of the model should choose between different options, e.g., whether ghosts should turn a corner, resulting in a more general model with a higher amount of non-determinism. Finally, since a "tick" of the game, i.e., one movement of the Pac-Man and each ghost, is made up of not one but several operations that are tested and executed, this impacts performance manifold.

In summary, there is a tradeoff between determinism and execution speed that is both driven by the rules of the game as well as by design decision when modelling it. Further research is needed to find out where the sweet spot between non-determinism and determinism lies when modelling games, in particular, when we have to take into account overheads caused by the animation engine, interactivity, generality, as well as the communication between ProB and ProB Java API. Furthermore, it would be interesting to see if we can develop modelling approaches leading to an optimal tradeoff in general.

2.3.2. Predicting the Future: Chess

In the chess example, we have two use cases. Firstly, we want two (human) players to be able to play against each other. Secondly, a (simple) chess AI should be available to play against.

As with Pac-Man, we use the formal specification in order to specify the rules of the game. The model offers all valid moves as enabled actions, checkmate is encoded as an invariant violation. Then, we can use the vanilla ProB animator to play chess (preferably with an additional visualisation of the current state).

The more interesting part is that a basic AI of a computer-opponent is hard to specify but somewhat easy to implement. Thus, the AI was written in Java using the ProB Java API: we implemented the Minimax algorithm with alpha-beta pruning [KM75]. The calculated game tree has the current state at its root and its children are the successor states representing all valid turns by the AI. Their children again are their corresponding successor states where each state represents a turn by the human player and so forth. For termination, we limit the depth of the state space that should be explored, i.e. the amount of turns the AI is able to look ahead. Hence, this depth determines the AI's strength.

The Java side hereby is responsible for two things. It decides which child states need to be expanded and picks the most beneficial action for the AI opponent based on the explored game tree. Figure 2.3 visualises the execution. After the user's turn, the state space is explored, uncovering all possible courses the game could take. Then, the best action is chosen and the current chess state is updated accordingly. Note that the calculation of successor states happens on ProB side, as the game logic is fully implemented in B.

In the model, the board itself is represented via a square-centric approach: a total function maps each position on the board to either a chess piece or a special "empty" value. While a partial function or piece-centric approach have their individual advantages, this offers benefits for constraint solving, visualisation and easy identification of empty fields. Moving operations are split into two, one that moves a piece and one that additionally takes a piece of the enemy. Their preconditions share predicates for identifying combinations of position and chess piece, movement paths and whether a player is in check.

In order to assign a weight to each state, we use a more sophisticated evaluation function that only depends on a single state. It incorporates both the amount of pieces on the board and their positions and is also specified in B. Then, after checking states until a given depth, the turn suggested by Minimax is picked for the opponent. This strategy is very similar to bounded model checking [BCC$^+$03], though execution is kept explicit instead of resorting to symbolic means. However, regarding this chess implementation we are not particularly concerned with violated invariants other than for identifying a checkmate state (which the AI accounts for). Instead, all possible outcomes are generated via execution of the model. Afterwards, a trace is chosen based on its Minimax value, eventually leading to an action that guarantees the most favourable outcome.

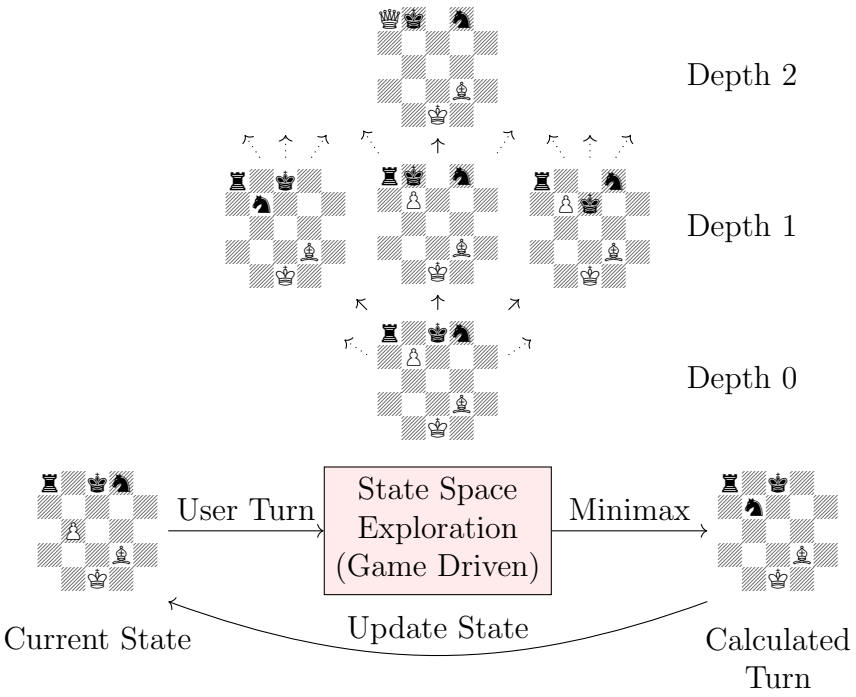


Figure 2.3.: Architecture of Chess Based on a Formal Model

This case study offers worse results than Pac-Man from a performance perspective. Due to the state space explosion caused by the sheer amount of possible moves, generating all successor states as deep as required by a strong chess engine is infeasible. An implementation in, e.g., plain C or Java is orders of magnitudes faster. Modern chess engines usually make use of additional heuristics, and opening and end game databases in order to improve performance. Using our approach following a somewhat naive implementation, only a small part of the state space from a given board position can be generated in reasonable time, which results in the AI being a rather weak opponent.

Main Contribution: Game-driven Model Exploration

In this case study, we replaced the common exploration strategies of ProB (depth-first, breadth-first and random) by an exploration strategy based on the current state of a game. The Minimax algorithm is used to drive the model checker, with the aim of expanding the most promising states, rather than exhaustively analysing the state space. Hence, we were able to implement a heuristic-based model checking approach.

Although ProB offers support for directed model checking [LB11] with a custom heuristic function already, our game-driven model exploration offers a huge advantage. Specifying an exploration heuristic in B is limited to the closed world of the calculated state. For each state, the heuristic provides a value after which it is sorted into a priority queue. It is not possible to argue about the heuristic values of, e.g., sister nodes in the search tree. By animating the model externally in ProB Java API however, we are able to do exactly that: comparing heuristic values of different nodes to decide which states do not need to be explored further by alpha-beta pruning.

In a way, this approach can be understood as a generalisation of directed model checking, as it is not restricted to searching for the common violations of interest in regular model checking scenarios (deadlocks, invariant violations, etc.). The search can, as in this example, be directed at a set of states fulfilling a certain set of criteria. While we here employed a check-mate as invariant violation, the desired criteria do not need to be formalizable in the first place (w.r.t. the state's closed world), but can also take meta information into account. Such meta information can consist of data collected over sister states, current path length, computation time needed for the state, historical data of current path, etc. Hence, it contributes highly customisable control over a highly formalisable set of operations, while not being restricted to pure model checking but allowing a wider range of analysis methods and other applications.

Lessons Learned: Model Complexity

Fully encoding all possible moves on a chessboard has lead to a model that is very complex and features a very large state space. Even though our traversal strategy avoids exhaustively expanding it, debugging and partial exploration were extremely difficult:

- Errors such as incoherences with chess' movement rules sometimes only occurred for certain paths in the state space. For example, castling is only allowed if the king and the corresponding rook did not move until that point. For these cases, it is not enough to verify proper execution of operations in arbitrary states. Instead, the model has to be driven into a particular state (which includes more than just the positions on the board). To some extent, these traces had to be compiled manually.
- Once a target state was reached, it was often hard to understand why particular, complex predicates evaluated to true or false in that state. While ProB offers some debugging tools to do so, debugging B models is not as comfortable as it is for modern programming languages.
- It was hard to determine whether a bugfix covered the error in all states or just in the ones we debugged it in.
- Positions with a high number of possible moves and counter-moves take long to be evaluated by ProB, since the enabledness of all outgoing transitions is computed. In consequence, traversing them during debugging attempts slows down debugging as well.

Furthermore, the high complexity prevented our proof efforts. Further investigation into a refinement-based implementation of chess might help to overcome the difficulties.

2.3.3. ProB Logic Calculator

As an answer to a challenge proposed by Leslie Lamport [Log] we implemented a logic calculator as a web application. The calculator accepts expressions and predicates in

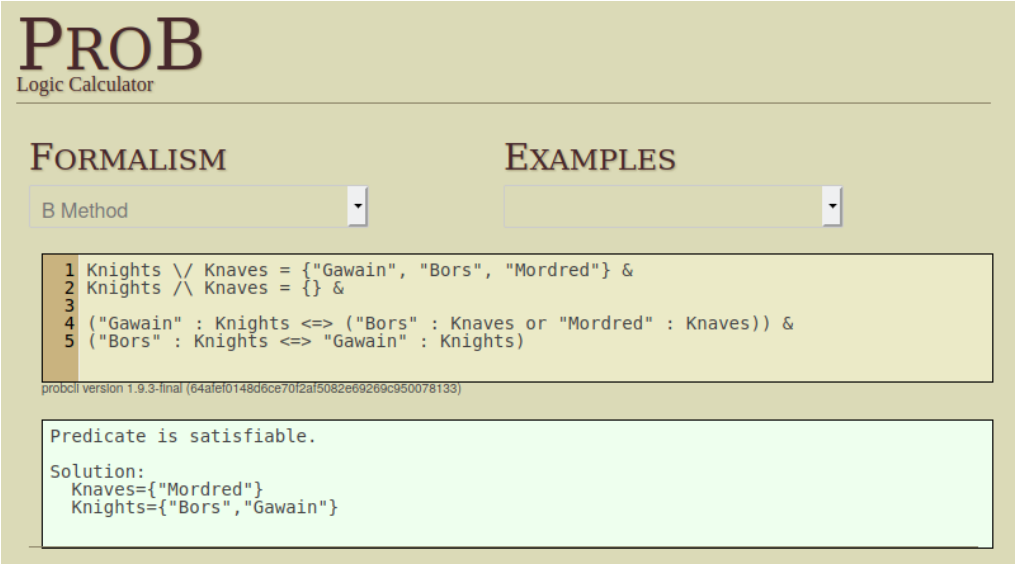


Figure 2.4.: ProB Logic Calculator `http://eval-b.stups.uni-duesseldorf.de` solving a Smullyan puzzle

either B or TLA$^+$ and evaluates them, treating all free variables as being existentially quantified.

As an example, let us find a solution to a puzzle by Smullyan. The puzzle involves Knights and Knaves. While Knights always tell the truth, knaves always lie. We have three persons Gawain, Bors and Mordred and the following propositions:

1 Gawain says: "Bors is a knave or Mordred is a knave"

2 Bors says: "Gawain is a knight"

The translation to B is straightforward. Figure 2.4 shows the web-interface of the logic calculator with a solution to the puzzle.

Main Contribution: Web- and Java Integration

The logic calculator shows that embedding a formal methods toolchain into an application is possible with little effort. Its backend is written in about only 220 lines of Clojure [Hic20] code on top of the constraint solving API that is provided by the ProB Java API. The application source code can be found on GitHub [Prob]. The logic calculator demonstrates how the ProB Java API constraint solver can be used from any language that runs on the JVM and is able to call Java. Furthermore, it shows that both desktop and web applications can be targeted.

Lessons Learned: Get Communication Right

The first version of the logic calculator was implemented as a PHP web page which called the ProB CLI via Common Gateway Interface. This had the drawback of startup time:

a new ProB CLI was started after every evaluation request of the user, leading to a noticeable lag. The ProB Java API enabled us to develop a more flexible website, without noticeable startup time. The formulae are actually evaluated as the user types them.

In consequence, the ProB Java API enabled an easy embedding of formal methods technology into a web service. The logic calculator requires no installation effort, and can be run from any device with a browser. It can be useful for first experiments in the B language and to check out the capabilities of ProB's solver.

The recently developed kernel for Jupyter based on the ProB Java API [GL20] provides a more powerful notebook interface. It is an evolution of the logic calculator, enabling to mix B formulas with text and visualisations, but it requires more effort from the end user to set up. Here, the ProB Java API becomes essential: processing a computational notebook requires multiple calls to ProB, with dependence upon earlier results.

2.3.4. DSLs on Top of B: lisb

The B language is rather inflexible, e.g., the original dialect only supports let and if-then-else constructs for *statements*. In the context of expressions, these features are not available. As an example, it is not possible to retrieve the absolute value of a number by writing `x := IF x > 0 THEN x ELSE -x END`[3].

Another issue is that the B language does not offer a proper macro system. The only means to define B snippets and use them in different places is via C-preprocessor-like macros refered to as *definitions*. These definitions are however not satisfactory, e.g., operator precedences are not always clear and variable identifiers may be captured by accident, which may result in erroneous replacements.

This combination of inflexibility and wonky definitions system lead us to work on lisb: lisb is an experiment that aims to leverage the syntactical flexibilities of a lisp-like language – in this case, Clojure. The key concept is that all B operators of the predicate sub-language (i.e., there is no support for state machines) are implemented in Clojure. Each operator is a pure function that generates a part of the AST (abstract syntax tree) that is used to solve the predicate that is formulated by the user. Then, several parts of the AST can be re-combined before it is sent to the ProB constraint solver.

To continue the example of the if-then-else expression earlier, one could write the absolute value function according to the re-writing rule given in [HL12]:

$$(\lambda t.(t \in \{\texttt{TRUE}\} \wedge (x > 0) | x) \cup \lambda t.(t \in \{\texttt{TRUE}\} \wedge \neq (x > 0) | -x))(\texttt{TRUE})$$

As this construct is not very readable, it might be preferable to write a function `myifte` *once* (using lisb) that *generates* this corresponding AST, and then call it via `(myifte (> x 0) x (- x))` instead.

Overall, this approach gives rise to new, flexible DSLs on top of B, as one can easily write pure functions that return a new AST, potentially combining many operator usages

[3] In recent versions of ProB, this is possible due to improvements discussed in section 2.3.5.

```
user=> (eval (to-ast (b (= (* 2 :x) (+ 1 2 3)))))
{"x" 3}
```

Listing 2.2: Solving a Predicate on a Clojure REPL

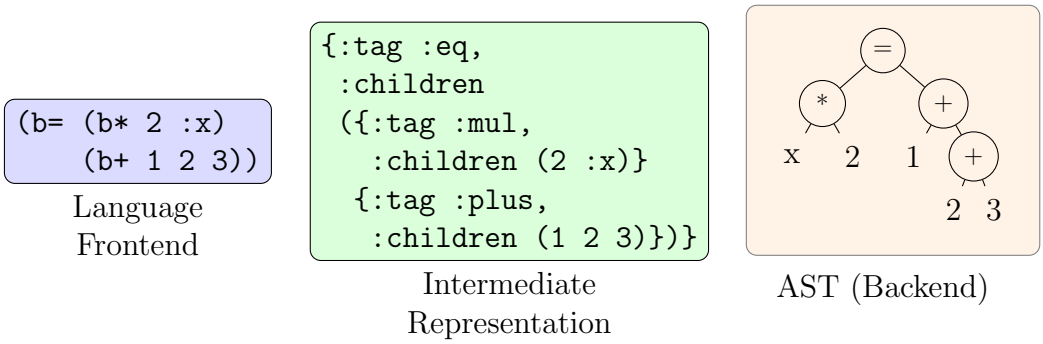


Figure 2.5.: Frontend, Intermediate and Backend Representation of Predicate in listing 2.2

to complex instructions. At the same time, DSLs can allow users to handle, explore and work with the results. Another aim was to explore whether this gives a viable approach for the case study presented in section 2.3.5.

Listing 2.2 shows how a simple predicate such as $2 * x = 1 + 2 + 3$ can be represented in lisb and solved by ProB. The form contained in `(b ...)` is rewritten by a macro into the code depicted as "Language Frontend" in fig. 2.5. This code is then executed in order to create an intermediate representation, also as shown in fig. 2.5. Finally, `to-ast` creates a ProB-specific AST, and `eval` evaluates it using the ProB Java API.

The main idea is that both the language frontend and tool-specific AST can be changed in order to become more independent of B. For example, predicates may be written in a syntax that is closer to other formalisms, e.g., Alloy or SMT, or more specific to a given problem. The intermediate representation only holds for mathematical information unrelated to any formalism. Finally, translations can be provided in order to use tools other than ProB, with the hope that, eventually, (most) predicates might be solved using Z3 [dMB08] or other solvers.

A more involved example is given in listing 2.3, which creates the mathematical constraints required to solve the well-known n-queens problem. How the constraints exactly describe the problem is not relevant here; what is interesting is that some expressions that are used repeatedly can be assigned to identifiers, such as the integer interval `width`. Each form in the code creates part of the intermediate representation and is combined in order to create the entire predicate. Also, the predicate can be instantiated with a size and a (partial) solution for the problem, returning a new intermediate representation in turn. It can be used to find a solution individually, that might in turn be re-used as input for other predicates. Alternatively, the returned predicate may be combined with other predicates as well.

```
(defpred nqueens-p [size sol]
  (let [width (range 1 size)   ;; AST blocks
        q1pos (apply sol :q1)  ;; that are repeated
        q2pos (apply sol :q2)] ;; and can be reused
    (and (member? sol (>-> width width))
         (forall [:q1 :q2]
           (=> (and (member? :q1 width) (member? :q2 width)
                    (> :q2 :q1))
               (and (not= (+ q1pos (- :q2 :q1)) q2pos)
                    (not= (+ q1pos (- :q1 :q2)) q2pos)))))))
```

Listing 2.3: Definition of N-Queens in lisb

Main Contribution: Exploring Domain Specific Languages

The ProB Java API and lisb integrate very nicely: under the hood, ProB ASTs are generated and send to ProB in order to find solutions. The aforementioned value translator then translates the results into Java data structures, which can be used in Clojure as well. In the other direction, it would be cumbersome to write a parser with a proper macro system oneself. Here, lisb takes the advantages of Clojure, or rather Lisp, and extends them into the (B-like) input language. This renders it very easy to create a domain specific language on top of B.

Lessons Learned: Predicates Are Not Everything

Ultimately, lisb only covers the subset of expressions and predicates in B. While the initial state, where all formulas are evaluated, can be set-up individually, state transitions are not (yet) considered. Yet, one of the main advantages of B is not only the expressiveness of the language, but also the usage of a clearly defined state machine. Thus, for a real application, lisb was not sufficient, and more involved usage of the API were necessary.

2.3.5. ProB as a Constraint Solver: PlüS

PlüS [PlU] is an application for planning university timetables [Sch17, SLW18]. The goal is to show that it is possible for students to finish their studies in legal standard time for all courses or combinations of major and minor subjects. If a course or a combination is found to be infeasible, the smallest conflicting set of classes and time frames should be provided such that it can be fixed manually. This process is started from the current timetables. Complete re-generation of timetables is avoided due to informal agreements, e.g., lecturers prefer given time slots or are unavailable on certain days.

A database stores information about all courses, e.g., for which subject they can be attributed, whether they are mandatory or if other courses are prerequisites. From this database, a B model is compiled. This is included in another B machine that allows

checking for feasibility of a subject, move lectures etc. from one time-slot to another and to calculate the unsatisfiable core if applicable.

The formal model is the foundation for a GUI in JavaFX. The initial state is the initial timetable setup. Each course and combination can be checked individually, which triggers the state transition that checks feasibility. If the B model returns that there are conflicts, they are highlighted in the GUI. Then, the user can move courses to different time slots and re-calculate. This is done via drag-and-drop and, again, triggers the corresponding operation in the B machine.

If a course works out with the current scheduling, the state variable that represents the timetable is used to generate PDF files containing a default timetable that can be given to students, so they know in what semester they should attend which courses.

In this application, the interaction with ProB is hidden from the user, i.e., they do not need to know about formal methods, states and transitions. It is currently used by the University of Düsseldorf.

Main Contribution: Improving the B Eco-System

PlüS was one of the earlier projects that used the ProB Java API extensively in the way presented. In particular, the value translator that translates B values into Java data structures, which is used in the other case studies, was created during the development of PlüS. Furthermore, certain shortcomings of B were identified: if-then-else statements are only available for substitutions, but not in the predicate and expression sub-language. Similarly, it is not possible to use `let`-like syntax to locally capture values for any identifier. These have been addressed in newer versions of ProB, which extend the syntax of B in these ways.

Lastly, it is hard to express function-like constructs that calculate values that can be used in predicates. B offers definitions, which offer a macro system similar to the C-preprocessor with all its shortcomings, e.g. shadowing of variable identifiers, which are unacceptable in a formal language. Currently, we work on a language extension for ProB that allows a more sophisticated construct to implement pure functions.

Lessons Learned: Model Interaction

Interacting with the model can be quite cumbersome: in particular, feeding information from scratch into the model can be slow or very complex. Instead, it is easier to generate a large model containing all information.

Initially, the idea was to work on pure predicates without a state machine in order to find scheduling conflicts. However, the aforementioned shortcomings in the language resulted in large predicates with many repetitions that were hard to debug. We found that incorporating the information into a state machine with given operations for manipulation of the schedule is more sensible. Additionally, this offers a simple undo-feature by reverting the trace to an earlier state.

2.3.6. Real Time Animation: ETCS Hybrid Level 3 Concept

We also used the ProB Java API in an industrial project, for a demonstrator of the ETCS (European Train Control System) HL3 (hybrid level 3) principles. HL3 is a novel approach to increase the capacity of the railway infrastructure, by allowing multiple trains to occupy the same track section. This is achieved by dividing the track sections into virtual subsections (VSS). While the status of the track sections is determined by existing wayside infrastructure (axle counters or track circuits), the status of the VSS is computed from train position reports.

In this application, the formal model was used as a component at runtime to control real trains in real time. This can be seen in the video presenting the technology at `https://www.youtube.com/watch?v=FjKnugbmrP4&t=163`, where in the lower center one can see the visualisation (using the ProB Java API) of the formal model. The visualisation shows that two trains occupy the same track section, but occupy disjoint virtual subsections.

The core of HL3 was written as a B model, managing the status of said virtual subsections. For the overall demonstrator, the HL3 model was interfaced with other real-world hardware components:

- an interlocking (IXL) which manages the signals and the status of the track sections,
- a Thales Radio Block Centre (RBC) which communicates with the trains and grants movement authorities
- and an Operation and Maintenance Server (OMS).

These three components fed information into the model via ProB Java API in order to drive the formal model.

The model itself is non-deterministic. Based on the inputs from the external sources, the corresponding operation is chosen. After updating the state of the model, the successor state is passed to a consumer in Java that in turn sends information to the IXL, OMS and RBC. The VBF application, comprising ProB, ProB Java API and the B formal model, performs well enough on a regular notebook computer for a real-life demonstration involving the management of actual trains on their VSS. More details about the model and the demonstrator can be found in [HLS+18, HLK+20].

The overall architecture of the VBF demonstrator is very similar to the Pac-Man example. The Pac-Man board can be seen equivalent to the railtrack topology, and the Pac-Man behaves similar to trains, as they move based on external input. Instead of only visualising the model state to the user, additionally the application reacts to it and communicates with other components.

Main Contribution: Application Based on Model Alone

The ETCS case study fully relies on an embedded model rather than on code generation. By doing so, it has proven our approach to be both feasible and efficient in a real-world application. The overall development time was low when compared to manual

or automated code generation. In addition, the formal model was very close to the HL3 natural language requirements. Changes to the requirements and model could be quickly carried out. Indeed, the use of our demonstrator has uncovered over 40 issues in the original HL3 principles paper, which were corrected in the official document along with our formal model. Of course, a fully refinement-based approach ending with code generation would be able to prove the system correctness and hence deliver a higher level of certainty than our approach does. However, we believe that for prototypes and demonstrators, a model-checked and well-tested specification that is directly executed can beat non-formal software development by a wide margin in terms of development time and costs.

Lessons Learned: Full-Stack Debugging Workflow

One important benefit of our approach was that we could store the formal model's behaviour in log files and later replay these traces in the ProB animator. This allowed us to analyse suspect behaviours, fix the HL3 specification and model, and then check that the corrected model solved the uncovered issues by replaying the trace again. That is, we automatically got record and replay capabilities of debuggers as in [NPC05].

2.4. Discussion: Should Formal Specifications be Executable?

When thinking of executing formal specifications, one usually has animation or code generation (cf. [GK91, WLB00]) in mind. We think that the term "execution" of formal specification is somewhat overloaded; its semantics differ when considering animation and code generation techniques. Finally, embedding gives a new dimension of this.

Thus, in this section, we will first consider differences between those three approaches concerning executability (section 2.4.1). Afterwards, we take another look at the B language in particular (section 2.4.2). Finally, we revisit arguments made in past discussions [HJ89, Fuc92, GH96] whether specifications should be executable or not in the first place in the content of embedding (section 2.4.3).

2.4.1. Executability

As mentioned above, executability of a formal specification can refer to several constructs depending on the context.

First, *animation* of a formal specification is an important means to quickly find errors by executing certain scenarios. This can either be done manually or even replaying a given trace automatically. Executing a longer trace by hand and verifying whether each encountered state is correct is very cumbersome and might be aided by state visualisations.

The most noteworthy feature about animation is that it is a means to develop, debug and reason about the correctness of a specification. Usually, the user interacts with the

tool directly and events are chosen by hand, even ones that should be picked by the environment. In our case studies, that includes movement of the ghosts in Pac-Man, moving the chess pieces of the enemy and providing the input of signals, points, etc.

When considering what it means to *execute* a model in this context, we can characterise it as follows:

- Computation of transitions does not need to be efficient (but it is preferable).
- Constraint solving may fail (or rather: time out), but execution may still continue when manually providing values satisfying the constraints.
- Animation is used during development and covers a large part of the language.

Second, *code generators* usually are applied to (a subset of) the language that offers precise executable semantics. In the case of the B language, this is usually a small subset named B0, which does not include functions, relations, deferred sets, etc. (cf. section 2.5.2 for more details). This approach has several advantages and drawbacks:

- Since B0 is very limited and very close to, e.g., a subset of C, such concrete specifications can immediately be translated to suitable low-level constructs. Then, computation is efficient even for the generated code, in particular when compared to approaches that try to emulate higher-level constructs (though may be less efficient compared to hand-written code).
- Constraint solving cannot fail, since the language (subset) cannot express any constraints.
- Development of the model (usually) is finished once code generators are applied.
- The generated code can directly interface with other components.
- The input language severely lacks abstractions and expressiveness.

Finally, *embedding* is a hybrid approach that tries to combine the best of both worlds:

- Computation must be efficient if the specification demands it, but can be inefficient for proof-of-concept implementation.
- Constraint solving must not fail, since execution would stop in this case.
- Embedding of the specification can be used during development of a prototype as well as be shipped as a finished product.
- The embedded specification can interface with other (existing) components.
- As with animation, large parts of the input language are "executable", yet usually *efficient execution* is required.

Overall, executability has different meanings, depending on the context. During animation, the characteristic question is "Can a solution be found automatically or be supplied?", for code generation, it is "Can the model be translated?", while for our approach, the relevant key question is "Can the specification be executed sufficiently efficiently?" In the following, we use the latter meaning of the word.

2.4.2. B as an Executable Language

For all intents and purposes, the B language is, foremost, a specification language. It was never intended to be executed; a software requirement document should be translated into an abstract model that works "at a more abstract level *execution is no longer possible*" [Abr06] (emphasis in original). In fact, as the B language has its roots in first-order logic, abstract constraints are not even decidable in general. Only a small, implementable subset called B0 (which will be discussed in more detail in section 2.5.2) has defined executable semantics. Why bother trying to execute a more abstract specification then?

During refinement in the traditional workflow, the abstract model is gradually transformed into a concrete model, that at some point is intended to be executed, or rather that code is generated from. Yet, in order to validate that the behaviour of the abstract model is correct in the first place, tools offering animation capabilities are required. Otherwise, errors or inconsistencies in the specification document can easily end up in the final software product. These tools usually rely on constraint solvers or user interaction in order to determine values for execution.

The point that justifies re-visiting the discussion presented in section 2.4.3 is that the following inherent limitation holds: not the entirety of the B language can be executed at all or in reasonable time. In these cases, either an efficient implementation would be algorithmically (nearly) impossible, or some refinement is required in order to state the problem in a way that the constraint solver is able to execute it. As argued, the consequence for shipping such a model to be used as part of software during run-time is different from animation during creation of the model and reasoning about its correctness. Thus, not just *any* specification can be used in a standalone tool. Instead, a certain level of concreteness is required, whereas higher levels of abstraction become *feasible*.

2.4.3. Should Formal Specifications be Executable?

The famous article by Hayes and Jones [HJ89] has led to quite a bit of controversy. It argues that formal specifications should not be executable and gives several counterarguments (CA):

CA1 Proof is more important than (finite) execution,

CA2 Forms of usable specifications are restricted,

CA3 Executable specifications tend to be over-specified,

CA4 Execution is inefficient.

This does not mean that we disagree with these arguments. In the context of the executability discussed in section 2.4.2, these arguments offer valid points *against* our approach. In the following, we want to consider these arguments and give our reasoning

why we deliberately go against this judgement and use a high-level specification language such as B.

Firstly (CA1), we find formal proof to be very important. However, we have observed that for most formal specifications, which are more involved and are written to be executable (in the sense of animation), it is very hard and cumbersome to discharge any proof obligation. On the other hand, models written to be proven usually are not executable (again, in the sense of animation) either. Yet, proof should always be complemented by animation in order to verify that not only the model is consistent in itself, but also describes the desired behaviour. This is a challenge for executability (in the sense of both animation and embedding) that indeed needs to be addressed in the future, may it be by improving the constraint solver that works on the set-theoretical foundations of B, or by searching for new techniques for provers. We think that, currently, we cannot offer embedding of a fully proven specification that is sufficiently complex. Yet, trying to prove unsound specifications may result in a counterexample that quickly raises awareness of an error that may not be uncovered (quickly) by testing or model checking.

Secondly (CA2), as discussed in section 2.4.2, B is a language that is very high-level and allows writing non-executable specifications (as one could encode a non-decidable problem in a single state transition). Instead of worrying about these issues, we try to provide an approach for specifications an animator can handle and execute (efficiently).

Surprisingly, most B and Event-B specifications can be animated with ProB. The major exceptions are mathematical models involving infinite domains, and some axiomatic specifications which require infinite models (e.g., algebraic specification of a stack). Also, sometimes users add complicated axioms to their model: this makes proof easier but animation more complicated. Thus, it can be necessary to separate proof axioms from animation axioms. In [CLM+19] we developed the prob-ignore pragma, to annotate proof axioms not necessary for animation. Animation configuration is kept in separate refinements of the proof models, which allows animators and provers to coexist peacefully on the same development. Animation ensures no inconsistency in model, proof ensures we scale to all instances/topologies.

Thirdly (CA3), over-specification does not seem to be an issue for our use case. An example from [HJ89] is a sorting algorithm. In B, this can be calculated by the constraint solver by purely specifying the *property* what it means for a sequence to be sorted. An example for a valid B predicate that can be solved by ProB in order to yield a sorted sequence is given in fig. 2.6. Note that no concrete implementation is specified, as the problem is solved declaratively. Moreover, in the typical workflow of the B-Method, a concrete implementation happens during refinement. Thus, the writer of the specification is usually *able to choose* the level of abstraction herself.

The last argument concerning performance (CA4) is carefully reviewed for each of our case studies individually in section 2.3 and overall in section 2.7. As different performance constraints are given on a case-by-case basis, it is hard to classify specifications that are suitable to be embedded.

$$
\begin{aligned}
&input = [12, -3, 42, 7] \land && \text{(input sequence)}\\
&output \in 1..size(input) \rightarrow ran(input) \land && \text{(type of output)}\\
&\forall e \in ran(input) \cdot (card(input \rhd \{e\}) = card(output \rhd \{e\})) \land && \text{(keep elements)}\\
&\forall i \cdot 1 \leq i < size(input) \implies output(i) \leq output(i+1) && \text{(ordering)}
\end{aligned}
$$

Figure 2.6.: Sorting Predicate

2.5. Related Work

There are several tools that are able to achieve part of the case studies presented, e.g., state visualisation tools and code generators that we will present in section 2.5.1 and section 2.5.2, respectively. Some additional applications that other researchers and industrial practitioners already built on top of the ProB Java API will be discussed briefly in section 2.5.3. Finally, there are approaches that work very similar to ProB Java API. We will take a look at those in section 2.5.4.

2.5.1. Visualisation

All the presented projects include a GUI which displays a visualisation of the current state. State visualisation by itself is a useful tool to understand the application state more easily and is often used during the development of a model, debugging, and also to explain it to a domain expert.

BMotionWeb [LL16, Lad17] is a tool for state visualisation based on web technologies. It also builds upon the ProB Java API and allows simple interaction with the model. The chess example from section 2.3.2 uses this tool both for visualisation and embedding the script that controls the AI. A heavy disadvantage however is the complex technology stack: BMotionWeb builds upon ProB Java API and uses Groovy, SVG, JavaScript and HTML5, where each component of the stack may go wrong, rendering development very cumbersome. Thus, a more simple successor was developed called VisB [WL20]. It also builds upon ProB Java API, but is easier to use and maintain.

State visualisation is not unique to the B formalisms: e.g., another tool that allows visualisations based on web technologies is WebASM [ZGS14], which works on top of CoreASM [FGG07]. CoreASM is a tool that can be used to execute abstract state machines (ASM). Another advanced visualisation framework is PVSio-Web [WRM18] for PVS. Also, for Event-B a series of other visualisation tools were developed, such as Brama, AnimB and JEB [YJS13] which includes a JavaScript interpreter for B.

2.5.2. Code Generation

A more traditional approach is to generate (low-level) code based on the specification. Translation tools usually cannot work on most constructs that high-level formalisms have to offer, e.g. calculation of an appropriate parameter for an operation, set compre-

hensions or solving quantifications usually require constraint solving techniques which are infeasible to generate.

A popular implementation-level subset of B is named B0 [Abr96, Cle16], from which translation into an imperative language is fairly straightforward. Many features of the B language are missing though, including many operators on functions, relations and sets as well as quantifications.

For B and Event-B, several code generators exist. One such code generator is C4B which is integrated in Atelier B [Cle16]. It allows generation of C code from the implementation level subset of B (i.e. B0). However, refining a model of industrial size down to B0 is a notably cumbersome task to do. Another code generator that is capable to cope with a subset of B0 is b2llvm [BDLM14] that generates LLVM code. A notable toolset for Event-B is EB2ALL [MS11], which allows code generation to several languages including C and Java.

Another approach attempts code generation from a higher level of abstraction. Modern programming languages offer, e.g., sets and maps, which allows easy translation of B constructs such as sets, relations and functions. Supporting code generation for more constructs from B that are not included in B0 might make an approach using code generation more feasible. One such code generator is EventB2Java [CR16, RCWR17] that translates higher-level constructs. More recently, B2Program [VHKL19] was presented.

When translating aforementioned data structures, one loses an important property of the resulting program: execution is not necessarily possible in constant memory, i.e., without dynamic memory allocation. B0, on the other hand, is intended to be generate code suitable for, e.g., embedded systems, as failure to allocate memory is not handled as part of the formal method. This aligns with the approach we present in this article: the overall idea is that prototypes can be executed during early development stages.

However, there is one fundamental difference to code generation: performance often is drastically better due to access to ProB's constraint solver. Generated code must, e.g., enumerate and apply a filter predicate to all possible integer values. As an example, the set $\{x|x \in NAT \land x < 3\}$ is calculated by the code generated by B2Program shown in listing 2.4. The code is far from optimal: since the range of 32-bit (signed) natural numbers is very large, computation of the set is very slow. While this approach is feasible if the domain is small enough, usually application of constraint solving techniques is far more preferable, especially on large or even unbounded domains.

Code generation is not exclusive to the B method: e.g., VDM specifications can be used to generate C++ or Java code [JLC15]. In particular, higher-level constructs such as set comprehensions are handled by this translation as well. Moreover, when targetting Java, this code generator can also translate pre- and postconditions to JML (Java Modelling Language) annotations, that allow optional checks of correctness of the system realisation [TJLL18].

```
BSet<BInteger> _ic_set_0 = BSet<BInteger>();
for(BInteger _ic_x :
        (BSet<BInteger>::interval((BInteger(0)),
                                  (BInteger(2147483647))))) {
    if((_ic_x.less((BInteger(3)))).booleanValue()) {
        _ic_set_0 = _ic_set_0._union(BSet<BInteger>(_ic_x));
    }
    ...
}
```

Listing 2.4: Java Code Generated by B2Program

2.5.3. Other Tools

A variety of third-party tools have been developed using the ProB Java API, highlighting that it can be used as good way to build tools.

- The HRemo tool for invariant discovery, where ProB is used to produce traces with undesirable states, fed to the machine learning system HR. HRemo is described in detail in chapter 4 of [Rod13] and used, e.g., in [GIL12].
- The VTG (Vulnerability Test Generator) system [SLF+12] is based on an Event-B model of the JavaCard bytecode operations. VTG then creates mutants of those JavaCard operations using the ProB Java API, and then uses ProB to generate test traces to exercise those mutants. Note that the generation of the mutants with ProB Java API was orders of magnitude faster than the original code within Rodin.
- CODA [BCE+13] is a refinement-based framework for modelling component-based embedded systems. The animation and simulation features were implemented via access to the ProB Java API.
- Cucumber Event-B `https://github.com/tofische/cucumber-event-b` is a tool to execute test scenarios described in the Gherkin language for Event-B models. It is used for high-level assurance tests [FD19].
- Meeduse `http://vasco.imag.fr/tools/meeduse/` is a tool for domain specific languages building upon EMF (Eclipse Modelling Framework). The domain specific languages are translated to B and the operational semantics realised with ProB. Meeduse has been applied, e.g., to develop a domain specific language for the railway domain [ILW+19a].
- The VDM interpreter of the Overture tool [LBF+10] was integrated with ProB in order to execute implicit VDM specifications [LIL15]. In particular, ProB's constraint solving capabilities are used to find solutions for parts of the specification, that are not in the executable subset of the VDM specification language. This integration makes use of an early version of the ProB Java API. Due to lessons learned from other projects and performance improvements, especially concerning

record types (as they were used in PlüS, presented in section 2.3.5), the current ProB Java API could render similar integrations with other tools much easier.

Other tools using ProB Java API exist, e.g., the data validation tool Rubin developed in collaboration between Thales and the STUPS group.

2.5.4. Other Approaches

Another formal specification language is part of the Vienna Development Method (VDM) [Jon90]. A well-known tool for VDM is Overture [LBF+10], which implements an interpreter in Java. In [NLL12], an extension to the VDM language and Overture was presented. It allows execution of Java code from VDM specifications and, in turn, to control the interpreter to evaluate expressions in the current state. The goal is to add visualisation of the current state to the model and to integrate models with legacy systems, as we did, e.g., in section 2.3.6.

An application that can also be understood as "execution" of a formal model is co-simulation [GTB+17]. Formalisms usually differ in their application area: e.g., B is a formalism used to model discrete events, whereas behaviour regarding continuous time is hard or impossible to express. Using interfaces such as FMI [BOA+12], one can combine several models in different formalisms. A co-simulation orchestration engine, such as Maestro [TLG+19], usually manages the passage of time and synchronises the execution of all models. Such a component that is orchestrated can be either be a *tool-wrapping FMU* (functional mock-up unit) or a *generated FMU*. Tool-wrapping FMUs implement the FMI in a way that tool exposes the behaviour of the model to the interface, such that a high-level model can be used. An example is the Overture FMU extension [TLL18]. Generated FMUs usually stem from a model and are exported by a tool. Then, the generated C code represents a dynamic system and implements the interface of the standard. Again, the Overture is able to generate such FMUs [BTJHL17].

Built on top of FMI, INTO-CPS [LFW+16] focuses on co-simulation with pragmatic integration of current industrial-strength tools. It also offers, amongst others, model checking, hard- and software-in-the-loop simulations. INTO-CPS also provides usage different levels of abstraction of the models [TN16], and gives rise to visualisation techniques such as augmented reality. One future endeavour is to create models of the physical world as a "digital twin", in order to simulate complex cyber-physical systems [FLP19].

Another approach [Num13] is interesting as well: in his thesis, Nummenmaa executes several example runs on probabilistic specification of games. The idea is to leverage non-determinism in order to simulate and analyse game design. For this, the DisCo method [JKSSS90] is utilised. Yet, in these simulations, the model does not interact with an environment.

2.6. A Look Into the Crystal Ball – Potential for the Future

In the previous sections, we have presented several applications that already use the ProB Java API and discussed the circumstances, under which we deem such an approach reasonable. Now, we want to take a look into the crystal ball and discuss some potential use cases that we did not implement yet, but seem very promising. We discuss integrations with more sophisticated artificial intelligence than the one used in the Pac-Man or Chess case studies and link to existing research on that topic in section 2.6.1. Finally, we will name some other future tools that ProB Java API allows us to implement in section 2.6.3.

2.6.1. Integration Potential with Artificial Intelligence

As already mentioned in the discussion of the chess case study in section 2.3.2, the approach of defining custom AI to control ProB's exploration strategy corresponds to the notion of directed model checking. Using AI heuristics for directed model checking is already a researched approach in the community, for instance utilising AI planning heuristics [KHDB06], Monte-Carlo Tree Search [PF15], or relaxation techniques for heuristic finding [SH09]. However, the integration with executable specifications allows for a more general notion of directed model exploration and analysis.

As the ProB Java API offers the possibility to simulate modified copies of a model in-memory, it is also possible to introduce a self-repairing model checking routine. Combining regular model checking techniques with a constraint-based repair method as outlined in [SKL18] would allow a ProB Java API based controller to alter the model once a violation is found by applying generated repair suggestions. The model checking could then continue on the repaired model to search for possible further violations. This can be done for multiple possible repair suggestions, allowing to directly discard faulty ones automatically or present the user a set of viable options once all states were explored.

On another note, a component-wise integration as proposed in [HMR+19] is also easily possible with ProB Java API. The idea is to compose independent executable specification components (ES-only components) with AI-only components, where both, the ES and AI components only address particular subgoals of the overall system. The refinements in this approach are done by extending the system with further components or by splitting components into smaller ones. The splitting step allows for substitution of a subgoal of the original component for an implementation of the respective other type, i.e., splitting an ES component into three subcomponents where one is implemented as AI component. Vice-versa, (partially) replacing AI components with ES components allows for a more provable system. Hence, applying this approach to an initial AI-only system might increase the provable guarantees of the system at least for certain subgoals while not having to switch to a fully formal workflow.

2.6.2. Tool-Wrapping FMU

The B language itself has no notion of time: all events are discrete and happen instantaneously. Yet, often time constraints are important and are expressed by state variables. One strategy is to add a time variable that is only incremented by certain events. The drawback is that this way, the state space usually becomes infinite and, thus, exhaustive explicit-state model checking is impossible. Another way to model time is to maintain a collection of deadline timers that count down instead. This is method is described and used in [CMR07, RC07, HLK+20, LMW20].

The ProB Java API allows users to superimpose any notion of time on their model, whatever modelling strategy is used – or manage time outside of the B model. Thus, it is feasible to implement the FMI standard as an individual tool-wrapping FMU (which, however, requires the development of C glue code via Java's native interface). It would also be sensible to explore how continuous behaviour can be modelled and verified as well, making use of hybrid automata [Hen00].

2.6.3. Future Use Cases

Embedding a formal model directly into applications has several benefits that might aid enabling future use cases.

First, the model is closer to the actual hardware. This allows it to be included in real-time simulations of the system, including all components and the actual (rather than a modelled) environment. This also allows usage of formal models for hardware-in-the-loop tests, which are common for instance in the automotive industry (cf. [FFHS06, SP08]). Having the model included in full system-level tests should help remedy some concerns regarding the loss of fully formal proof.

Second, as part of a regular application, formal models can easily be accessed programmatically. This enables new kinds of analyses not readily available in current formal methods toolchains. For instance, the ProB Java API can be used to define, execute and analyse test case as well as user usage scenarios. This is especially handy, when formal modelling in used together with specifications including classical use case definitions. Ultimately, the ProB Java API can be used to connect formal models with frameworks such as JUnit and thus enables a tighter integration of formal methods and non-formal development. In particular, it might be easier to formulate test cases in the sense that they can be expressed in a way that is closer to the specification or more involved than, say, an LTL formula.

2.7. Conclusions

In this paper, we presented the ProB Java API, which offers an easy to use interface to the ProB animator and model checker. The ProB Java API renders it possible to write applications that interact with a formal model at runtime, offering declarative programming, rapid prototyping and easy debugging. Furthermore, we embedded formal models into actual applications and investigated this approach via five different case

studies. We also considered counterarguments regarding executable specifications and re-evaluated them given the gained experiences.

Overall, we can draw the following conclusions:

- We think that specifications can and should indeed be executable, as it allows verification of an interpretation or an implementation against the specification. Given a suitable high-level specification language, many counterarguments such as over-specification do not hold. With a tool as presented in section 2.2 or in [NLL12], it is possible and (often) viable to use that specification as a library in an application, allowing embedment of declarative programming into traditional, imperative programming languages.
- Development of complex components is significantly eased by the level of abstractions provided by a high-level specification language, such as B. Integration with existing code, written in other programming languages or running on different machines, is very useful. When adapting the formal model, changes can immediately be evaluated via a test scenario in the context of an entire application. In contrast, adapting a traditional implementation is more cumbersome and more prone to introducing new, unrelated bugs. Tool support such as model checking or animation proved to be invaluable to uncover errors early on which may otherwise have gone unnoticed for a longer time.
- The main concern for real-life applications, as already stated in 1989 by [HJ89], is performance. Low-level applications written in traditional imperative, functional or even logical programming languages can be orders of magnitudes faster because they can work at lower levels of abstraction. Hence, for many time-critical applications the execution of formal specifications is not the way to go yet. However, as long as performance requirements are reasonable (e.g., if data sets are rather small), utilising formal models at runtime allows us to quickly deploy complex applications that can make use of the eco-system associated with formal methods, from proof to animation and model checking.
- The presented case studies clearly show that the integration of formal models in a typical software development life cycle is possible. Yet, since the entirety of the API can be accessed, the ProB Java API allows for applications that are way more involved and may prove to be the foundation for game changers concerning use cases and accessibility of formal methods. We think that this approach is just scratching the surface of what is possible, especially regarding the integration with AI components, and we are excited to see what academic and industrial usages may emerge.

2.8. Declarations

Funding

The HL3 case study in section 2.3.6 was funded by Thales.

Conflict of interest

The authors declare that they have no conflict of interest.

Availability of data and material

Not applicable

Code availability

All code of ProB Java API and our public case studies is available on GitHub:

ProB Java API source code `https://github.com/hhu-stups/prob2_kernel`

ProB Java API Maven artifacts `https://search.maven.org/artifact/de.hhu.stups/de.prob2.kernel`

API example `https://github.com/hhu-stups/executable_spec_example`

Pac-Man plug-in `https://github.com/pkoerner/EventBPacman-Plugin`

Chess `https://github.com/pkoerner/b-chess-example`

Logic Calculator `https://github.com/hhu-stups/prob-logic-calculator`

lisb `https://github.com/pkoerner/lisb`

PlüS `https://github.com/plues/plues`

ProB Jupyter Kernel `https://gitlab.cs.uni-duesseldorf.de/dgelessus/prob2-jupyter-kernel`

The code implementing the HL3 case study is confidential and cannot be disclosed here.

Acknowledgments

We thank Christoph Heinzen and David Geleßus for authoring and improving the presented Pac-Man application, as well as Philip Höfges for the chess model, AI and GUI. Additionally, we want to thank the many people who were involved in the development of both ProB and ProB Java API, the Slot Tool and the ETCS Hybrid Level 3 case study.

3. A Verified Low-Level Implementation and Visualization of the Adaptive Exterior Light and Speed Control System

Abstract

In this article, we present an approach to the ABZ 2020 case study, that differs from the ones usually presented at ABZ: Rather than using a (correct-by-construction) approach following a formal method, we use C for a low-level implementation instead. We strictly adhere to test-driven development for validation, and only afterwards apply model checking using CBMC for verification. In consequence, our realization of the ABZ case study serves as a baseline reference for comparison, allowing to assess the benefit provided by the various formal modeling languages, methods and tools.

3.1. Introduction

The ABZ 2020 Case Study [HR20] describes two assistants commonly found in modern cars. The overall system consists of two loosely coupled components, the adaptive exterior light system (ELS) and the speed control system (SCS). The ELS controls head- and taillights, while the SCS controls the vehicle's speed. Both have to take into account the environment and parameters defined by the driver. Obviously, both are safety critical components, rendering safety and security a development priority.

In this article, we present our implementation of ELS and SCS. Our approach differs from the other case study implementations in that we do not employ a fully formal development method. Instead, we attempted an approach closer to what might happen in industry, where formal methods are not common yet. To do so, we implemented both the ELS and the SCS directly in (MISRA) C, following a test-driven development workflow. Only afterwards, we attempted formal verification directly on the C code, using the CBMC model checker [CKL04]. Both MISRA C and CBMC will be introduced more thoroughly in section 3.2.1 and section 3.5.2, respectively. Test-driven development and mocking of test objects will be presented in section 3.2.2.

Rationale. Sometimes formal methods practitioners claim to hold a high ground over "traditional" software development or at least claim that there rarely are disadvantages [Hal90, BH95]. The argument seems convincing; yet, we are not aware of any (case) study comparing two teams working on the same project, one employing a formal approach and the other working "traditionally". For this case study, we aim at providing a baseline for comparison with fully formal approaches or other approaches combining formal and informal verification, e.g., as suggested for spacecrafts [YZ16]. We opted to postpone verification as much as possible, to allow a fair evaluation of (dis-)advantages of the individual approaches. Our aim is to examine, whether a rigorous approach is beneficial in the context of the case study. If so, we hope to add to the body of evidence that formal methods actually *are* beneficial compared to "traditional" software development.

Distinctive Features. Several features render our approach unique: Firstly, as the implementation is written in C, it could be directly deployed to an embedded system. Models written in formal specification languages would have to be refined to an implementation level before code can be generated. Furthermore, code generators usually are not proven and might introduce new errors. In cases where code generation is not easily applicable, side-by-side development of code is suggested. However, this approach is error-prone as well.

Secondly, the implementation is close to actual hardware. Code that interacts with sensors or user input is separated, i.e., it could immediately be linked to real sensors. Additionally, our implementation makes use of threads, just as the subcomponents of the system would run in parallel. We expect that most specifications using formal methods simply allow some non-determinism concerning the ordering of state transitions.

In consequence, our implementation allows real-time simulation of the system, whereas state transitions in formal methods usually happen instantly and do not amount for any time elapsed. This also allows usage of our implementation for hardware-in-the-loop tests, which are common for automotive software [FFHS06, SP08].

Thirdly, C together with the MISRA rules restricting its usage stems from the automotive industry and is widely used in practice. Thus, our implementation closely mimics real-world development conditions.

Team Overview. Our team comes from a formal methods background: While all members are very familiar with the B method [Abr96, Abr10], we did not have particular expertise with C development or verification tooling for C. The basic code structure as well as the fixture for the test scenarios was developed by SK and PK in a synchronous meeting. Afterwards, SK implemented the ELS, JD was responsible for the SCS and tests were provided by PK and KR. Formal verification was done by SK and PK. Both the ad-hoc and the 3D visualization was provided by KR due to her individual knowledge in this area.

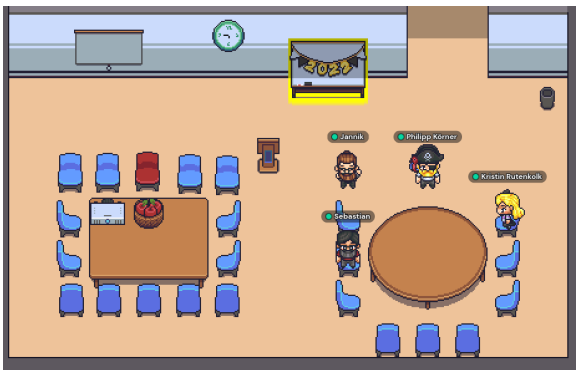


Figure 3.1.: Meeting in a Virtual Seminar Room

Collaboration. As we were working from different locations, we used asynchronous messaging via Mattermost[1] for coordination and progress reports. The code was version-controlled using Git and GitHub. Finally, during the COVID-19 pandemic, we additionally employed the software Gather[2] for synchronous meetings discussing progress and next steps. In Gather, one controls an avatar through a virtual world and video conferences are started automatically based on predefined rooms or proximity. A screenshot of the authors meeting in Gather is shown in fig. 3.1.

Additional Contributions. This article is based on our case study submission [KKDR20] and extends it by

- a discussion on how the given requirements are represented and how far we can trace the impact of requirements on the implementation,
- a thorough presentation of our approach to development,
- improved visualization,
- additional information on the development team as well as the tools and techniques used,
- an evaluation of readability and comprehensibility of our implementation, and
- a comparison to the other case studies.

3.2. Background on Used Methodology and Tools

Our implementation complies with MISRA C and was developed in a test-driven manner. Afterwards, CBMC was employed for formal verification. Below, we briefly introduce these methods and tools.

[1] `https://mattermost.com/`
[2] `https://www.gather.town/`

3.2.1. MISRA C

MISRA C is a set of development and style guidelines for C, introduced by MISRA, the Motor Industry Software Reliability Association. The standard [mis13] defines a subset of C meant to be used for safety critical systems, in particular in the automotive sector. In fact, both ISO 26262 [ISO11] and the software specification by AUTOSAR [aut19] suggest the usage of MISRA C for automotive applications.

The overall goal of MISRA C is to increase both safety and security by avoiding common pitfalls. Thus, the rules prohibit or discourage the use of unsafe constructs, try to avoid ambiguities, and so on. The MISRA C standard distinguishes between three kinds of rules: those that are mandatory, those that are required but could be ignored if a rationale is given, and rules that are advisory only. For instance, there is a required rule stating that any switch statement should have a default label and a mandatory rule stating that any path through a non-void function should end in a return statement.

While most rules could be checked by hand, we used cppcheck[3] to verify compliance of our implementation. Given that some rules are undecidable, the result is only an indication and manual review is required as well.

Despite its prevalence, MISRA C has been criticized regarding both efficiency and ease of use. In particular, the possibilities of false positives [Hat07] and of introducing new errors by (thoughtlessly) changing code to adhere to the rules [BM08] should be carefully considered. Despite the criticism, MISRA C remains the de facto standard in the automotive industry and is used throughout all production code in this case study.

3.2.2. Test-driven Development and Mocking

Test-driven Development is an approach to software development, that follows a certain development cycle: before implementing a new feature or fixing an issue, an appropriate test case is formulated and executed [Bec03]. Without code change, the test is expected to fail.

Afterwards, the code is extended and improved to make the test pass. As a result, a high test coverage and resulting confidence is achieved. Furthermore, the test suite helps during later refactorings.

To simplify formulating tests and to allow testing program parts in isolation, mocks can be used. A mock is an object or library that simulates the input and output behavior of program parts [Bec03]. However, rather than implementing the full functionality, mocks are usually much simpler than the code they replace. For instance, mocks often behave deterministically or even to provide constant outputs. For testing purposes, mocks can record their inputs and provide them to assertions.

3.2.3. CBMC

CBMC [CKL04] is a model checker for programs written in C. It uses bounded model checking [BCCZ99] to verify a default set of properties, mostly related to common pro-

[3] `http://cppcheck.sourceforge.net`

gramming errors, such as: memory safety, including bounds checks and pointer safety, occurrence and treatment of exceptions, and presence of undefined behavior due to C quirks. While those are worthwhile to find and correct, they only ensure general correctness but not adherence to the requirements.

To check individual properties, CBMC can be used to verify user-given assertions stated as C-style assertions using the macros in `assert.h`.

3.3. Requirements and Modelling Strategy

In this section, we give an overview on how we transformed the requirements into code and test, our validation strategy and the limitations of our implementation.

3.3.1. Process From Requirements to Code and Assertions

We used the requirements given in the case study description without further modification or transformation. For each requirement we covered, we generated:

- Unit tests, which are used for test-driven development. See section 3.5.1 for details.
- Assertions to be checked via CBMC as presented in section 3.5. These assertions are meant to verify that properties hold in general rather than just in the test scenarios.

The validation sequences were taken from the Excel file and encoded in integration tests, using the same techniques as the unit tests.

Using CBMC to verify assertions can of course result in counterexamples. Those are given as traces, which can be used to create additional validation sequences by replacing erroneous steps by desired ones. Again, these tests can then be used to improve the implementation and ensure the absence of the counterexample.

The combined approach using both testing for validation and model checking for verification has its merits and provides a high degree of certainty. However, it also has its drawbacks. In particular, the double meaning of assertions can lead to confusion: C-style asserts are used both to encode properties for CBMC and to fail tests. Yet, there is no combined methodology to react on failing assertions and errors uncovered by model checking have to be handled differently from failing tests.

3.3.2. Code Structure

The overall architecture of our implementation is depicted in fig. 3.2. We follow a structure that is fairly similar to the one the specification provides. Since two subsystems are specified, the code is separated into two folders, one for the cruise control and the other for the light system. This is to help ensure that the systems are independent of each other. Shared type definitions, e.g., the pedal deflection, the sensor state enumeration, and shared sensors, are stored separately. An artificial time sensor was introduced for testing, but can easily be replaced by an actual clock.

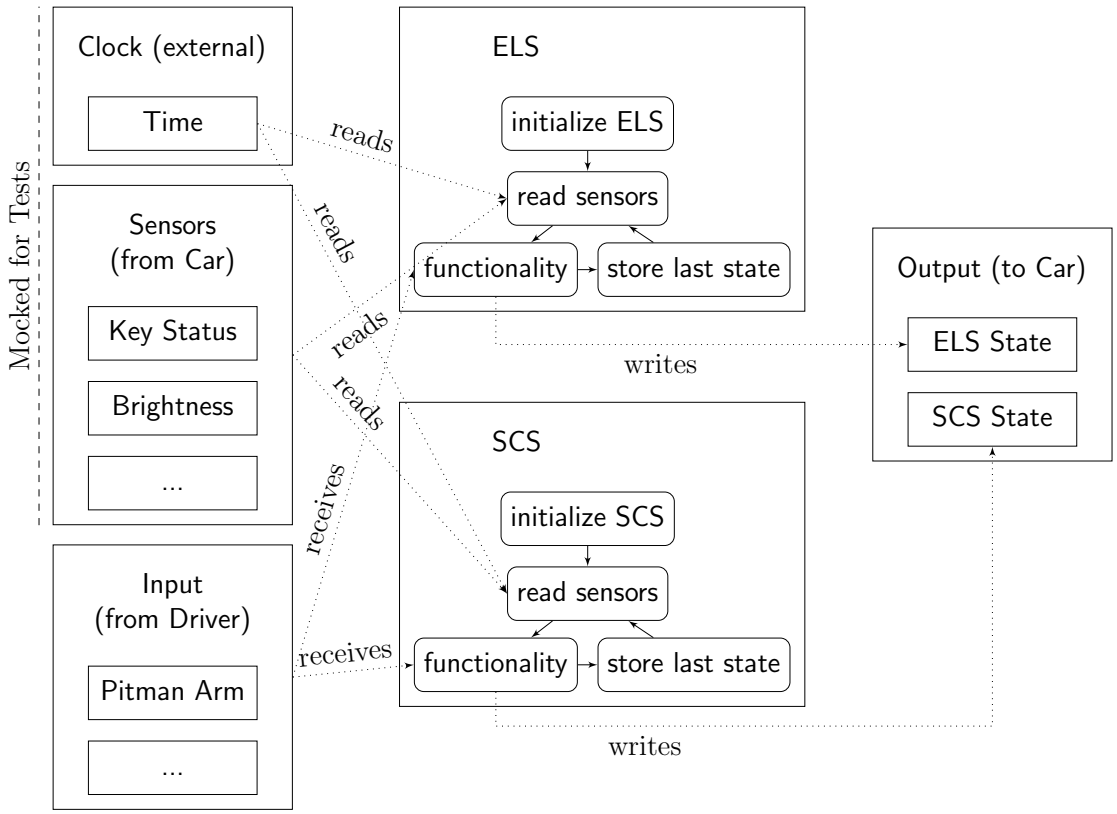


Figure 3.2.: System Architecture and Internal Communication

Each of the subsystems is split into three header files and implementations. The first header file declares the accessible and shared sensors for the subsystem, and contains relevant type definitions. Another header file defines the user interface, e.g., how the pitman arm may be moved or what input the pedals for gas and brakes may yield. The last header file contains definitions for the actuators, i.e., what the system is allowed to do. Only the latter two header files are actually implemented, eventually resulting in three C files:

- A state struct that contains all the data relevant to the subsystem.
- The user interface to simulate inputs. This changes some internal variables that keeps track of the state of the UI; in a deployed system, this can be replaced by additional sensors. The attributes correspond to the signals that the subsystem has to communicate.
- The realization of the state machine with several guarded state transitions. This is the actual implementation of the specified safety properties.

For the test cases, sensors are mocked. In order to get an actual executable, real sensors have to be linked during compilation. The time spent for development, validation and verification is given in table 3.1.

Table 3.1.: Development Time

Task	Time in Hours
basic implementation and code structure	2
ELS implementation, tests and scenarios	30
SCS implementation, tests and scenarios	22
model checking	3
refactoring and code cleanup	2
state visualization	6

For the sake of brevity, we will only show small code snippets in this paper. The full implementation is available at

`https://github.com/wysiib/abz2020-case-study-in-c-public`.

3.3.3. Traceability of Requirements

As the other case studies, we do not employ a fully formal approach to traceability. The only form of traceability we provide is by using naming patterns. Our unit tests in general contain the requirement they are concerned with in the name of the test routine. Larger integration tests also reference the validation sequence they represent, which in turn contains the requirements justifying individual steps.

As a consequence, we can only trace which test cases are validating certain requirements and to what extent requirements are covered. Comments aside, we have no link between a requirement and the individual part of the code realizing it.

3.3.4. Variability of Requirements

In our current implementation, we handled the requirements' variability by introducing artificial sensors that produce a single constant value, i.e., sensors for

- the driver position, (left or right);
- the marked code, (USA, Canada or EU); and
- if the vehicle is armored.

The values returned are fixed and assigned during the system's initialization. While this approach was easy to implement on top of our initial system, it also pollutes both the code and the program state to some extent.

In the automotive industry, a commonly used approach for handling variability is software product line development [CN02, KD06]. If the variability had been larger, we could have split ELS and ECS into a common base product used for all market segments and used a software product line approach to develop individual manifestations, e.g., for Canada vs. USA.

Given that the variability in the requirements was not that large, we opted for the simpler implementation in order to concentrate on validation and verification.

3.3.5. Properties Addressed & Limitations

Due to time constraints, we opted not to implement every single requirement but tried to cover as much as possible. Aside from the emergency brake light, all requirements have been taken into account for the ELS. For the SCS, we implemented about two-thirds of the requirements, up to (including) SCS-28. While it would be nice to have a more complete implementation, we do not think that it would impact our gathered conclusions.

A feature of the requirements that is not addressed satisfyingly are timers. We are convinced that any modern CPU to be used in cars is fast enough to execute an iteration of the state machine withing a reasonable time frame. Thus, any real system realized following our approach should be able to guarantee execution within the smallest time resolution that is relevant to the subsystems and their respective requirements.

Yet, it is hard to give any real-time guarantees. The only evidence that can be given is to run the system often enough and measure whether execution is kept in the specified tolerances. However, this is still better than what we expect of more formal approaches, which usually do not account for wall time at all.

3.4. Model details

In the following, we will detail implementation idioms we employed to simplify handling and verification of the involved state machine, and explore some snippets of our code to showing these idioms in practice.

3.4.1. Formalization Approach

As stated, we postponed verification as much as possible. Instead, as our first step, we set up the validation sequences as test cases. Then, following test-driven development, we added to the implementation code by only considering the first failing assertion in a scenario. Once the test passed, we moved on to the next. In a second step, we added test cases that are directly related to one or sometimes several requirements.

Finally, we set up CBMC and tried to verify the properties described by the requirements. As stated, we use the same code for testing and formal verification, avoiding any translation between verification and testing environments as done for instance by Chen et al. [CRW+17] and others. However, both approaches remain distinct rather than being combined into a single verification procedure [YSAA97].

3.4.2. Modelling Idioms

Besides sticking to the MISRA C guidelines and test-driven development we also adopted two further idioms during modelling: only use enumeration types, and do not expose mutability. We will motivate and elaborate on these in the following subsections.

Use Enumeration Types

We opted to define all types as enumeration types. This is to be expected for some data types, which are true enumerations, such as:

```
typedef enum {Ready, Dirty, NotReady} sensorState;
```

Yet, we also defined integer types as enumerations:

```
typedef enum {
    percentage_low = 0,
    percentage_high = 100
} percentage;
```

The reasons for this are twofold: first, we can easily identify thresholds and the value range for each type. While percentages are straightforward, other values such as the steering wheel angle are not easily represented in a human-understandable format. An excerpt of the corresponding type definition is as follows (analogously for turning the steering wheel to the right):

```
typedef enum {
    st_calibrating = 0,
    st_hard_left_max = 1,   /* 1.0 deg */
    st_hard_left_min = 410,
    st_soft_left_max = 411, /* 0.1 deg */
    st_soft_left_min = 510,
    st_neutral_maxl = 511, st_neutral = 512,
    ... /* analogous for the right side */
} steeringAngle;
```

Such a type definition renders it easier to identify, e.g., in what direction the steering wheel is turned and how far. For instance,

```
st_hard_left_max <= angle &&
angle <= st_hard_left_min
```

can be used to check if the wheel has been turned far to the left.

C behavior is undefined if a value that is out of range of the corresponding enumeration is passed. Thus, our second intention was that model checking tools could easily deduce the actual value range rather than having to consider integers exhaustively. This will be discussed further in section 3.5.2.

Do Not Expose Mutability

It is easy to write broken code when using mutable structs, especially if they are used in order to communicate between threads. Instead, we pass *values* to and from interface functions. This means, that values are copies of the data which are not referenced from anywhere else in the program and the receiver may do however they please with it. An example is that the state from the light sub-system can be queried (for test cases). The returned value will never change unless the test case chooses to do so; no action in the ELS influences it. This also allows reading multiple output variables consistently.

On the other hand, frequently changing *internal* variables, are declared as local (using the `static` keyword). They are always stored in the same "place" and may not be exposed; in particular, there are no getter functions for these variables.

3.4.3. Coding Examples

Below, we present some key snippets of our implementation. We focus on the concept of the ELS systems, as the SCS is structured the same way.

The core of our ELS is the `light_do_step` function, spanning over almost 300 lines of C code, that is called in a loop. Some auxiliary functions exist to properly set the high beam light, blinkers, etc. where it was appropriate to heat the DRY principle. The `light_do_step` function can be divided in three major parts, described below.

Sensor Reads and Type Information for CBMC First, all relevant sensors are read and stored locally. For verification with CMBC (as discussed in section 3.5.2), it is necessary to provide type information for integer ranges and enums. Listing 3.1 shows this for three examples: first, all possible states of the key are enumerated. Second, as C represents booleans as integers, boolean values must be specified to be exactly true or false. Third, integer ranges such as the possible values for the battery voltage have to be provided as an axiom. Additionally, we add assumptions based on the specification, e.g., that the engine state is linked to the key position.

As noted, we implemented time as a sensor as well. Listing 3.2 shows that we also had to add assumptions that the timestamp only increases.

Implementation of Requirements Requirements are encoded by a collection of if-statements. Interestingly, no else-branch exists in the function — most likely, because the specification does not contain the words "else" or "otherwise". In the snippet in listing 3.3, we show how two smaller requirements are realized.

Assertions The last part of the `light_do_step` function contains code for invariant verification. Listing 3.4 contains assertions that can be checked using CBMC to verify two requirements.

```
keyState ks = get_key_status();
__CPROVER_assume(ks == NoKeyInserted || ks == KeyInserted || ks ==
   KeyInIgnitionOnPosition);
bool engine_on = get_engine_status();
__CPROVER_assume(engine_on == true || engine_on == false);
voltage voltage_battery = get_voltage_battery();
__CPROVER_assume(voltage_battery >= voltage_min && voltage_battery <=
   voltage_max);
...
__CPROVER_assume(implies(ks == KeyInIgnitionOnPosition, engine_on ==
   true));
__CPROVER_assume(implies(engine_on == true, ks ==
   KeyInIgnitionOnPosition));
```

Listing 3.1: Sensor Reads and CBMC Assumptions

```
size_t tt = get_time();
__CPROVER_assume(tt >= when_light_on);
__CPROVER_assume(tt >= blink_timer);
__CPROVER_assume(tt >= ambi_light_timer);
__CPROVER_assume(tt >= pitman_arm_move_time);
```

Listing 3.2: Time as a Sensor

3.4.4. Modelling of Time Constraints

When writing code that takes time into account, one is easily tempted to access the current time provided by the operating system. This is a bad when time-based properties are to be tested, as tests would have to be enriched with sleep statements to achieve proper timing for the situation under test.

Instead, we introduced an artificial sensor reporting the current time in milliseconds, comparable to a unix timestamp. For testing, this sensor is mocked and some artificial time is provided. The code does not know anything about time, it just reads a sensor returning an integer value.

The only assumption made is that one cannot go back in time. In consequence, the step functions can be called in a continuous loop, independent of the computing speed and time needed for a single iteration. On fast hardware, there might even be several executions within the same timestamp (e.g., if the resolution is milliseconds) or timestamps might pass without an execution following (e.g., when using nanoseconds). Mocking the sensor also has the advantage that test scenarios, that would take several minutes of wall time, can be executed in milliseconds instead.

If the entire piece of software was to be shipped, it would be trivial to swap out the sensor: One only has to link an implementation that provides the real time, which may be the provided by the operating system.

```
// ELS-16 (has priority over ELS-17)
if(!engine_on && (last_lrs != lrs_auto)
    && (get_light_rotary_switch() == lrs_auto)) {
    set_all_lights(0);
}
...
// ELS-41: reverse gear
if(reverse_gear) {
    set_reverse_light(100);
}
if(!reverse_gear) {
    set_reverse_light(0);
}
```

Listing 3.3: Implementation of two Requirements

```
// ELS-22: low beam => trail lights
assert(implies(blinking_direction != hazard,
   implies(get_light_state().lowBeamLeft > 0,
   get_light_state().tailLampLeft > 0 ||
   get_light_state().tailLampRight > 0)));
...
// ELS-41: reverse gear turns on reverse lights
assert(implies(reverse_gear, get_light_state().reverseLight > 0));
assert(implies(!reverse_gear, get_light_state().reverseLight == 0));
```

Listing 3.4: Verification of two Requirements

3.4.5. Readability and Comprehensibility

Aside from the MISRA rules and the idioms presented in section 3.4.2, we did not employ further guidelines to increase readability. However, as MISRA C is already designed to improve readability [BBH18], and given that C is a widely used language we assume our implementation to still be accessible for non-expert developers unfamiliar with formal methods. In the following, we will revisit our implementation and discuss the readability of the code and hold it against this assumption.

Readability Metrics over ELS and SCS

An intuitive metric for readability seems to be the number of lines of code (LOC). ELS and SCS include 605 and 642 LOC, respectively, not counting 328 blank and 200 comment lines. However, more precise metrics for readability have been suggested.

Buse and Weimer [BW10] show that the average number of identifiers per line, the average line length, or average nesting depth are negatively correlated with readabil-

ity. Simultaneously, the average number of comment lines, and the average number of semantically breaking blank lines are positively correlated with readability. The negative impact of nesting on readability is further pointed out by Johnson et al. [JLY+19]. Taking this into account, our code still seems to be readable. We observe a minimum amount of nesting, with only if-constructs introducing mostly only one extra level of indentation, nesting for at most two levels. The average line lengths for the ELS and SCS are 34.37 and 36.27 characters, respectively, with maxima of 170 and 125 characters. These numbers suggest that the majority of lines is short and comprehensible, with some outliers rendering individual parts of the code less readable.

Subjective Readability of ELS and SCS

Besides readability metrics, a more subjective way of estimating the code readability is to simply try reading it again. The main interest hereby lies withing our two step functions, which are continuously looped over. For the ELS and SCS modules, these functions consist of 276 and 127 LOC, respectively. Both start by accessing all the sensors, partly without processing their return values. The rest of the implementation follows a clear pattern: if-statements checking for a condition to act upon. While the code lacks some comments which explain why certain things are done, the references to the respective requirement from the case study accompany the code fragments as annotations. Overall, as the code is not written in a high-level specification language which more closely captures natural language, the overall readability seems to be limited by the general readability of (MISRA) C code. While this can be seen as a drawback, one could also argue that no further understanding of higher mathematics or set theory is required as for instance in certain formal languages. Thus, the code remains equally readable to experts and non-experts alike.

Readability of the Unit Tests

Following, we want to take a closer look at our tests' readability. Each unit tests follows a pattern of arrange, act, assert, as shown in listing 3.5. As this is a well-known technique, we assume the tests are comprehensible by non-experts.

However, we can observe two points negatively impacting readability. Firstly, some tests involve multiple assertions. Especially in terms of time-sensitive behaviors, we observe patterns where the test advances the timer, asserts a specific property (e.g., light on or off) then repeats the process to assert the property change after a certain time.

Secondly, as this was such a common pattern specifically in the ELS, we introduced macros which reduced boilerplate code, but might have reduced readability. The macro `progress_time_partial` in listing 3.5, line 25 for instance advances time for a given time frame and asserts that a property retains a given value along the way. While incredibly valuable for writing the tests, we acknowledge that the macro's name is not descriptive enough as it does not convey its role as an assert statement. Hence, the readability of the test itself decreases.

Code Pollution due to CMBC Annotations

As C is not designed for formal verification, we found that some annotations for CMBC started to pollute the code. While adding asserts into the code to introduce invariants is straightforward and immerses into the C code quite well, we needed to add further axioms to the code so CMBC was able to properly work with our enum types. This resulted in cluttering of the sensor reads as can be seen in listing 3.1. Here, we ended up with one big block of sensor read and value axiom pairs which impacts readability. However, in most formal languages, this would be a non-issue as they were designed with invariants and axioms in mind and include the as first class citizens appropriately. Furthermore, we added these axioms at the very end of the development process whereas they are much more involved in early development stages in fully formal methods.

3.5. Validation & Verification

We tried to validate our implementation throughout the whole development process by using test-driven development, as we will discuss in section 3.5.1. In addition, we used the CBMC model checker to fully verify different properties of our implementation directly on the C code as we will describe in section 3.5.2.

3.5.1. Test-Driven Development Using cmockery

We used test-driven development based on the provided scenarios. For this, we rely on Google's cmockery library[4], which provides a unit testing framework and allows mocking functions. Since we did not want to execute all tests in real-time, we mocked functions that extract sensor data as well as the current time in our test cases. We used two different kinds of test cases for a first quick validation:

- The provided scenarios were automatized and used as integration tests.
- In addition, we implemented unit tests for all requirements given in the specification document. Of course, each unit test only covers a minimal scenario that shows how the requirement is supposed to be understood and automatizes the verification of that single scenario.

A snippet taken from the test case of the requirement ELS-3 is shown in listing 3.5. The system is initialized to belong to an EU-based car with left-hand drive and without any extras such as ambient light. Initialization and assertions regarding the correctness of the initial state are not shown. Afterwards, in lines 2 to 9, we update the sensors to the values they should hold at the start of the test scenario and the code setting up the mocked functions is called. In particular, we set the time sensor used to simulate the actual clock as described in section 3.4.4. Overall, the test setup phase ensures that our artificial sensors report the required values.

[4] `https://github.com/google/cmockery`

```
1  // ignition: key inserted + ignition on
2  sensor = update_sensors(sensor, sensorTime, 1000);
3  sensor = update_sensors(sensor, sensorBrightnessSensor, 500);
4  sensor = update_sensors(sensor, sensorKeyState,
       KeyInIgnitionOnPosition);
5  sensor = update_sensors(sensor, sensorEngineOn, 1);
6
7  mock_and_execute(sensor_states);
8
9  sensor = update_sensors(sensor, sensorTime, 2000);
10 pitman_vertical(pa_Downward5);
11 mock_and_execute(sensor_states);
12
13 assert_partial_state(blinkLeft, 100, blinkRight, 0);
14 pitman_vertical(pa_ud_Neutral);
15 sensor = update_sensors(sensor, sensorTime, 2000);
16 mock_and_execute(sensor);
17
18 pitman_vertical(pa_Upward7);
19
20 progress_time_partial(2000, 2499, blinkLeft, 100, blinkRight, 0);
21 progress_time_partial(2500, 2999, blinkLeft, 0, blinkRight, 0);
22
23 int i;
24 for (i = 3; i < 6; i++) {
25    progress_time_partial(i*1000,       i*1000 + 499,
26                          blinkLeft, 0, blinkRight, 100);
27    progress_time_partial(i*1000 + 500, i*1000 + 999,
28                          blinkLeft, 0, blinkRight, 0);
29 }
```

Listing 3.5: Test of Requirement ELS-3

Line 10 shows the difference between sensors and driver interaction: While sensors have to be mocked in order to simulate an actual system, user input is given directly. This corresponds to what will happen in an actual car: the system has to react to user input immediately, while it can read sensor data arbitrarily.

Line 13 asserts that the left blinker is on 100% and the right one is on 0% once the step function was executed after the user input was given. We use the function `assert_partial_state`, since we only make an assertion regarding the two variables `blinkLeft` and `blinkRight`, rather than making an assertion over all state variables.

Finally, Lines 20–21 as well as 25–28 assert that for each millisecond in the time interval, the provided values remain the same, i.e., that the step function does not change output values during that time frame.

As can be seen, we have implemented different C macros to simplify test case development:

- `assert(_partial)_state` which checks if the internal states of ELS and SCS correspond to given assertions. The assertions can specify the state both partially, as done in the listing, and fully.
- `progress_time(_partial)` combines assertions on the state with a progression of time as reported by the time sensor.

Validation Results. As expected, using test-driven development provided the usual benefits:

- having to formulate test cases helped us gain an understanding of the requirements and how they are supposed to work,
- refactoring was made easier and more secure, and
- the implementation was closer to the actual specification from the start.

The fact that we are working with an actual implementation made test-driven development come naturally. However, different ways of combining formal methods with test-driven development have been discussed [Bau04] as well. In addition, developing specifications using continuous testing has been suggested for former ABZ case studies in the context of the B method [HLW+15, HLS+18].

Influences on Code. Using the macros above, our initial design of splitting sensors, user input and actuators did not have to be adapted further to be testable. Yet, it created a vast amount of code entirely dedicated to testing. Of 5223 source code lines (including a Makefile and code used for visualization but not counting comments and blank lines), 3786 lines are test code.

3.5.2. Model Checking Using CBMC

As stated above, we used CBMC to verify properties of our implementation directly on the C code. Depending on where we place C-style assertions, they correspond to different kinds of properties commonly used in state-based formal methods:

- If placed at the end of the loop implemented by the ELS and the SCS state machines depicted in fig. 3.2, assertions correspond to safety invariants that have to hold in every state reachable by one of the subsystems.
- If placed anywhere inside the loop, assertions can be used as invariants on intermediate states.
- If placed outside the loop, we can check if properties hold after a certain number of iterations (controlled by CBMC's unrolling preferences).
- By using additional variables for unrolling state traces, we can implement a lightweight verification of temporal properties. Of course, this is not as powerful as LTL or CTL.

Examplary Verification of ELS-22. Requirement ELS-22 is a great example for an invariant. It states "Whenever the low or high beam headlights are activated, the tail lights are activated, too". For this, we can add an assertion such as:

```
implies(get_light_state().lowBeamLeft > 0,
        get_light_state().tailLampLeft > 0 ||
          get_light_state().tailLampRight > 0)
```

The disjunction in the second part of the implication is important for American cars: as tail lamps are used for indicators, it is accepted behavior if one tail lamp is temporarily deactivated during a flashing cycle. When running CBMC, it immediately came up with a counterexample. A part of the output trace can be found in listing 3.6.

The counterexample shows how the two system variables `ks`, i.e., the key state, and `engine_on`, i.e., the engine's ignition state, change while `light_do_step` is executed.

The main issue with such a counterexample is that each variable assignment, function call and return from a function introduces a new state. While this representation mimics the internal workings of the C code, it does not correspond to the mental model: comparable to common state-based formal methods, we regarded a state change to include multiple variables at once.

Hence, as we were only interested in comparing state variables per full iteration of `light_do_step`, the output was barely readable to us (the counterexample consists of more than 200 lines).

CBMC can optionally reduce the output by removing assignments unrelated to the property. This did not work well for us, as the assignment of signals for the low beam headlights was removed as well. We ended up manually writing state variables in a spreadsheet to comprehend the scenario and ultimately created our own visualization which we will present in section 3.6.2 A (condensed) version can be found in table 3.2. Here, the state changes between two full iterations of our step function are shown, rather than changes of individual variables during the execution. This representation aligned better to our mental model of the implementation and was thus more helpful for debugging.

```
State 59 file light/light-impl.c line 242 function light_do_step thread
   0
----------------------------------------------------
 ks=/*enum*/NoKeyInserted (00000000000000000000000000000000)

State 63 file light/light-impl.c line 242 function light_do_step thread
   0
----------------------------------------------------
 ks=/*enum*/KeyInIgnitionOnPosition (00000000000000000000000000000010)

State 65 file light/light-impl.c line 244 function light_do_step thread
   0
----------------------------------------------------
 engine_on=FALSE (00000000)

State 69 file light/light-impl.c line 244 function light_do_step thread
   0
----------------------------------------------------
 engine_on=TRUE (00000001)
```

Listing 3.6: Partial CBMC Output

The error in our code was that, based on ELS-17, only the low beam headlights were activated due to activated daytime running light. This was not uncovered by the test scenarios, since daytime light was only tested by night, where, coincidentally, other triggers activated the tail lamps.

Verification Results. However, the assertion still failed to verify. Upon further analysis of the property, we discovered a conflict between ELS-22 and hazard blinking in Canadian and US cars. In those cases, hazard blinking deactivates both tails lights for the dark cycle, thus violating the property. We extended our assertion by checking our variable for blinking direction beforehand:

```
assert(implies(blinking_direction != hazard,
               /* old assertion */));
```

Afterwards, we were able to successfully verify the property using CBMC.

Influences on Code. At first glance, using CBMC only required to add assertions to the code. As assertions are often introduced as part of understanding certain scenarios, this does not change the modeling strategy itself. Yet, CBMC comes with a flaw: it is not able to detect integer ranges given by enumerations. This means it frequently finds errors with invalid values for enumerations. As a consequence, one has to add assumptions about value ranges, which cannot be compiled to actual code. Another

Table 3.2.: Example Trace Violating ELS-22. KeyIn = KeyInIgnitionOnPosition.

State Variable	Iteration 1	Iteration 2
key_state	NoKeyInserted	KeyIn
engine_on	FALSE	TRUE
all_doors_closed	FALSE	TRUE
brightness	0	37539
speed	0	936
daytime_light_was_on	FALSE	TRUE
low_beam_left	0	100
low_beam_right	0	100
last_engine	FALSE	TRUE
last_key_state	NoKey	KeyIn
last_all_door_closed	FALSE	TRUE

assumption that needs to be added is that consecutive timestamps cannot get smaller. Thus, for useful verification, some form of conditional compilation is required.

Corrigendum. In preparation of this thesis, I reviewed the verification attempts made in this article. Despite best efforts, the changes above indeed do not suffice for verification with CBMC. It presented a counterexample because *internal state changes* were made transparent for the tool. However, *external state changes* by the environment, e.g., the driver interacting with the pitman arm or toggling the hazard lights, were still left opaque. Indeed, if no user interaction is made possible and not even the engine can be started, there is little to verify.
Fortunately, CBMC is able to recognise that a function allows user interaction possible if the function name is prefixed with `nondet_`. Then, all possible return values are considered based on the type declaration. Surely, this collided with our naming conventions and interfered with readability. After such a refactoring, it was possible to prove `false` as no valid execution path was found anymore. It was necessary to relax assumptions stated via `__CPROVER_assume`. However, we found no way to encode that values must occur in a certain order. For example, the light rotary switch cannot be turned from *off* to *on* directly but must have the *auto* position in between.
Additional re-write of the code was necessary for what would be constants in B, e.g., information on the market code (USA, Canada, EU, ...), whether it is an armoured vehicle or what optional features are installed. Originally, it was intended to initialise these values once and query them if needed. With this approach, using the `nondet_` prefix might change constants on each query. However, CBMC also does not seem to allow non-deterministic choice of function arguments to initialise the values once.
The workaround we implemented is that the functions returning constants are treated as `nondet_` functions. Then, all constants are retrieved once before the main loop

Table 3.3.: Runtime and Memory (Geometric Mean) of CBMC Verification Tasks With Bound n.

Property	Measurement	$n=1$	$n=2$	$n=3$	$n=5$	$n=10$	$n=15$
ELS-14	runtime (sec)	0.73	1.52	2.63	6.84	27.26	72.27
(4 assertions)	memory (MB)	80.66	96.95	114.09	169.10	338.18	511.23
w/ trace	runtime (sec)		1.54	2.75	9.10	41.47	118.33
	memory (MB)		83.13	92.12	133.50	261.81	412.81
ELS-15	runtime (sec)	0.73	1.49	2.60	6.34	26.81	72.58
(1 assertion)	memory (MB)	80.44	97.18	113.78	169.15	341.18	518.05
w/ trace	runtime (sec)		1.36	2.44	6.79	38.87	117.24
	memory (MB)		82.67	91.24	127.87	252.78	400.20
ELS-16	runtime (sec)	0.75	1.47	2.62	6.20	26.66	71.83
(1 assertion)	memory (MB)	80.47	97.13	113.76	169.15	341.15	511.05
w/ trace	runtime (sec)		1.38	2.55	6.98	31.05	94.28
	memory (MB)		82.96	91.52	128.45	246.76	386.80
ELS-18	runtime (sec)	0.74	1.49	2.61	6.29	29.70	69.47
(1 assertion)	memory (MB)	80.45	97.14	113.86	169.18	341.23	518.05
ELS-22	runtime (sec)	0.75	1.48	2.60	6.48	27.46	82.62
(1 assertion)	memory (MB)	80.45	97.16	113.78	169.15	338.23	511.22
ELS-41	runtime (sec)	0.73	1.46	2.54	6.15	25.82	71.28
(2 assertions)	memory (MB)	80.80	97.67	113.40	169.19	338.15	510.97
ELS-43	runtime (sec)	0.73	1.45	2.55	6.33	27.70	77.42
(1 assertion)	memory (MB)	80.44	97.16	113.82	169.16	341.19	517.86
all the above	runtime (sec)	0.76	1.70	3.01	7.04	33.09	87.28
(11 assertions)	memory (MB)	80.31	97.08	114.29	169.22	342.73	518.11

starts by calling the `nondet_` getters. Each call to the getter function had to be re-written to use the local values instead.
Overall, significantly more work is required for verification if done correctly, including additions and changes to the code. This insight impacts our impression that CBMC works right out of the box and that little expertise is required.

Addendum: Performance. Calling CBMC for verification requires that an entry point is specified. Initially, we used the `light_do_step` function that executes a single step. Using this entry point, only a single (high-level) state change is executed, i.e., a bound of $n = 1$ is used and no loop is unrolled. As all state variables are left open (and are valuated via the aforementioned `nondet_` functions), such a CBMC run

already was helpful to locate initial errors. However, this is severely limited, as errors that require two state changes cannot be located.
As an entry point for actual verification, we used the `light_loop` function that simply calls `light_do_step` in a loop. Then, one can specify a bound, i.e., the number of loop iterations that CBMC unrolls for verification.
Table 3.3 contains the runtime (wall time) and memory consumption (maximum resident set size) for different CBMC runs. Assertions were grouped according to the requirement they belong to. The verification of ELS-14, ELS-15 and ELS-16 only fails for a bound $n \geq 2$. In these cases, the data for generating a trace to a counterexample (that is beautified, i.e., a greedy heuristic is used in order to shorten it) is also stated. All benchmarks were executed ten times on an Intel i7-7700HQ CPU. Version 5.6 of CBMC was used. The values in table 3.3 represent the geometric mean (according with Fleming and Wallace [FW86] of all repetitions.
From table 3.3, one can observe that, quite naturally, runtime for low bounds is very fast (< 1 second). The runtime and memory consumption seem to grow exponentially with the bound n. Further, the number of assertions of the verification task only has a small impact on the overall runtime. Requesting a trace to an error seems to cause a significant overhead for larger bounds: for ELS-14 and ELS-15, the runtime nearly doubles. Surprisingly, however, the memory consumption slightly decreases.
Note that CBMC additionally generates an unwinding assertion: only if this assertion is proven by the backend, the property is proven regardless of the bound. For all our benchmarks, the unwinding assertion could not be proven and, thus, even the assertions that could not be falsified are not fully proven.

3.6. Other Observations

Our implementation work allowed us to identify several flaws in the specification as well as shortcomings of our implementation strategy. In the following, we document such issues and give suggestions and solutions.

3.6.1. Specification Ambiguities, Flaws and Suggested Improvements

During development, we identified several shortcomings or ambiguities within the specification. These issues were found during analysis of the requirements and during implementing test cases. As we only performed validation steps after implementation, the validation steps just uncovered shortcomings of our own implementation and non-compliances w.r.t. the specification. Due to page limitations, we will only present some of them:

ELS-37 is somewhat broken or at least highlights an incompleteness in the specification. For now, there is no way to discern whether an adaptive cruise control is part of the vehicle; from the specification, we had to assume that it is installed in every system.

Then, according to SCS-1, there does not even have to be a desired speed. We think that, in order to make sense at all, it rather should be "is active" than "is part of the vehicle". Also, this is the only part of the specification that refers to an **advanced** cruise control.

ELS-42 does not specify what should happen in case of sub-voltage. The only given information is that the adaptive high beam headlight is not available. What should happen remains unclear, e.g., should the manual high beam headlight be used instead?

ELS-19 contains a contradiction: first, it states that ambient lighting *prolongs* already active low beam headlights. Later, it says that the headlamps "remain active or *are activated*". We think that some actions are reasonable to activate the headlight even if it was not on before (e.g., opening the doors). Others definitely should not activate the headlight (e.g., if the brightness falls below the specified threshold, as passing cars and the setting sun might trigger the brightness sensor). Also, it does not have any constraints regarding the light rotary switch: if the switch is in the "off" position, we think the ambient light should not activate at all.

While `currentSpeed` is specified as a sensor in the ELS, it is not clear how the SCS accesses this value. No sensor is provided according to the specification, and only the brake pressure is mentioned as actuator but not the gas pedal. Thus, the SCS as specified appears to only be responsible for determining the desired speed but not for actually deploying it to the current speed? To our understanding, the measured current speed should be a sensor to the SCS to allow it to work properly.

SCS-23 specifies a safety distance for the adaptive cruise control of $2.5\,\mathrm{s} \cdot$ `currentSpeed` when the current speed is below 20 km/h, and a safety distance of 2 m if both vehicles are standing. Assuming `currentSpeed` $< 0, 2.88$ however, the safety distance according to SCS-23 is below 2 m and effectively approaches 0 the closer the vehicle gets to a standstill, e.g., $2.5\,\mathrm{s} \cdot 2.8\,\mathrm{km/h} = 2.5\,\mathrm{s} \cdot 0.77\,\mathrm{m/s} = 1.925\,\mathrm{m} < 2\,\mathrm{m}$. But once a standstill is reached, the safety distance is reset to 2 m and thus violated instantly. It remains unclear whether these 2 m distance is meant as minimum or intended to delay the reaction to eventual acceleration of the vehicle in front.

SCS-28 references a maximum deceleration value, which was only described for the adaptive cruise control in SCS-20 and SCS-21. We assume that it references the same maximum deceleration of 5 $\mathrm{m/s^2}$. It further specifies the acoustic signal which is to be played if the time to reach a standstill with maximum deceleration (5 $\mathrm{m/s^2}$) is greater than the time until impact. This acoustic signal however may overlap with the signal specification given in SCS-21.

3.6.2. Improvements to our Employed Methodology

We are surprised how easy it was to implement the case study in C, given that none of the authors is a professional C developer. While we were unsure during implementation, given our test harness and the results of CBMC, we now have more confidence in the correctness of our implementation. However, CBMC's output was hard to interpret as we discussed above.

Figure 3.3.: OpenGL based visualization

To improve, we created different visualizations. One such visualization is a state visualization based on PlantUML[5] (cf. fig. 3.5). A second visualization was a domain-specific visualization in C++ with OpenGL, using the existing sources directly as part of the compilation. However, development was incomplete and thus omitted for the initial article.

Revisiting the visualizations for this extended article, we found that both are not fully satisfactory: The PlantUML-based visualization is very technical, directly referring to implementation details. While it is still beneficial for understanding test failures, it relies on knowledge about the internal workings of the implementation and is thus not presentable to external stakeholders.

The C++ visualization in contrast provides a domain-specific visualization showing a car and is thus understandable without knowing implementation details. However, as the visualization was directly linked to the C sources, it proved rather inflexible and prone to breakage when implementation details changed. While it was still useful as a mere demonstration tool, its value for development was diminished.

To improve, we decided to further separate implementation and visualization. Our goal was to keep interacting with the implementation simple, but also allow replacing it with a new version straightforwardly.

To achieve this, we added a small sensor implementation, to be controlled from the outside and linked as a shared library. On top, we used F# to develop a visualization using the RayLib library. This approach worked well, reducing the communication of the two components to a simple and somewhat stable C interface.

While the older OpenGL based Visualization looked pleasing, it almost completely omitted numerical feedback. This decision stroke us as too extreme in retrospect, so

[5] `https://plantuml.com/`

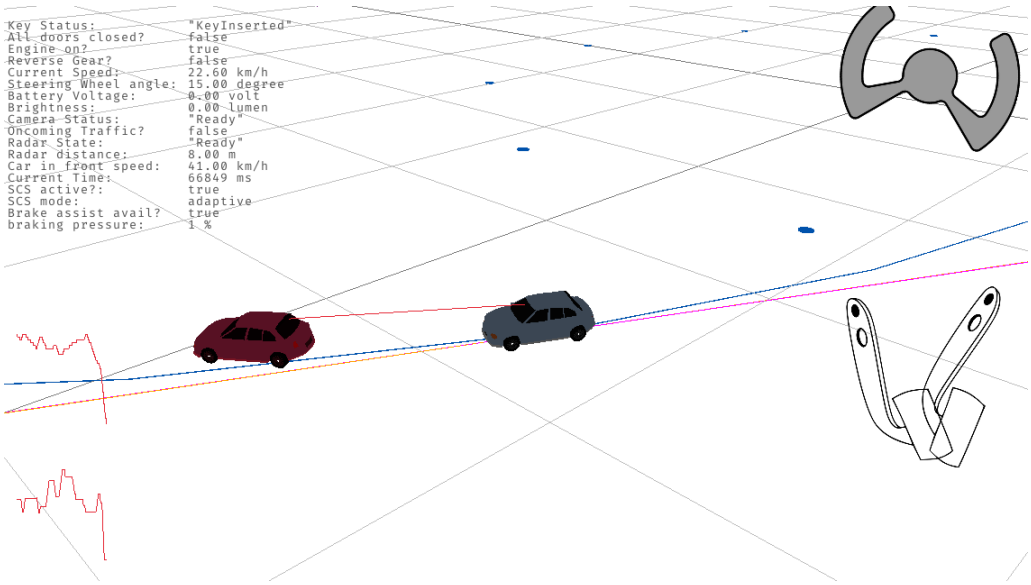


Figure 3.4.: Newly developed RayLib visualisation

we opted for a more minimal look, but including both a domain-specific visualizations and values of state variables. For example the steering angle directly corresponds to the displayed image of a steering wheel as well as to the traced out path the vehicle is currently headed. We found the visuals very helpful in gaining a quick understanding of the implementation's behavior. The geometric intuition provided by the 3D view was a welcome addition.

In contrast to the old visualization, we included an interactive component, enabling the user to experiment and explore the behavior. By default, an automatically animated car will perform a lemniscate around an attractor point. This was surprisingly effective in finding behavior, that doesn't conform with expectations: For instance, when using the pedals some reaction is to be expected. Yet, even though the pedal position indeed changed as seen in the visualization, nothing happened w.r.t. the car's movement. In consequence, we noticed that neither does the gas pedal cause acceleration, nor does the brake pedal decelerate the vehicle. As far as the SCS is concerned, the brake pedal merely disables cruise control as mandated by SCS-16.

Using C for implementation proved very flexible, as there exists a plethora of ways to interact with other languages. Thus, that it would have been easy to use other ways of animation. For instance, we were able to execute our implementation in a browser by compiling it to wasm via the Emscripten Compiler Frontend (emcc) and then interact with it using JavaScript.

3.6.3. Note about Deriving a Software Implementation

As we have started from a low-level implementation in C, the software implementation was always readily available. Hence, in our case, the "model" can be directly compiled and executed. However, testing the executable would still be interesting if we look

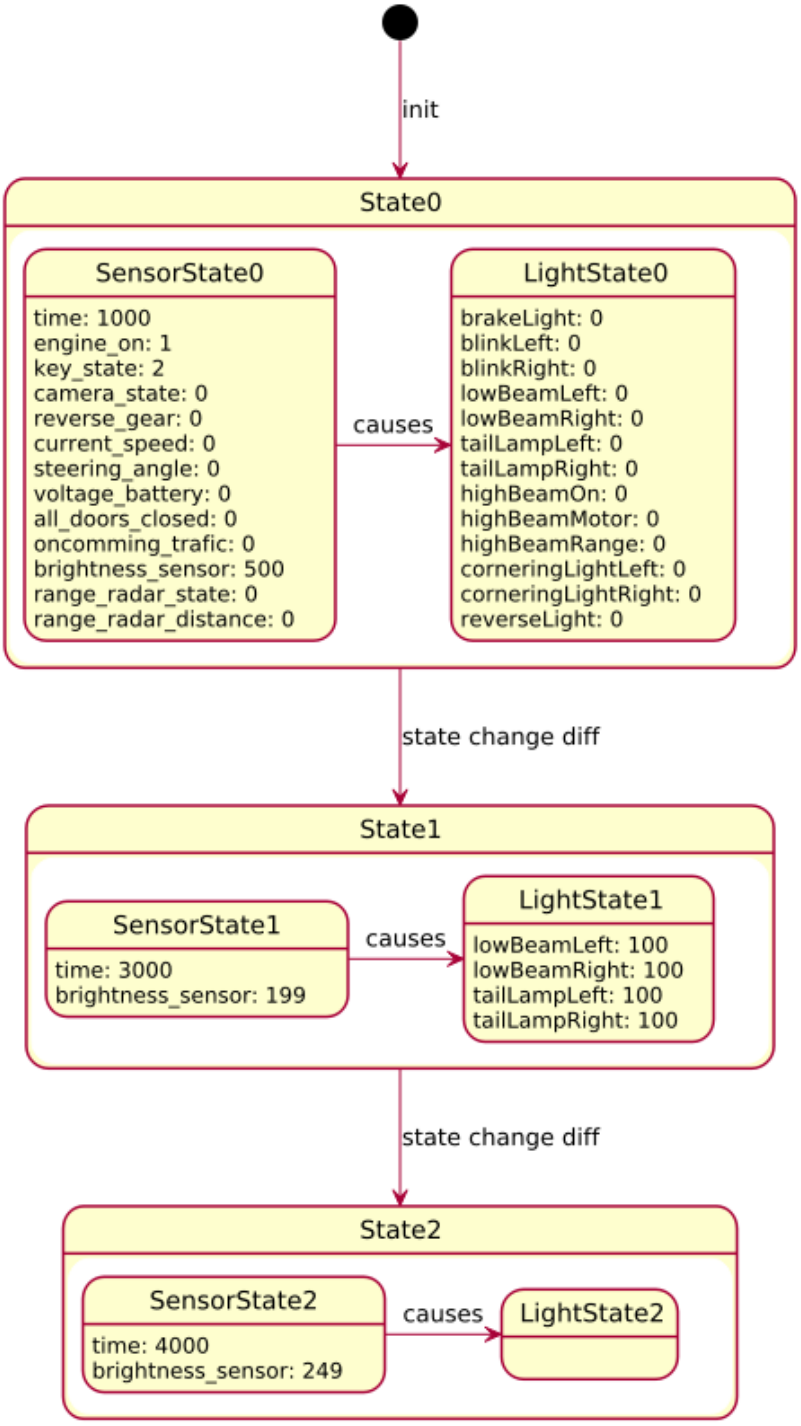


Figure 3.5.: Visualization using PlantUML

beyond our simple tests, i.e., with an actual implementation at hand, hardware-in-the-loop tests would be desirable.

3.7. Comparison

In the 2020 ABZ Proceedings, five other contributions were published, all of them providing verified formal models rather than implementations:

- Arcaini et al. [ABG+20], who utilized abstract state machines (ASMs) [BGR03] and the ASMETA framework [AGRS11],
- Cunha et al. [CML20], who modeled their solution in Electrum [MBC+16], an extension to Alloy [Jac12],
- Leuschel et al. [LMW20], who developed their solution in classical B [Abr96] and later translated to Event-B for proof [Abr10],
- Mammar et al. [MF20, MFL20], used Event-B for two distinct models of the ELS [MFL20] and the SCS [MF20].

The main difference between our approach and the other case studies is that we tried to verify an existing low-level implementation after it has been developed. In comparison, the formal approaches undertaken by the other case studies tend towards following a correct-by-construction approach. That is, they later derive an implementation from the formal model by code generation, e.g., as possible for B [VHKL19].

The case study implemented using ASMs by Arcaini et al. [ABG+20] uses a formalism of a much higher abstraction level than our concrete implementation. However, code generation is available for ASMs as outlined by the authors. As a result, a C++ (rather than plain C) implementation could be derived from the formal models and might look somewhat like our direct implementation.

Furthermore, ASMETA allows doing conformance testing, i.e., deriving unit tests from the formal model rather than writing them by hand as we did.

The actual implementation aside, Arcaini et al. designed their models in roughly the same fashion as we designed our implementation. ELS and SCS are coupled very loosely and are developed as independently as possible. At the same time, they share some signals, comparable to the actuators we defined above. During development, features were added gradually while keeping a (proven) refinement chain intact. While we added features in roughly the same fashion and order, we had no access to a formal proof of refinement. Thus, we had to rely on our test cases entirely.

The approach followed by Arcaini et al. does not verify timing issues, as there is no continuous time in ASM. This is a weakness when compared to our low-level implementation which could be executed in realtime.

The case study performed by Cunha et al. [CML20] follows an approach to verification and validation that is similar to ours. Initially, the authors use test case given as animation scenarios (i.e., small test cases) and reference scenarios (i.e., the execution

sequences given in the specification). This is comparable to the test-driven development we employed as discussed in section 3.5.1.

Formal verification of the requirements is performed after (some of) the development steps. This is again is comparable to our approach of using CBMC on an already existing implementation (cf. section 3.5.2).

The case study by Leuschel et al. [LMW20] models time in the same way we did. There is a dedicated model implementing timers based on elapsed milliseconds. In contrast to our simple clock module shown in fig. 3.2, Leuschel et al. use a more involved timer, supporting deadlines and even triggers.

The case study also uses a domain specific visualization tailored specifically for the case study. Here, using a well-established formal method shows its merits. For B and Event-B, a multitude of animation frameworks and tools is available and can be used without much overhead. In the case study, a visualization tool called "VisB" is used, which allows modifying SVG graphics based on state variables. While this approach is comparable to our visualization, no custom implementation aside from some glue code connecting the image to the state values was needed.

Visualization and timers aside, the B and Event-B models are much more formal than our implementation and rely heavily on proof (by Rodin) and model checking (by ProB) rather than testing.

The two articles by Mamar et al. concentrate on the ELS [MFL20] and the SCS [MF20] individually. A particular focus is on the model's differentiation between the two systems and the environment. The model for the environment closely resembles the sensors and inputs modules shown in fig. 3.2.

Again, the case study uses ProB for model checking. Given that the authors intentionally tried to avoid rather costly LTL model checking and proof, their properties resemble what we check using CBMC. In particular, simple properties on state sequences are rendered model checkable by storing the pre-state in individual variables available for comparison with the current state.

Further Related Work. An alternative to both our immediate low-level implementation and to the code generation approaches that would usually follow with the other case studies is to embed the formal models in runtime code directly, i.e., without compiling them to some other language.

For B, this has been outlined in a demonstration of the ETCS hybrid level 3, where a classical B model is able to control real trains [HLS+18]. The approach uses the ProB Java API [KBD+20] to connect the formal model to the outside world and allows interacting with it.

3.8. Conclusions

To summarize, we have implemented a low-level version of the ABZ 2020 case study in MISRA C, a language commonly used in the automotive industry. We relied on test-driven development for validation as well as on formal verification using model checking.

Compared to case studies using more rigorous approaches, our approach shows both advantages and disadvantages. In particular, our implementation stays close to the actual system, can easily be deployed to an actual car, and could be used for simulation and hardware-in-the-loop tests. Furthermore, due to the popularity of C in the automotive industry, it is more approachable by developer untrained in formal methods.

Corrigendum. As discussed in section 3.5.2, it might be too easy to verify the wrong thing. While the approach may still be more accessible by developers, we have to re-evaluate that a larger amount of training is necessary than assumed initially.

However, we certainly missed the expressiveness and mathematical clarity of more rigorous approaches, as well as having invariants and other properties as first class citizens rather than inserting them artificially via macros and external functions. Compared to a formal method, we were only able to do very lightweight verification of temporal properties and would certainly have favored to be able to model check LTL or CTL properties. Thus, while we were able to verify our implementation to a certain degree, we suspect that a more thorough approach would be able to provide stronger guarantees.

Furthermore, we currently do not validate any properties on time constraints aside from simulating an external clock in the test cases. As part of possible future work, we intend to use CBMC to try to provide real-time guarantees and to verify the correct behavior in presence of scheduling and limited by the actual specifications of an embedded device.

Both could be verified by providing a Verilog model of the hardware, sensors and connections. Afterwards, co-verification of the implementation with the Verilog circuit model can be performed by CBMC [CKY03]. Additionally, we would like to consider other tools that work directly on the C code, e.g., Symbiotic [CVS18] or Klee [CDE08].

Further future reasearch could be done in the combination of formal and informal approaches, e.g., when thinking about code generators: proven invariants on a high-level model could be compiled to C assertions. Then, they could be verified on the low-level code as well, effectively demonstrating the correctness of the code generators.

4. Treating Specifications as Data

Abstract

Considering programs as data enables powerful meta-programming. One example is Lisp's macro system, which gives rise to powerful transformations of programs and allows easy implementation of domain-specific languages. Formal specifications, however, usually do not rely on such mechanisms and are mostly written by hand in a textual format (or using specialised DSL tools).

In this paper, we investigate the opportunities that stem from considering *specifications* as data. For this, we embedded the B specification language in Clojure, a modern Lisp. We use Clojure as a functional meta-programming language and the ProB Java API to capture the semantics of B, i.e., to find solutions for constraints or animate machines. From our experience, it is especially useful for tool development and generation of constraints and machines from external data sources. It can also be used to implement language extensions and to design domain-specific languages.

4.1. Introduction

Formal specification languages are usually employed to gain a mathematical description of problems, algorithms and state machines. Rightly so, the syntax and features of many formalisms is set in stone in order to capture a precise semantics. Thus, e.g., introduction of new operators requires changes to a specific tool-chain, even when they can be mapped to a combination of existing operators. Infrastructure for domain-specific languages and program transformation tools usually is not available. One such model-driven engineering (MDE) methodology is the B method.

An interesting approach is to embed a specification language into a programming language: Examples include αRby [MJ14], which allows writing Alloy [Jac03] constraints in a DSL that generates a Ruby program and an Alloy model. Then, one can interact with (partial) solutions as they are automatically translated into Ruby data structures. However, this approach is less suitable for programatic generation or transformation of models, as Ruby code is not represented by plain Ruby data (and one would have to resort to working with complex internal representations of the code or string concatenation to generate a new program).

We implemented a similar embedding called *lisb* (section 4.3), which embeds the B language in Clojure [Hic20] (a modern Lisp that runs on the JVM) as an internal DSL. In *lisb*, Clojure is used as a meta-programming language due to its rich macro system. We use the ProB Java API [KBD+20] to capture the semantics of B, to construct B

machines under the hood, to find solutions for constraints and for all verification & validation activities (V&V).

Our goal is to investigate the benefits of programmatic construction of constraints and entire B models. Thus, we re-visit the ideas behind Lisp's mantra *code is data and data is code*: In Lisp (and other homoiconic languages), meta-programming is ubiquitous, and generation as well as transformation of programs is mere data transformation. Further, it allows a safe and expressive macro system, which can be used to create domain-specific languages.

In this paper, we aim at transferring these benefits from programming languages to formal B specifications in order to facilitate the work of tool developers (rather than modelling experts). Following, we give a brief introduction to B and Clojure in section 4.2, and introduce the internals of the *lisb* tool in section 4.3. In section 4.4, we present *lisb*'s capabilties for tool development based on an automatic refinement tool. Further, the implementation of a small imperative DSL is described in section 4.5. We also show how *lisb* can be used to fix certain shortcomings of B en passant (section 4.6), in particular addressing the definition system (which is an error-prone C-preprocessor-style macro system) and extending the language with new operators (e.g., a ternary if-then-else operator on expressions).

4.1.1. Motivation

Overall, the development of *lisb* was driven by two experiments of our group:

The initial idea came during implementation of a case study on data validation of university curricula [Sch17, SLW18]. Briefly summarised, the goal was to verify that all combinations of major and minor subjects at the faculties of Arts & Humanities and Business Administration & Economics at Heinrich Heine University Düsseldorf can be studied in a legal standard time (typically six semesters). One idea was to generate conforming timetables from scratch using a constraint-based approach [SLW15].

A first version of *lisb* only covered the B sub-language that contains expressions and predicates (but covered no state changes via variable substitutions) and was created with two design goals in mind: First, to address several shortcomings with the B language — in particular, the lack of convenience operators (such as let and if on the level of expressions instead of substitutions[1]) and an error-prone definition system (see section 4.6.1) that was used to avoid repetition of predicates and substitutions that are shared between several operations — and, second, to interact with (partial) solutions that the constraint solver provides. The main application was to transform the course information obtained from the electronic course catalogue into constraints that can be programmatically manipulated, combined and extended.

Later, another motivator was a student project aiming at translating Solidity contracts [Dan17] to B machines. Thus, it was required to extend *lisb* to cover the entirety of the B language in order to capture state changes. This experiment helped to expose the potential for domain-specific languages. At this point, *lisb* was mature enough so that

[1] During the project, ProB was extended to introduce these operators.

more complex tools can be created on top of it. As mentioned above, an example is a tool that applies certain refinement steps in order to gain an equivalent version of the machine that exposes more information during a specific static analysis (section 4.4).

4.2. Background

In this section, we give a brief primer on the languages involved in the embedding, the B specification language and Clojure.

4.2.1. The B Specification Language

Roughly, the B methodology supports a correct-by-construction approach: Starting with an abstract, state-based model that specifies the desired behaviour, one adds more details by refining the model. Each refinement step is linked to the one before by proof obligations one has to discharge in order to show that the specification did not diverge. The models are written in the B language [Abr96], which is based on first-order logic and set theory. A rather simple B model specifying Peterson's algorithm is given in listing 4.1 (where we prefer standard mathematical symbols over the ASCII notation).

In the `SETS` clause, an enumerated set of statuses is defined (equivalent to enumerated types in programming languages). In the `CONSTANTS` clause, the constant `other` is introduced, which is constrained to the sequence `[2,1]` (equal to the relation $\{(1,2),(2,1)\}$) in the properties clause. The state variables are declared in `VARIABLES` clause and initially assigned during the `INITIALISATION` (note that x is chosen non-deterministically and two possible initial states exist). The state is then manipulated as specified by the guarded substitutions in the `OPERATIONS` clause; here, it is encoded that either process must acquire the mutex before it may enter the critical section. Afterwards, it leaves the critical section again. The safety property that both processes may not be in the critical section at the same time is encoded as part of the `INVARIANT` clause. The invariant is typically verified by proof (e.g., using AtelierB [Lec14a]) or exhaustive model checking (e.g., using ProB [LB08]).

Indeed, such a mathematical notation is very powerful and expressive. Different tools have been created or extended to exploit the high abstraction level in order to concisely capture constraints and perform data validation on large data sets. Examples include ProB [HSL16], OVADO [AV14], PredicateB or DTVT [LBL12]. Further industrial uses are described in [BKK+20].

We built *lisb* on top of the ProB [LB08] toolchain in order to programmatically interact with solutions. ProB is an animator, model checker and constraint solver for the B language that has been used in various industrial settings [BKK+20]. While written in Prolog, a Java API exists to interact with the interpreter core and even to write entire applications that embed B specifications [HLS+18, KBD+20].

```
MACHINE Peterson
SETS Status = {noncrit,wait,crit}
CONSTANTS other
PROPERTIES other = [2,1]
VARIABLES pc,b,x
INVARIANT b ∈ 1..2→ 𝔹 ∧ x∈1..2 ∧ pc ∈ 1..2→Status ∧
       not(pc(1)=crit ∧ pc(2)=crit) ∧
       ∀i(i∈1..2 ⇒ (b(i)=⊤ ⇔ pc(i)=wait ∨ pc(i)=crit))
INITIALISATION
  b ≔ [⊥,⊥] || x :∈ 1..2 || pc ≔ [noncrit,noncrit]
OPERATIONS
RequestCS(Proc) = PRE Proc∈1..2 ∧ pc(Proc)=noncrit THEN
  pc(Proc) ≔ wait || b(Proc) ≔ ⊤ || x ≔ other(Proc) END;
EnterCS(Proc) = PRE pc(Proc)=wait ∧ Proc∈1..2 ∧
                    (x=Proc ∨ b(other(Proc))=⊥) THEN
  pc(Proc) ≔ crit END;
LeaveCS(Proc) = PRE Proc∈1..2 ∧ pc(Proc)=crit THEN
  pc(Proc) ≔ noncrit || b(Proc) ≔ ⊥ END
END
```

Listing 4.1: B Specification of Peterson's Algorithm

4.2.2. Clojure

Clojure is a functional programming language that runs on top of the JVM. It has facilities to interoperate with other JVM languages and, thus, code can call and be called from Java, Scala, Groovy, etc. We chose Clojure over other JVM languages because its strengths complement several weaknesses of B: Clojure offers a rich standard library that facilitates generation and processing of data from disk or other sources; it taps into the rich JVM ecosystem for library and tool support; and, most importantly, the rich macro system simplifies code transformation and empowers development of DSLs. Finally, all of Clojure's data structures are immutable by default (as required for parallel substitutions in B), which also eases transformation, re-combination of and interaction with parts of variable values, predicates or state machines. As Clojure is not a mainstream language, we will briefly introduce some key concepts here.

As a Lisp dialect, function calls are written as lists. For example, the call `(f a b)` is equivalent to calling `f(a, b)` in Java. In nested calls, arguments are evaluated before calling the function.

Homoiconicity — the property that code is written as data structures of the language — gives rise to a rich macro system. Such macros facilitate DSL development (even of entire DSL stack, e.g., [HE10]). In Clojure, macros are like functions, but the evaluation mechanism differs: here, arguments are *not* evaluated when calling the macro. Rather, the macro code re-writes (expands) the arguments into a new expression that is evaluated instead.

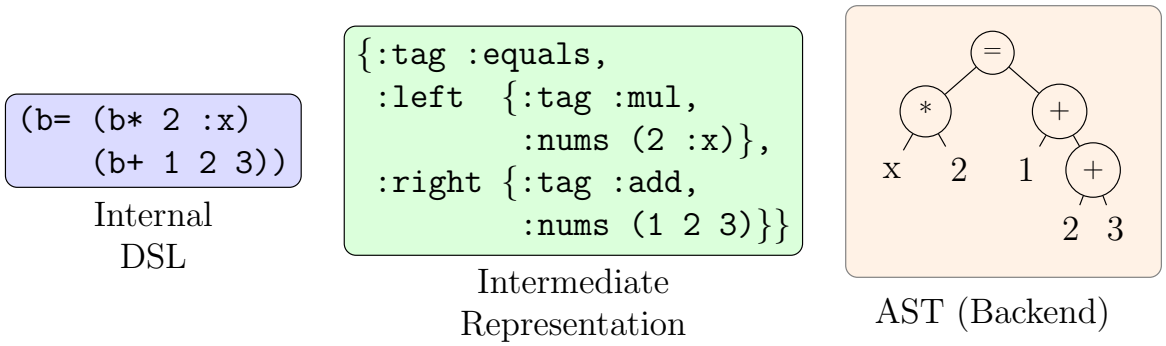


Figure 4.1.: Frontend, Intermediate Representation and Backend

A built-in macro that we use later is the threading macro `->>`. It takes a form and iteratively inserts it as the last argument of all preceding forms. As an example, `(->> a (b c) (d e))` is macro-expanded to `(d e (b c a))`.

4.3. lisb — Internals

lisb is an embedding of the B language in Clojure. Similar to αRby, one can express constraints and B machines in a lightweight internal DSL in Clojure. Under the hood, a B machine is generated and loaded using the ProB Java API [KBD+20]. Using this API, one can calculate and interact with solutions of constraints as well as animate and model check complete B machines.

In listing 4.2, the machine from listing 4.1 is loaded from disk and transformed into *lisb*'s internal DSL. While this language is human-readable, it is less suitable for programmatic manipulation and transformation. Instead, one can translate this internal DSL into an intermediate representation, that trades readability for programmatic processability, simply by evaluating it. An impression of these two languages is given in fig. 4.1 and explained in section 4.3.2. Before we jump into details, we will give an overview of *lisb*'s architecture and different usage scenarios in section 4.3.1.

4.3.1. Architecture Overview

A typical user program that uses *lisb* is situated as depicted in fig. 4.2. The individual components will be discussed in detail in section 4.3.2.

lisb serves as an intermediate layer between a user program and the ProB Java API; ultimately, it generates B models from data in order to interact with them using ProB. The different notations (*lisb*'s internal DSL, and the IR) can be mixed arbitrarily, and each of them can be generated programmatically. Indeed, users can write their own DSLs (User DSL in fig. 4.2) in Clojure (or, in principle, any JVM language). A small example is the algorithm description language presented in section 4.5). The creation of more involved DSLs is also possible, such as object-oriented notations or graphical DSLs (e.g., UML and BPMN). Typically, each construct of the user DSL has to be translated to a

```
1  user=> (require '[lisb.translation.util :as util]
2                  '[lisb.translation.lisb2ir :refer :all])
3  user=> (def lmch  (util/b->lisb (slurp "Peterson.mch")))
4  user=> (clojure.pprint/pprint lmch) ; pretty print, shortened
5  (machine :Peterson
6   (sets (enumerated-set :Status :noncrit :wait :crit))
7   ...
8   (variables :pc :b :x)
9   (invariants
10   (member? :b (--> (interval 1 2) bool-set)) ...
11   (for-all [:i]
12    (member? :i (interval 1 2))
13    (or (<=> (= (fn-call :b :i) true) (= (fn-call :pc :i) :wait))
14        (= (fn-call :pc :i) :crit))))
15  (init (parallel-sub
16         (assign :b (sequence false false))
17         (becomes-element-of [:x] (interval 1 2))
18         (assign :pc (sequence :noncrit :noncrit))))
19  (operations ...
20   (:LeaveCS [:Proc]
21    (pre (and (member? :Proc (interval 1 2)) (= ...))
22         (parallel-sub (assign (fn-call :pc :Proc) :noncrit)
23                       (assign (fn-call :b :Proc) false))))))
24 user=> (def ir (eval `(b ~lmch))) ; generate IR
```

Listing 4.2: Loading the Peterson Machine in *lisb*

lower-level DSL (e.g., the *lisb* internal DSL) or generate the corresponding IR directly. In the following, we aim to give an intuition on how *lisb* can be used in different scenarios.

Specification Transformation Users interested in transforming existing constraints and specifications usually will load a machine and work on its IR. Typically, one would generate a mixture of IR (stemming from the original machine) and a lightweight DSL (such as *lisb*'s internal DSL or an individual one). Because the internal DSL eventually generates IR, one can interleave both representations arbitrarily. Then, one can load the result and interact with it, or save it to a new machine file.

Specification Generation and Domain-Specific Languages If constraints or machines shall be generated from external data sources, one would generate bits of the internal DSL and evaluate it to the IR. Then, one can re-combine these bits of IR programmatically. DSLs would typically be re-written to other, lower-level DSLs or *lisb*'s internal DSL. Technically, it may also directly emit IR code at the cost of readability of the translation rules.

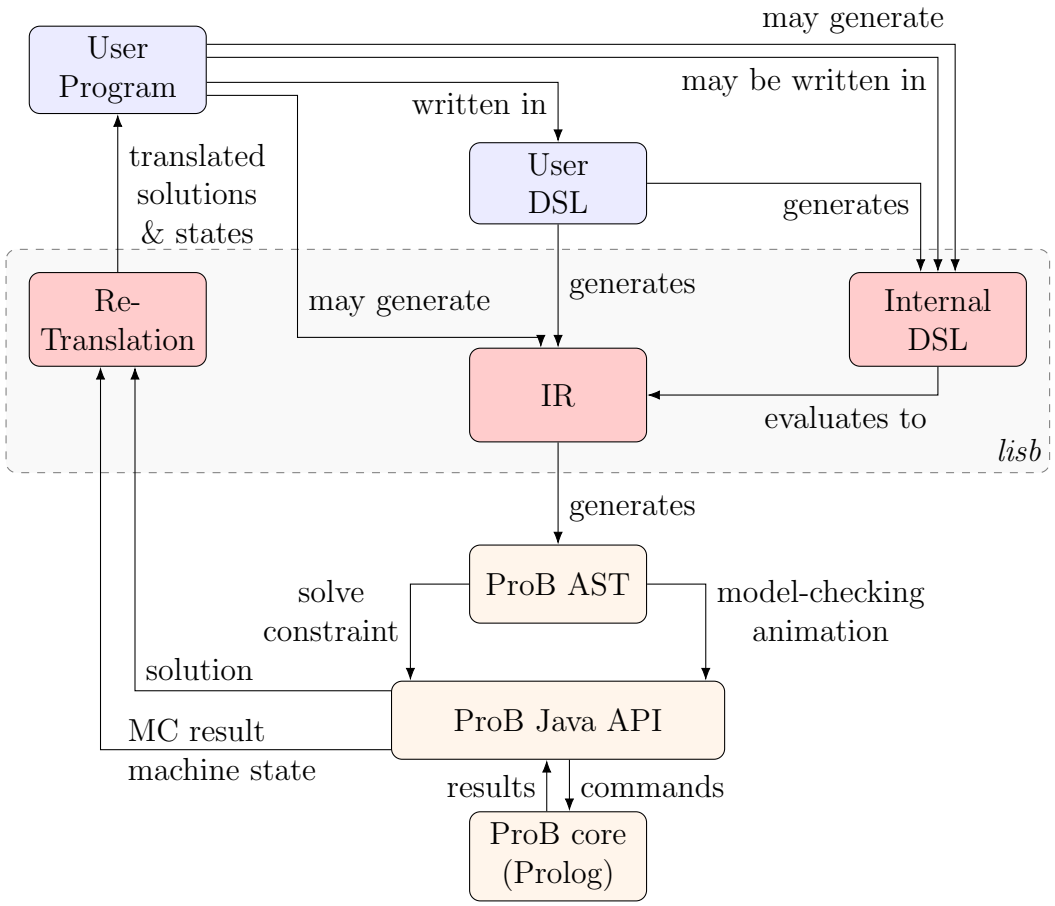


Figure 4.2.: Architecture of a Program Using *lisb* — Arrows Denote Possible Data Flows

Modelling Activities A modelling expert typically would not be interested in using the IR. Instead, they would use a mixture of the internal DSL and, potentially, custom DSLs.

4.3.2. Components

In the following, we will describe the individual components shown in fig. 4.2 in more detail. Afterwards, we put everything together and describe what tools using *lisb* have access to.

ProB Java AST The backbone of *lisb* is the AST library provided by the ProB parser, which is used by the ProB Java API. Its nodes are Java classes and can be instantiated in Clojure using Java interop.

Unfortunately, the Java classes themselves are automatically generated by the parser generator SableCC [GH98] and are not intended to be constructed manually: First, the unwieldy code depicted in listing 4.3 is required in order to create a small predicate such as `x*2=1+2+3` (cf. the *lisb* code in fig. 4.1). Second, since AST nodes are automatically

```
1  new Start(
2    new APredicateParseUnit(
3      new AEqualPredicate(
4        new AMultOrCartExpression(
5          new AIntegerExpression(
6            new TIntegerLiteral("2")),
7          new AIdentifierExpression(
8            Collections.singletonList("x"))),
9        new AAddExpression(
10         new AAddExpression(
11           new AIntegerExpression(
12             new TIntegerLiteral("1")),
13           new AIntegerExpression(
14             new TIntegerLiteral("2"))),
15         new TIntegerLiteral("3")))),
16     new EOF());
```

Listing 4.3: Creating the Java AST for $x * 2 = 1 + 2 + 3$

generated, the usage of many nodes is unintuitive (e.g., a singleton list is required in line 6 of listing 4.3). Third, as the nodes are mutable, it is not possible to insert sub-trees in multiple locations of the same AST and assume that the result is correct. Overall, this AST is neither suitable for transformation nor offers readability.

Intermediate Representation The intermediate representation (IR) is intended for programmatic processing and consequently addresses some issues of the ProB AST directly. It is a pure data representation with the following advantages: First, it avoids encapsulation of the AST's information and, thus, yields a data literal that can be written and accessed without too much boilerplate. Second, we claim that the IR is more intuitive because it is based on the semantics of the actual operators in the language instead of grammar rules used for parsing. Third, as the IR is just data, one may copy & paste sub-trees without worrying to break something. The equivalent IR of the code in listing 4.3 is depicted in the middle box of fig. 4.1.

The IR of scalar values (booleans, numbers, sets and strings) is the corresponding Clojure data literal. Variable identifiers are represented as Clojure keywords (roughly, identifiers prefixed with a colon that stand for themselves). All mathematical operators contained in the B language are represented by maps containing the key `:tag` for identification and an additional key for their operands. The representation of the mathematical sub-language for predicates and expressions is, in principle, agnostic to the formalism that it is compiled to. Machine nodes, e.g., the invariant or operations clause are represented in the same way as operators; however, this representation is B-specific and aligns with ProB's AST nodes.

Internal DSL *lisb*'s internal DSL is built on top of its IR. Contrary to the IR, it is designed for humans to write (parts of) constraints and machines in a Clojure style and

Table 4.1.: Examples of *lisb* Syntax

B (ASCII)	*lisb*	Intermediate Representation	Description
42	42	42	number
`"foo"`	`"foo"`	`"foo"`	string
x	`:x`	`:x`	Variable
{1,2,3}	`#{1 2 3}`	`#{1 2 3}`	enumerated set
NATURAL	natural-set	`{:tag :natural-set}`	set of natural numbers
1 `|->` 2	`(maplet 1 2)`	`{:tag :maplet, :left 1, :right 2}`	tuple
	`(|-> 1 2)`		tuple (alternative)
a + b	`(+ :a :b)`	`{:tag :add, :nums (1 2)}`	addition
a + b + c	`(+ :a :b :c)`	`{:tag :add, :nums (:a :b :c)}`	addition
0 < x & x < 42	`(< 0 :x 42)`	`{:tag :less, :nums (0 :x 42)}`	less than
a : {1,2}	`(member? :a #{1,2})`	`{:tag :member, :elem :a, :set #{1 2}}`	membership
#(x).(x > 42)	`(exists [:x] (< :x 42))`	`{:tag :exists, :ids [:x], :pred {:tag :less, :nums (:x 42)}}`	existential quantification
`MACHINE foo` ...	`(machine foo ...)`	`{:tag :machine, :machine-clauses ..., :name :foo, :args []}`	B machine
`OPERATIONS` ...	`(operations ...)`	`{:tag :operations, :values ...}`	operations clause
RequestCS(Proc) = ...	`(RequestCS [:Proc] ...)`	`{:tag :op, :returns [], :args [], :name :RequestCS, :body ...}`	operation definition
`PRE X THEN Y`	`(pre (lisb X) (lisb Y))`	`{:tag :precondition, :pred (IR X) :subs (IR Y)}`	precondition

to create an abstraction of the intermediate representation. It thus can be regarded as a lightweight DSL for B. Its foundation consists of pure functions that generate the corresponding IR. All operators and machine clauses of B are available in the internal DSL with a one-to-one mapping to AST nodes in B. Examples are `b=`, `b+` or `b*` in fig. 4.1. The operator names are prefixed with `b` in order to avoid name clashes with the default Clojure core functions (so that they can still be used easily). In the context of a seperate `b` macro, function symbols such as `=`, `+` or `*` then are replaced by the functions that generate the IR, and will eventually yield an equality, addition or multiplication node in the Java AST.

An excerpt of the syntax is given in table 4.1.[2] Note that no new semantics is defined; one can map all internal DSL functions directly to corresponding B operators and clauses. In fact, the exact same AST nodes ProB uses to represent B machines are generated. Some operators have multiple aliases (such as generating tuples via `|->` or `maplet`). Others take a variadic number of arguments (such as `<`) to accommodate a Clojure-style of writing predicates. In a pre-processing step, it is replaced by a proper conjunction of predicates.

Re-Translation The re-translation module consists of two parts: first, the ProB Java API returns solutions, states, traces and model checking results as Java objects. While they are directly accessible from Clojure using interop, we also provide a small translation layer that transforms such Java objects back into Clojure data. Infinite sets that are stored as a symbolic value in ProB (such as the set of even numbers), are not represented as Java objects and, thus, cannot be translated. However, one could translate them to a corresponding snippet of IR. Second, ProB's Java AST used for constraints and machines can also be translated into the internal DSL.

4.4. Case Study: Machine Transformation

In the following, we present a transformation of B machines that was required in the scope of another research endeavour. The idea is that certain data refinement rules are applied to an existing B machine in order to yield a new one. Initially, we wanted to integrate this transformation into ProB (in the Prolog kernel) directly; however, various implementation details of ProB (e.g., a complex machine-loading mechanism and source-mapping of AST parts) render such an endeavour challenging. In order to evaluate whether the approach is promising (and worth the hassle to implement it in ProB), a prototype was written in *lisb* instead. Arguably, a functional programming language like Clojure (that integrates almost seamlessly with Java) is also more accessible for software engineers than a logic programming language like Prolog.

Ongoing, there is an attempt to improve the precision of the static analysis of B machines for partial order reduction (POR): the ultimate goal is to locate as many "independent" pairs of operations. Without too many details, the ideas are as follows:

[2] A full overview can be found `https://github.com/pkoerner/lisb/blob/master/doc/Lisb.md`.

```
1 VARIABLES pc1, pc2, b1, b2, ...
2 INVARIANT b1∈𝔹 ∧ b2∈𝔹 ∧ pc1∈Status ∧ pc2∈Status ∧ ...
3 OPERATIONS ...
4 LeaveCS1 = PRE pc1=crit THEN pc1≔noncrit || b1≔⊥ END
5 LeaveCS2 = PRE pc2=crit THEN pc2≔noncrit || b2≔⊥ END
6 END
```

Listing 4.4: Excerpt of Desired Re-Writes

for example, in listing 4.1, the `LeaveCS` operation is statically known to contain *two* operations, one with the parameter $Proc = 1$ and the other with $Proc = 2$. Per definition, every operation "depends" on itself (so `LeaveCS` would be flagged as "dependent"); yet `LeaveCS(1)` can be determined to be "independent" of `LeaveCS(2)`. Thus, it is useful to generate a new operation for each parameter value.

Second, the analysis has to find pairs of operations that can be executed in either order. A syntactical approach is very fast, but is limited to variable access, e.g., `LeaveCS(1)` and `LeaveCS(2)` both access the variables `b` and `pc`. However, there exists a disjoint decomposition of these variables, as both operations write to different "slots" of the `b` and `pc` functions. The idea is to make the slots explicit and assign them to different variables; naturally, all mentions of the original variables have to be re-written as well.

The desired machine for analysis would look like the excerpt in listing 4.4: judging only from the accessed identifiers, one can determine that these operations must be independent of each other, as the read and write sets of the operations are disjoint.

For this transformation, the following features of *lisb* were used:

1. The original machine is parsed and the resulting AST is transformed into the intermediate representation that is susceptible to programmatical manipulation.

2. To find statically finite set variables, the machine is loaded via the ProB Java API and all variables are type checked (e.g., in order to get all values of the `Status` set).

3. For the operation unrolling, parts of the IR (containing slightly re-written guards) are fed to the constraint solver in order to find all solutions for operation parameters.

4. With the information above, the IR is transformed in multiple steps, first generating new operations that eliminate parameters (when possible) and afterwards the re-write of set variables and their usage. Finally, a simplification step eliminates redundant conjuncts, assignments, etc.

5. The ProB Java API's pretty printer is used to emit a new B machine.

Based on *lisb*, a prototype of such a rather involved automatic refinement tool that is capable to work with complex machines (such as the one discussed in [LBH14]) can be

```
user=> (require '[com.rpl.specter :as s])
user=> (defn TAG [t] (s/path #(= (:tag %) t)))
user=> (def CLAUSES (s/if-path (s/must :ir) [:ir :clauses]
                                            [:machine-clauses]))
user=> (defn CLAUSE [name] (s/path [CLAUSES s/ALL (TAG name)]))
user=> (set (apply concat (s/select [(CLAUSE :operations) :values
                                     s/ALL :body :pred :preds] ir))
#{{:tag :or, :preds
    ({:tag :equals, :left :x, :right :Proc}
     {:tag :equals, :left
       {:tag :fn-call, :f :b,
        :args ({:tag :fn-call, :f :other, :args (:Proc)})},
     :right false})}
  {:tag :equals,
   :left {:tag :fn-call, :f :pc, :args (:Proc)}, :right :crit}
  {:tag :member, :elem :Proc, :set {:tag :interval, :from 1, :to 2}}
  {:tag :equals,
   :left {:tag :fn-call, :f :pc, :args (:Proc)}, :right :wait}
  {:tag :equals,
   :left {:tag :fn-call, :f :pc, :args (:Proc)}, :right :noncrit}}
```

Listing 4.5: Retrieving the IR of all Unique Guard Conjuncts from the Peterson Machine (Continues Listing 4.2)

written in about 750 lines of Clojure code[3], of which about 60 lines are devoted to a small DSL. The IR harmonises — because it is plain data — with widespread transformation libraries in Clojure, such as Specter[4]. For example, listing 4.5 shows that one can retrieve the IR of all guard conjuncts (without duplicates) in a few lines of code.

One can generate a modified copy by simply calling Specter's `transform` instead of `select`. Using the path from listing 4.5, one would transform all guards based on an argument function. Naturally, the described refinement tool has to transform more parts of the machine than just the guards.

4.5. Case Study: Algorithm Description Language DSL

Domain-specific languages are increasingly used in order to give domain experts access to formal methods. E.g., Meeduse [Ida20] aims at building proved DSLs for the B method and has recently been applied in the railway domain [ILW+19b, YICD20]; The SafeCap platform [ILR13] provides a graphical DSL that is used to capture railway topologies, their logical structure and signalling rules.

[3]The tool can be found at `https://github.com/JanRossbach/fset`.
[4]`https://github.com/redplanetlabs/specter`

```
procedure(name: "mult") {
    argument "x", "NAT"
    argument "y", "NAT"
    result "product", "NAT"
    precondition "x >= 0 & y >= 0"
    postcondition "product = x * y"
    implementation {
        var "x0", "x0 : NAT", "x0 := x"
        var "y0", "y0 : NAT", "y0 := y"
        var "p", "p : NAT", "p := 0"
        algorithm {
            While("x0 > 0",
                invariant: "p + x0*y0 = x*y") {
                If("x0 mod 2 /= 0") {
                    Then("p := p + y0")
                }
                Assign("x0,y0 := x0/2,y0*2")
        }
        Assert("p = x*y")
        Return("p")
}}}
```

Listing 4.6: Multiplication Example from [CBH+16]

In the following, we give an impression of how to implement a smaller DSL in *lisb*. There have been several implementations of DSLs built on top of the ProB Java API directly. We chose to re-implement an algorithm description language that originally was translated to the Event-B notation [CBH+16] (while Event-B is quite similar to B, it does not contain while-loops or if-then-else statements; see [Leu21]). This DSL is rather limited, containing only:

- procedures and procedure calls,
- variable assignments,
- assertion statements,
- if-then-else statements,
- while-loops,
- sequential composition of statements.

An example algorithm in this DSL that calculates the multiplication of two numbers via repeated addition is depicted in listing 4.6.

Roughly, the corresponding Event-B Machine is generated by inserting a program counter (PC) variable, adding an event for each assignment and two events of each if-statement and while-loop (one guarded by the PC and the condition of the if-statement or while-loop, the other by the PC and the negated condition).

```
(adl :Multiply
     (var :x (in :x nat-set) 5)
     (var :y (in :y nat-set) 3)
     (var :p (in :p nat-set) 0)
     (algorithm
       (while (> :x 0)
         (assert (= (+ :p (* :x :y)) (* 5 3)))
         (if (not= 0 (mod :x 2))
           (assign :p (+ :p :y)))
         (assign :x (/ :x 2), :y (* :y 2)))
       (assert (= :p (* :x :y))))))
```

Listing 4.7: Example Usage of Algorithm DSL in *lisb*

```
(defn assign [pc & kvs]
  {:pc (inc pc)
   :ops (fn [jump?]
          (let [opname (keyword (str "assign" pc))
                newpc (if jump? jump? (inc pc))]
            [`(bop ~opname []
                   (bprecondition
                    (b= :pc ~pc)
                    (bassign ~@kvs :pc ~newpc)))]))})
```

Listing 4.8: Implementation of Assignments in *lisb*'s Algorithm DSL

As a case study, we implemented this DSL (aside from procedure calls to avoid generating multiple machines) in *lisb* that generates the corresponding (classical) B machine. In favour of a fair comparison with the original DSL, we refrained from using while-loops and if-then-else substitutions. We further opted not to use strings containing B formulas like the original algorithm description language: while calling the B parser is trivial, we strive for the ability to hide as much B syntax from the user as possible. The *lisb* version of the example in listing 4.6 can be seen in listing 4.7. Note that, again, both the DSL and the embedded code is plain data that can be generated by a Clojure or Java program.

As an example, we show the function that generates the operation corresponding to an assignment in the algorithm DSL in listing 4.8. It takes the current program counter (PC), which is maintained by an `algorithm` macro, and a number of pairs of variable names and expressions. Internally, a new operation with a unique name is generated, with the only guard being that the PC matches. Aside from the given assignments, the PC is set to a new value. Jumps require that the next PC is passed later by calling a function: the last statement of a then-branch needs to jump behind the else-block (or to the start of a while-loop, etc). Yet, the target PC may still need to be calculated after the operations are generated, because the last PC of the then-branch is required

```
1 DEFINITIONS
2 add(xx,yy) == xx+yy
3 egt(xx) == (∃ yy.(yy ∈ 1..99 ∧ xx < yy))
```

Listing 4.9: Two Suspicious Definitions

to generate the start of the else-branch. Thus, we pass the target PC via the `jump?` argument to finalise the operations.

The entire code for the DSL containing assertions, assignments, while-loops and if-statements that also generates the entire B machine can be written in about 100 lines of Clojure[5]. No additional parser or any other tooling aside from *lisb* is required. Similar to PlusCal [Lam09] and TLA$^+$ [Lam02], *lisb*'s internal DSL that embeds all B expressions may also be used.

4.6. Addressing B-specific Issues

The B language has a so-called definitions system that is based on text replacement (similar to macros in the C language). One could argue that certain (local) transformations and DSLs can be implemented directly using definitions. Below, we examine drawbacks of this system and illustrate why *lisb* offers a cleaner solution.

4.6.1. Language Semantics — Definitions

The definition mechanism of the B language is similar to C preprocessor macros and has similar drawbacks: Actual operator precedences may be misleading and differ based on the tool. Further, it is also possible to capture variables on accident. Below, we present two examples which have been discussed by Leuschel [Leu21] in detail.

Operator Precedences One issue is that the definitions mechanism is interpreted differently by different tools. Consider the first definition in listing 4.9: When calling the definition in AtelierB as `2*add(0,5)`, the result will be 5 because it will be expanded to $2 * 0 + 5$. However, evaluating the expression with ProB will expand the definition to $2 * (0 + 5)$ and 10 will be returned.

Variable Capturing The second definition in listing 4.9 shows the issue of variable capturing. Calling $egt(5)$ will yield true (since $yy = 6$ exists). However, $yy = 5 \wedge egt(yy)$ will result in the rewritten predicate $yy = 5 \wedge (\exists yy.(yy \in 1..99 \wedge yy < yy))$, which is false.[6]

[5] The implementation can be found at `https://github.com/pkoerner/lisb/blob/f22cb5962b87c047f6ab107dcee28f81d3b8aaf0/src/lisb/adl/adl2lisb.clj`.

[6] For such cases, ProB will generate a warning.

```
(defpred add [xx yy] (+ xx yy))
(defpred egt [x] (exists [:y]
                         (and (in :y (interval 1 99))
                              (< x :y))))
```

Listing 4.10: Two Safe Predicates

lisb's Alternative to Definitions *lisb*'s solution to code reuse is the predicate abstraction (`pred`). It is a macro that internally replaces all variables (i.e., keywords) with new variable names. The code snippet in listing 4.10 contains the safe equivalent expressions to the B definitions in listing 4.9. First, the `add` predicate is unambiguous wrt. operator precedence as, highlighted by the parenthesis, the result is an addition that is directly inserted into the AST. Second, the `egt` predicate cannot capture the variable `y` because of the renaming of all prefixes of all local variables with `lisb_`. As an example, the B code $\exists \text{lisb}_{5355}.(\text{lisb}_{5355} \in 1..99 \wedge 5 < \text{lisb}_{5355})$ results from calling $egt(5)$ in *lisb*.

4.6.2. Introducing Convenience Operators

The B language only supports a branching if-then-else construct on the level of variable substitutions. However, it is missing if-then-else on the expression level, e.g., one cannot get the absolute value of an integer by writing `IF x > 0 THEN x ELSE -x END`. During initial development of *lisb*, ProB's dialect introduced support for such an expression, which required changes to its parser and its constraint solver core. We argue that one should be able to define an operator based on the (admittedly unwieldy) tool-agnostic re-writing rule below presented by Hansen [HL12].

$$(\lambda t.(t \in \{\texttt{TRUE}\} \wedge (x > 0) | x) \\ \cup \lambda t.(t \in \{\texttt{TRUE}\} \wedge \neg(x > 0) | -x))(\texttt{TRUE})$$

In *lisb*, one can introduce such a ternary operator easily by simply defining the re-writing rule. The entire implementation of an `ifte` expression and of an absolute value function, which require no further changes to ProB or its parser, is given in listing 4.11.

4.7. Related Work

High-level formalisms have already been embedded into programming languages: as already discussed, we drew inspiration from αRby [MJ14], as well as from PlusCal [Lam09]. However, these tools seem to be more tailored towards modelling experts rather than tool developers who have to examine and interact with specifications on a more fine-grained basis.

The idea of programmatic construction of specifications is not new: solvers such as Z3 [dMB08], Coq [BC10] and also ProB itself have APIs that allow building constraints. *lisb* attempts to hide low-level details (such as creation of suitable types) and provides a more abstract DSL to this end.

```
(defpred ifte [condition then else]
  (fn-call (union (lambda [:t] (and (member? t #{true})
                                    condition)
                                    then))
                  (lambda [:t] (and (member? t #{true})
                                    (not condition)
                                    else)))
           true))

(defpred abs [x] (ifte (> x 0) x (- x)))
```

Listing 4.11: Manual Implementation of if-then-else and its Usage

The Rosette framework employs similar macro-based techniques in order to construct solver-aided domain specific languages (dubbed SDSLs) [TB13]. However, they are used for specification transformation with the goal to gain symbolic computation and partial evaluation capabilities.

4.8. Conclusions

In this paper, we have presented *lisb*, which embeds the B language into Clojure in order to meta-program specifications. While it may be less appealing for modelling experts, as they are confronted with another programming language, *lisb* certainly is a helpful library for rapid tool development.

By embracing the ideas of Lisp and treating specifications as pure data, existing specifications can easily be transformed and new ones can be generated from external data sources. Especially for large datasets, it can be significantly faster and more memory-efficient to avoid parsing a textual representation and to generate the AST programmatically instead. Moreover, Clojure's macro systems provides support for easy creation of domain-specific languages.

Overall, we conclude that formal methods tools will heavily benefit from such a data-oriented approach. As we assume that the majority of formal methods experts does not have a background in Clojure, facilities for generating (parts) of specifications, e.g., a proper macro system, could also be a useful part of formal languages. Alternatively, a structured subset of the macro language could be extracted for a designated DSL design tool.

Embedding into programming languages makes formal methods also more accessible for programmers. Then, an interesting idea is that DSLs could generate parts of a model *and* of an executable program at the same time.

```
(defn dwyer-s-responds-p-between-q-and-r [S P Q R]
  (□ (U (=> (∧ Q (○ (◊ R)))
            (=> P (U (¬ R)
                     (∧ S (¬ R)))))
        R)))
```

Listing 4.12: Definition of an LTL Pattern

4.8.1. Future work

lisb opens doors leading to many directions: first, many higher-level specification languages such as TLA$^+$ or Kodkod share a similar abstraction level. One could use the work of existing translations and add support for their tools or output specifications in other languages. Here, DSLs and pattern matching libraries can help to reduce awkward or inefficient translations by providing constructs closer to a language's idioms.

Second, in constrast to the current focus on model checking, animation and embedding into applications, one could provide a DSL that generates constructs known to work well with provers. This can be useful since many models written to work with animators such as ProB do not work well with proving tools, and vice versa.

Third, one goal of *lisb* is to provide more DSLs so that model extraction from existing software becomes feasible. As demonstrated, constructs such as if-statements and loops can easily be expressed, whereas function calls, classes and interfaces require more complex translations. Polymorphic and recursive functions are known to be particularly challenging to express in B [Leu21]. FASTEN [RGS19, RNM$^+$21] demonstrates the power of entire DSL stacks that can be composed, e.g., support for components with inputs and outputs, contract-based design, and allows unit testing of particular components for test-driven development.

LTL Pattern DSL

Finally, DSLs are also often needed for related formalisms, e.g., LTL. While the formal semantics are clear, nested LTL formulas often become hard to understand for humans. Thus, it may be worthwhile to define certain LTL patterns as well: With a similar approach as presented in this paper, preliminary work suggests that it is straightforward to create a small DSL, e.g., based on the patterns presented by Dwyer et al. [DAC98].

As an experiment, we implemented some of these patterns in the same vein as *lisb*[7]: We build upon ProB's Java AST for LTL formulas, define a small intermediate representation and add a tiny DSL that defines LTL operators as unicode symbols (such as □, ◊, ...). On top of that, we implemented a few of Dwyer's patterns. In listing 4.12, we show the definition of the pattern "S responds to P between Q and R".

Finally, we also added a function that acts as another small DSL layer: the `dwyer` function (see listing 4.13) dispatches based on the pattern (e.g., absence or response)

[7]The proof-of-concept is available at: `https://github.com/pkoerner/lisb/blob/046dfbdce4926064ab39cec25b31b17ac1160a05/src/lisb/translation/ltl/ltl.clj`

```
user=> (dwyer :response "x=1" "y=2")
"□((y=2) => (◊(x=1)))"
user=> (dwyer :response "x=1" "y=2" :before "a=2")
"((y=2) => ((¬(a=2)) U ((x=1) ∨ (¬(a=2))))) U ((a=2) ∨  (□(¬(a=2))))"
user=> (dwyer :response "x=1" "y=2" :between "a=3" :and "b=42")
"□((((a=3) ∨ (○(◊(b=42)))) => ((y=2) => ((¬(b=42)) U  ((x=1) ∨
   (¬(b=42)))))) U (b=42))"
```

Listing 4.13: Generation of LTL Formulas

and the scope (e.g., by default globally, before X, or between X and Y). As the generated Java AST object that can be passed to the ProB Java API offers no readable output, the function returns a pretty print of the generated LTL formula instead.

Acknowledgments

The authors would like to thank Kristin Rutenkolk for her feedback, Jan Roßbach for his implementation of the data refinement tool and David Geleßus for his quick fixes in the ProB toolchain. The first author also thanks David Schneider, Jens Bendisposto and Michael Leuschel for their fruitful suggestions and support.

Part II.

Towards an Improved Partial Order Reduction for B

5. Towards a Shared Specification Repository

Abstract

Many formal methods research communities lack a shared set of benchmarks. As a result, many research articles in the past have evaluated new techniques on specifications that are specifically tailored to the problem or not publicly available. While this is great for proving the concept in question, it does not offer any insights on how it performs on real-world examples. Additionally, with machine learning techniques gaining more popularity, a larger set of public specifications is required. In this paper, we present our public set of B machines and urge contribution. As we think this to be an issue in other communities in scope of the ABZ as well, we are also interested in specifications expressed in other formalisms, for example Alloy, TLA$^+$ or Z.

5.1. Introduction and Motivation

Our group in Düsseldorf has collected since 2003 thousands of B and Event-B machines: our ProB repository contains around 13 000 machines, of which more than 3500 are publicly available. The examples are used for ProB's regression, performance and feature tests. Those public examples contain some duplicates, as they are compiled from different sources: e.g., from tickets in our bug tracker, teaching, literature, case studies, or student projects.

Naturally, not all machines are relevant to all research questions: infinite state spaces might be interesting in order to evaluate symbolic model checking techniques [Kri17], whereas large yet finite state spaces are the important class for distributed model checking [KB18]. Other use cases, such as data validation [HSL16] work by executing a model along one particular, linear path, while others, like constraint solving problems, sometimes work on machines without variables, consisting of a single state. Most recently, machine learning (ML) techniques are applied to model checking or synthesis as well, and require a large number of specifications, e.g., in order to extract and re-combine predicates [DKS19]. Even with access to numerous machines, it is time-consuming and cumbersome to identify machines to use for benchmarking, especially since only a small amount of data can be presented in a typical research article. Without any doubt, other research groups have their individual set of B machines they use for testing and evaluation. Thus, we propose that individual sets of benchmarks from different parts of the community are combined into a global, shared repository. With this paper, we start this

endeavour, and create an index of our specifications as described in section 5.2. Benefits include:

- Benchmarks are publicly available and experiments can be replicated easily.
- Performance comparisons of several tools in different versions can be drawn.
- Suitable benchmarks can be quickly identified.
- Examples for translations between formalisms or ML are available.
- Particularly successful examples can be shared for teaching.

While we are most involved in the B and Event-B community, we think that similar issues are present in other communities which make up the ABZ conference. Thus, we explicitly want to invite everyone to contribute specifications written in other formalisms as well. The repository is located at:

`https://github.com/hhu-stups/specifications`

5.2. Proposed Index

Since our initial set of models is rather large, it is vital that a sufficient amount of meta-information is attached to the models. For this, we suggest usage of edn[1], a serialisation format with parsers available in most mainstream programming languages. For each specification, some basic information should be offered:

- Which formalism is this specification written in?
- A SHA-256 hash code to identify duplicates, and to ensure reproducibility of experiments regarding the specification.
- Number of deferred sets, enumerated sets, constants, state variables and operations / events, number of included machines, etc.
- Number of states and state transitions in the machine (if known).[2]
- Presence of invariant violations, deadlocks, etc. (if the property is known).
- Optional link to another (previous) model (e.g., a correction or evolution).

The information above is known to never change, but can be extended once further properties are considered. Additional information depending on the tool, its configuration or the use case altogether can be included as well, such as temporal properties (e.g., expressed in LTL or CTL) which are expected to hold or to be violated, tool name and version/revision which is able to parse or execute the specification, or settings, walltime and memory usage required for application of a technique such as model checking.

[1] Extensible Data Notation, see: `https://github.com/edn-format/edn`

[2] Note that different tools count the number of transitions and states slightly differently. it might be necessary to keep track of the number of initial states and, e.g., the virtual constants setup states of prob. then, one can derive the expected statistics for other validation tools. some settings can also influence the number of states, e.g., the default scope for deferred sets or maximum number of transitions per operations. in that case, it is preferable not to specify a number of states, but rather include that number in a specific run of the tool (see below), that also includes the settings needed for replication.

```
1  (def META-INF-DIR (java.io.File. "../meta-information"))
2
3  ;; get a sequence of all meta-information files in the directory
4  (def meta-files (remove (fn [file] (.isDirectory file))
5                          (file-seq META-INF-DIR)))
6
7  (defn read-meta-file [f] (read-string (slurp f)))
8
9  (->> meta-files
10      (map read-meta-file)
11      (filter (fn [data]
12                (and (= (:formalism data) :b)
13                     (number? (:number-of-states data))
14                     (> (:number-of-states data) 100000))))
15      (map :file))
```

Listing 5.1: Finding Specifications Based on Their Information

Optional Fields. Naturally, this data must also be extensible via optional fields. For instance, additional information due to a new use case can be gathered, e.g., the amount of states when using state space reduction techniques. As runtime might depend on the hardware it was ran on, relevant data should be included as well. They also allow extension of the information, e.g., for further tools such as Atelier-B [Cle16] or handling of entirely different file formats, e.g., Rodin [ABHV06] archives. In order to select suitable set of specifications, one can simply apply a filter predicate testing the formalism or dialect of it. Furthermore, optinal fields enable links between different machines (e.g., due to refinement or different parameter instantiation) and to external information, such as references to articles describing the model, descriptions of the models as well as the author(s) and their contact information. Finally, certain metrics do not make sense for specific use cases of a formalism, or cannot be applied to other formalisms at all. Thus, such data must not be a mandatory field (but may be mandatory for a given formalism)[3].

Filtering Specification. As previously mentioned, we use edn for the meta information because this format can easily be processed. A short example written in Clojure is given in listing 5.1. There, all files containing meta-information in the directory are located (ll. 1–5). Then, they are read in and filtered (ll. 7–15). The expression starting in l. 9 returns a list of all file names of specifications written in the B formalism that are known to have a state space of at least 100 000 states. At the time of writing, there are 45 such machines. This example shows that finding specifications based on certain criteria is fairly easy and necessary for verification tool maintainers.

[3]It would be sensible to define different standard formats for different formalisms. These can be automatically enforced in a CI pipeline, e.g., by Clojure Spec [Clo], before pull requests are accepted.

Table 5.1.: Overview of available machine meta data with a timeout of 30 min.

Errors on Load			310
Formalism		763 Event-B	2886 Classical B
Deadlock found	1080 yes	1576 no	683 timeout
Invariant violated	255 yes	2498 no	586 timeout
	max	*avg*	*usage in # machines*
States	1 000 002	8743	2624
Transitions	5 570 544	53 296	2624
Included Machines	13	1.18	3339
State Variables	10 000	7.49	2282
Operations	2000	6.00	2497
Deferred Sets	50	0.44	669
Enumerated Sets	19	0.79	1310
Invariants	10 000	9.39	1958
Constants	10 000	8.63	2090
Properties	12 015	17.51	2094
Static Assertions	188	1.46	646
Dynamic Assertions	54	0.20	200
Definitions	374	2.75	1265

Table 5.1 provides an overview of the information of B machines currently present in the repository, compiled after running each machine with a timeout of 30 minutes in the ProB model checker.

On Updating Versions. We strongly argue that the published version of a specification *must not be replaced*. Once they are online, they may be used by any researcher. Even though git clearly documents the history of a file, it would be unclear which version was used as a benchmark or presented in an article. If mistakes were spotted, new versions can be submitted *as a modified copy*.

5.3. Conclusions, Related and Future Work

We firmly believe that a shared repository of specifications will benefit all communities coming together at ABZ. Aside from making benchmarks available for replication, it can assist courses teaching the formal methods. Furthermore, it builds the foundation for exciting new research that relies on such a dataset.

Similar issues have been found in other communities. This led to the creation of central benchmarking sets, e.g., BEEM for models written in DVE [Pel07], or the PRISM benchmark suite [KNP12] for models written in PRISM. Yet, to our knowledge, it is not possible to contribute to these databases. This has led to criticism that, e.g., not many models that are large enough are featured. Also, a fixed set of benchmarks is not

a viable approach in the B community, that creatively uses the B language in order to solve very different types of problems.

In other communities, such as SMT and SAT solving, shared benchmark sets are established for many years [BST10, HS00]. They both grow via community contributions and are the foundation for solver competitions [BDMS05, JLBRS12]. SMT-LIB in particular is a success story, containing more than 100 000 benchmarks. There are many other examples for competitions and problem collections, e.g., SV-COMP[4], TPLP[5] [Sut17], which we cannot exhaustively list here due to page limitations.

An interesting question we could not answer in this paper is to what extent our examples match the reality of (confidential) industrial specifications. An answer requires to take a closer look at the data that is available to us. When considering state space size, number of variables and operations as well as idioms used, e.g., usage of program counters or certain data structures, it might be possible to label some public machines accordingly.

Furthermore, research papers often contain links to download pages not only for benchmarks, but also tools themselves. Some tools presented years ago are hard or near impossible to find now. Some conferences, e.g., POPL, established artifact evaluation committees, yet making artifacts permanently available often is optional. ACM conferences offer different badges[6] depending on availability, replicability, etc. A similar, *mandatory* repository containing at least one binary version or even the source code of tools presented at conferences might prove useful to the research community as well. Worth mentioning here is the StarExec platform [SST14], that allows storage and execution of tools and benchmark problems, which may serve this effort to a satisfactory extent already.

In order for the presented endeavour to be successful, the effort of the entire community is required and their contributions to this repository will be appreciated.

Acknowledgments

Computational support and infrastructure was provided by the "Centre for Information and Media Technology" (ZIM) at the University of Düsseldorf (Germany). We thank the many persons who contributed to the repository (a list is available at the project's website).

[4] `https://sv-comp.sosy-lab.org/2020/`

[5] Which inspired the second author to generate another library, Dozens of Problems for Partial Deduction `https://github.com/leuschel/DPPD`.

[6] Cf. `https://www.acm.org/publications/policies/artifact-review-badging`

6. Interlude: Empirical Evaluation of POR for B

Abstract

Background and purpose: Partial order reduction (POR) is a technique to tackle the state space explosion problem. In low-level formalisms, such as Petri nets, it is known to reduce the amount of states by several orders of magnitudes. Usually, one does not achieve such results for specifications in B. We investigate what improvements are necessary.
Design and methods: We make use of a public repository of B specifications and measure the impact of ProB's POR algorithm.
Results: Around 85 % of the state spaces yield no reduction during deadlock checking, more than 95 % of the machines yield no reduction during invariant checking.
Conclusions: (i) Specialised solvers may be required to determine independence of operations. (ii) Textbook B modelling style may be counterproductive for POR.

6.1. Introduction

In order to evaluate the impact of ProB's partial order reduction, we consider a much larger collection of B and Event-B specifications taken from the repository [KLD20] and compare the state space sizes with and without applying POR. For our experiments, we select B machines with at least two operations in order to have an opportunity for independent events to occur. The set of machines and produced results can be found on GitHub[1].

6.2. Setup

All machines were model checked for 30 minutes (per configuration) with 2 GB of RAM. Each job was allocated a single CPU core of an Intel E5-2697v2 (Ivy Bridge EP) running at 2.70 GHz. A nightly version of ProB 1.11.0 (commit `1b6f14bbd533c2459b1ce675eb57ab24fee89caa`) was used.

[1] `https://github.com/hhu-stups/specifications/tree/aa25fdcecdac58095a1fa9c9e917b524db55e8b7`

6.3. Results

Since we are considering only machines with at least two operations, "only" 1894 B machines from [KLD20] can be analysed. The actual number of results reported per category (deadlock vs. invariant checking) varies due to timeouts and errors, as explained below.

In both cases, the data set is further split in two based on whether an error (deadlock or invariant violation) is found or not. As the state space exploration is halted once an error occurs, we cannot reliably argue about state space reduction in case an error is found. Indeed, the POR algorithm may re-order operations. Thus, even with a deterministic breadth-first-search, this can lead to considerable fluctuations in the runtime, due to finding an error state earlier or later.

Deadlock Checking For deadlock checking, we had to exclude 519 machines that could not be fully checked within the specified timeout of 30 minutes by neither the default settings nor using POR. 17 further machines caused a timeout only when using POR, whereas 25 machines (around 4 % of all machines encountering any timeout) only timed out using the vanilla baseline implementation. We can assume some reduction occurred for these 25 B machines.

After removing 3 machines due to some other error (e.g., CLP(FD) overflows during analysis), 1330 machines are subject to this analysis. Of these, 1121 are deadlock-free, while the other 209 contain a deadlock.

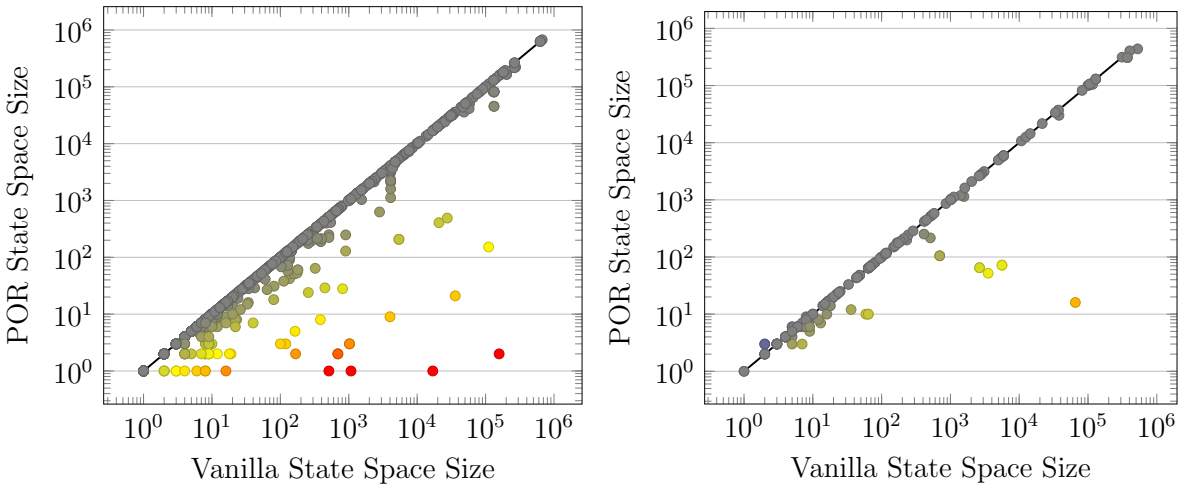
The original and reduced state space sizes are given in fig. 6.1a and fig. 6.1b. All data points on the diagonal correspond to cases where no reduction occurs, while data points below the diagonal correspond to a reduction due to POR. In the right figure (fig. 6.1b) data points can also be found above the diagonal, meaning that here model checking with POR did find the deadlock later than without POR.

Of the 1121 deadlock-free machines, only 191 (17 %) showed some reduction with POR. On average the reduced state space has 54 % of the original size (i.e., a reduction of 46 %) for these 191 machines. The median is 56 % of the original size. Similar, of 209 machines containing deadlocks, we can observe 36 (17.2 %) with a reduced state space and 9 with a larger one (as discussed earlier).

Thus, even when adding the 25 machines with timeout when not using POR above, we have less than 20 % of models where POR reduces the state space.

Invariant Checking For invariant checking, we can analyse 1385 machines after excluding 452 where both model checking algorithms time out, 2 machines that only time out with POR and 55 machines that only time out without POR. Again, we can assume some reduction for the latter cases (around 11 % of all machines featuring any timeout). Further, we exclude 55 additional machines due to other occurring errors, such as well-definedness issues or overflows in the CLP(FD) backend. This leaves 1331 machines to analyse here.

1169 machines preserve the invariant. The (reduced) state space sizes are visualised in fig. 6.2a. Of these, we can observe a state space reduction in 37 machines (3.2 %).



(a) Deadlock-Free Models

(b) Deadlock-Containing Models

Figure 6.1.: (Reduced) State Space Sizes for Deadlock Checking

On average the reduced state space has 76 % of the original size (i.e., a reduction of 24 %) for these 37 machines. The median is 86 % of the original size. Unexpectedly, a single outlier lies above the diagonal, i.e., yields a larger state space with POR. This is a machine that acts as a test for ProB's randomisation library, and hence the state space can change with each run. Even when assuming that all 55 machines with timeout produce a reduction, we have a reduction in less than 10 % of cases.

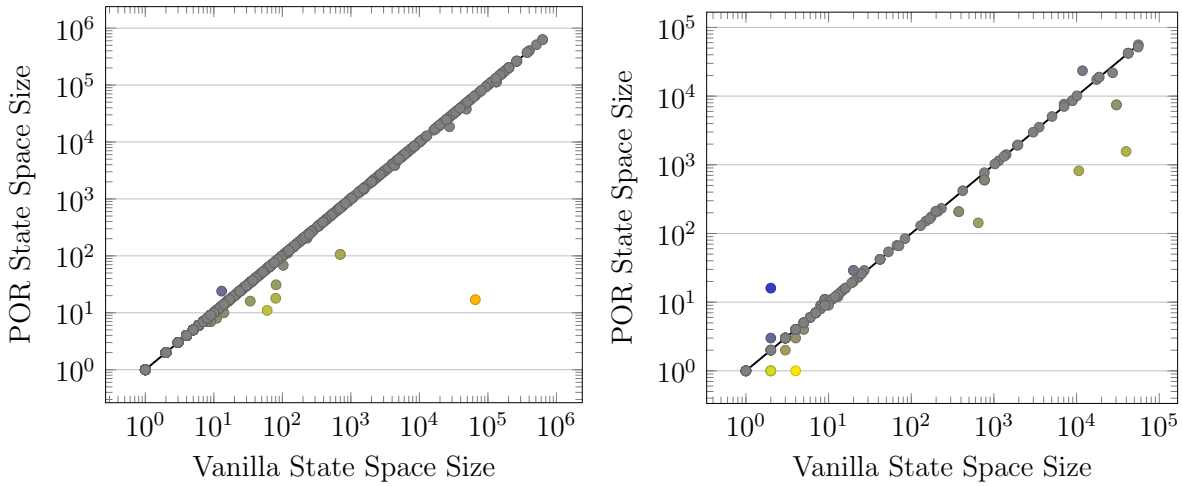
Of 162 machines with invariant violations (fig. 6.2b), we observe 30 machines (18.5 %) with a reduced state space and 18 with a larger one.

6.4. Threats to Validity

One needs to address the following points:

- Many machines time out and are excluded. This could in principle include large machines that are more suitable for POR. Some timeout is needed as many of these machines have an infinite state space. Due to the exponential state space explosion problem, a larger timeout will only include a few additional machines. From our sample, we can also observe the trend that smaller machines exhibit state space reductions more often (cf. figs. 6.1a and 6.2a). Further, we inspect issues that are more prevalent in larger machines that hinder POR in section 7.5.

- The set of machines may not be representative, as it includes many examples from literature, small machines used for teaching, different versions or instantiations of the same machine, etc. In particular, it does not contain larger, confidential machines from industry. From our experience, POR does not work well for these machines. The bias may even be *towards* machines well-suited for POR, as it includes several models meant for testing of the algorithm.



(a) Invariant-Preserving Models

(b) Invariant-Violating Models

Figure 6.2.: (Reduced) State Space Sizes for Invariant Checking

- Note that machines are excluded that time out using regular model checking but not when using POR. However, this rarely occurs and does not significantly impact our conclusions, as the vast majority of models still does not exhibit any reduction.
- Note that we cannot present reduction of memory consumption or runtime in detail here. Due to the additional analysis phase and overhead during runtime, POR may be slower even if it reduces the state space. The reader is invited to inspect the full data set made available on GitHub (footnote 1).

Acknowledgments

Computational infrastructure and support were provided by the Centre for Information and Media Technology at Heinrich Heine University Düsseldorf.

7. Towards Practical Partial Order Reduction for High-Level Formalisms

Abstract

Partial order reduction (POR) has considerable potential to reduce the state space during model checking by exploiting independence between transitions. This potential remains, however, largely unfulfilled for high-level formalisms such as B or TLA$^+$. In this chapter, we report on our experiments regarding POR: We analyse why POR fails to achieve reductions and identify minimal examples without reduction that make use of high-level constructs in B, and provide several new ideas to make POR pay off for more complex formal models. A proof-of-concept implementation then yields two orders of magnitude reduction in the state space for a particularly challenging case study, a railway interlocking model that escaped our POR techniques thus far.

7.1. Introduction

Partial order reduction (POR) [God90, Pel93, Val89] is a technique to tackle the state space explosion problem in model checking [CGP99]: Instead of executing all interleavings of independent behaviour, only one is explored in the best case. In *low-level formalisms*, such as Petri nets or Promela, and in process algebras like CSP or mCRL2, POR is known to reduce the state space by several orders of magnitudes [BJL$^+$19, GRHRW15, Hol97, LPVDPH16].

In contrast, the application of POR to *high-level formalisms* like TLA$^+$ [Lam02] or B [Abr96, Abr10] has been disappointing thus far. Attempts at using POR for TLA$^+$ using TLC [YML99] were not successful and abandoned[1]. POR has also been implemented for B using the ample set approach within ProB [DL14, DL16, Dob17]. While considerable reduction can be obtained for some specifications, the technique does not seem beneficial for real-life examples. Another attempt of using POR for B was made using LTSmin together with ProB [BvdPW10, LPVDPH16]. It uses ProB to solve predicates and calculate the next states while POR is provided by LTSmin. LTSmin's approach to POR is based on the stubborn set theory [Val89] and works well for *low-level* formalisms.

[1] Private communication from Stephan Merz to Michael Leuschel at Schloß Dagstuhl; see also the presentation by Kuppe [Kup18].

```
MACHINE NoReduction
VARIABLES xx, locked
INVARIANT xx ∈ POW(1..2) ∧ locked ∈ 𝔹
INITIALISATION xx ≔ ∅ ‖ locked ≔ ⊥
OPERATIONS
add(yy) = SELECT locked = ⊥ ∧ yy ∈ 1..2 ∧ yy ∉ xx
          THEN xx ≔ xx ∪ {yy} END;
lock    = SELECT locked = ⊥ THEN locked ≔ ⊤  END;
unlock  = SELECT locked = ⊤ THEN locked ≔ ⊥ END
END
```

Listing 7.1: Adding a Value Into a Set — No Reduction

```
MACHINE HasReduction
VARIABLES xx_1, xx_2, locked
INVARIANT xx_1 ∈ 𝔹 ∧ xx_2 ∈ 𝔹 ∧ locked ∈ 𝔹
INITIALISATION xx_1 ≔ ⊥ ‖ xx_2 ≔ ⊥ ‖ locked ≔ ⊥
OPERATIONS
add_1  = SELECT locked = ⊥ ∧ xx_1 = ⊥ THEN xx_1 ≔ ⊤ END;
add_2  = SELECT locked = ⊥ ∧ xx_2 = ⊥ THEN xx_2 ≔ ⊤ END;
lock   = SELECT locked = ⊥ THEN locked ≔ ⊤  END;
unlock = SELECT locked = ⊤  THEN locked ≔ ⊥ END
END
```

Listing 7.2: Unrolled and SAT Encoded Version of listing 7.1 — POR is Successful

Compared to ProB's approach in [DL14, DL16, Dob17], the approach of LTSmin is more fine-grained (wrt. guards), yet rarely achieves (mostly slightly) better reduction for B models[2]. Overall, POR rarely seems worth the effort for practical B models.

This chapter takes a closer look at deadlock checking B models with POR; The main insight we gained is that static analysis of a model (before model checking) often does not determine a precise enough independence relation. The techniques described in the rest of this chapter focus on POR for deadlock checking (as effectiveness is already low and LTL model checking requires even more constraints): Many B models contain operations drawing a parameter from a known finite set; such operations are treated as a unit and, thus, independence between certain instances cannot be captured. We propose to *unroll* such operations by replacing them with a new operation for each parameter (section 7.3). Additionally, operations that access a shared set variable usually only interact with a small subset of its elements. We discuss benefits and drawbacks of a constraint-based analysis as well as encoding sets to SAT variables before applying a syntactical analysis (section 7.4).

[2] Already the results in section 4.3 and table 3 of [KML18] for POR were unsatisfying. Other techniques of LTSmin were very effective, however.

As an example, the model in listing 7.1 can (automatically) be re-written to an equivalent model depicted in listing 7.2 by *unrolling* the `add` operation and encoding the set `xx` as booleans. The former model yields no state space reduction using ProB's POR, whereas the latter one does. Though some specifications may require additional re-writes or more involved analysis techniques, the combination of these two techniques allows state space reduction by POR on large, real-world models. In section 7.5, we share key insights based on a grand challenge we set ourselves, a large model with many real-world features whose state space should be significantly reduced using POR, yet escaped our approach so far. With the techniques above, the expected reduction occurs.

7.2. Background

The B-Method

[Abr96] and its successor Event-B [Abr10] are methodologies that rely on a correct-by-construction approach, i.e., an abstract specification is proven correct and is iteratively refined as more details are added. Proofs accompany all refinement steps, linking each iteration to the ones before.

Both B and Event-B have seen particular use in the railway industry [BKK+20]. While the former focuses on software development, the latter is designed for modelling systems. Event-B is most commonly used via the Rodin toolset [ABH+10], and exported proof information can be used for model checking [BL09]. B and Event-B are very expressive, encompassing first-order logic with (higher-order) sets, sequences, functions, relations and records. Both formalisms are state-based with (possibly non-deterministic) initial assignments of constants and state variables, and guarded transitions (named operations in B and events in Event-B)[3] yielding successor states. A state of a B model is composed of values for all the constants and variables of the model.

While we study both B and Event-B models, we will use the term operation to denote both B operations and Event-B events. Small examples of a B specification are given in the motivating example in listings 7.1 and 7.2. B machines might include additional clauses such as the `CONSTANTS` clause (that declares identifiers of constants similar to the `VARIABLES` clause), the `PROPERTIES` clause (constraining the constants) or the `SET` clause (that contains, e.g., enumerated sets). While the following concepts of operation and operation instance are related, it is important to distinguish between them:

Notation. *An* operation *is the name of a guarded substitution (aka statement) that may be parameterised. E.g.,* `add` *or* `lock` *in listing 7.1 are operations. The guarded substitution is also called the* body *of the operation.*
An operation along with values for all its parameters is called an operation instance. *E.g.,* `add(1)` *is an operation instance. Another one is* `add(2)`.

An operation instance is thus a transition label.

[3] Or actions in TLA+.

ProB

[LB03, LB08] is an animator, model checker and constraint solver for the B language. It is written in SICStus Prolog [CM12b] and its constraint-solving backend makes use of coroutines and the CLP(FD) library [COC97]. Alternative backends are available via translations to SAT and SMT: the work of Plagge and Leuschel [PL12] uses the Kodkod [TJ07] library to translate B to SAT, while the works of Krings, Schmidt and Leuschel [KL16, SL21] translate B to SMT for using Z3 [dMB08] as a solver.

Partial Order Reduction

POR [BK08, Pel93, Pel94] is a model checking technique that only explores a subset of the state space. POR is considered to be appealing because, for n independent operation instances, one has to explore (in the best case) only a single ordering rather than $n!$ many. Thus, exponential reductions are possible in concurrent systems that synchronise on few events. While the underlying idea seems simple, the conditions to ensure correctness are intricate[4].

POR exploits *independent* operation instances: Two operation instances are independent, if they can be performed in any order without changing the resulting state. This is visualised in fig. 7.1: If α and β are independent and simultaneously enabled in the original state space, this implies that β can be executed after α and vice-versa, and the resulting states are identical. In short, this is the case if the operation instances commute and do not disable each other.

Below, we will give a more formal definition. Note, as is usual when presenting POR, we assume that operation instances are deterministic, i.e., given an operation instance α and a state s there is at most one successor state s' such that $s \xrightarrow{\alpha} s'$.[5]

Notation (Enabling Predicate)**.** *For an operation e, we define en_e to be its enabling predicate (its guard) that is evaluated over a state s.*

Definition 1 (Independence)**.** *Two operation instances α and β are independent, if the following constraint holds. Otherwise, they are dependent.*
$\forall s, s_1, s_2 : en_\alpha(s) \wedge en_\beta(s) \wedge s \xrightarrow{\alpha} s_1 \wedge s \xrightarrow{\beta} s_2 \implies \exists s' : en_\beta(s_1) \wedge en_\alpha(s_2) \wedge s_1 \xrightarrow{\beta} s' \wedge s_2 \xrightarrow{\alpha} s'$

The operation instance `lock` depends on `add(1)` (and vice versa, as the independence relation is symmetric), because performing `lock` may (and will) disable `add(1)`. The operation instance `add(1)` is independent of `add(2)`.
Usually, one approximates the independence relation during static analysis before model checking based on operations. Two operations are independent if all respective operation instances are independent.

[4]For example, an error in a twenty-year-old algorithm was recently discovered [Sie19].

[5]For Event-B it is straightforward to lift all non-determinism into parameters. In Classical B this is more difficult; but the formalisation of independence with non-determinism would make the presentation overly complex and detract from the main points of the chapter.

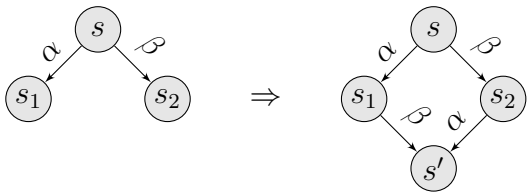


Figure 7.1.: Visualisation of the Operation Independence Definition

As an example, the operations `add` and `unlock` are independent of each other because they write different variables (and the read in the guard of `add` of `unlock` is not conflicting)[6].

The Ample Set Approach

As the POR implementation in ProB relies on the ample set approach[7], we introduce it more formally. For this chapter, it is not necessary to understand *why* POR works in detail, but only *what information* is required.

By $op(\alpha)$ we denote the operation associated with an operation instance α. We also define the enabled operations in a state s by $enabled(s) = \{op(\alpha) \mid \exists s' : s \xrightarrow{\alpha} s'\}$.

An ample set is a subset of enabled operations in a state (referred to as s in the following formulas) that are considered by model checking. In other words, all operation instances for operations not contained in the ample set are ignored. For example, in Figure 7.1, we could choose $ample(s) = \{op(\alpha)\}$ and thus ignore β in s. To reach a sound reduction of the state space, one requires the following conditions to hold (taken from [Dob17]):

(A 1) **Emptiness Condition:** $ample(s) = \varnothing \Leftrightarrow enabled(s) = \varnothing$

(A 2) **Dependence Condition:** Along every finite path in the original state space starting at s, an operation dependent on $ample(s)$ cannot appear before some operation $e \in ample(s)$ is executed.

The conditions (A 1) and (A 2) suffice for deadlock checking; LTL model checking (which is used for invariant checking) has additional conditions (stutter and cycle), yet those are out of scope for this chapter. In ProB's implementation, two local criteria are used instead of (A 2). They have been proven correct in [DL16, Dob17]:

(A 2.1) **Direct Dependence Condition:** Any (ignored) operation $e \in enabled(s) \setminus ample(s)$ is independent of all operations in $ample(s)$.

(A 2.2) **Enabling Dependence Condition:** Any (disabled) operation $e \in Events \setminus enabled(s)$ that depends on some operation $f \in ample(s)$ *and* is possibly co-enabled with f may not become enabled by execution of operations $e' \notin ample(s)$.

[6] More precisely, all operation instances of `add` are independent of `unlock` because they can never be enabled at the same time.

[7] The implementation in LTSmin uses stubborn sets. There is not much difference concerning our argument as the analysis must extract mostly the same information.

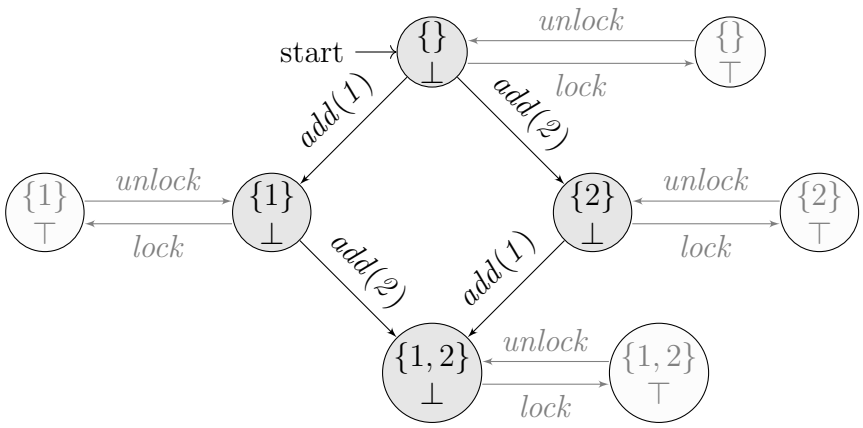


Figure 7.2.: State space of the machine in listing 7.1. Each state consists of the set `xx` (at the top) and the boolean `locked` (at the bottom). The commutativity of the *add* operation instances is highlighted.

Two operations are considered to be possibly co-enabled if there exists a state s in which both guards are satisfied. Note that such a state may not be reachable.

Thus, in practice, the independence relation, an enabling relation and a "may be co-enabled" relation between operations are approximated during a static analysis phase (which we will refer to as *POR analysis*).

7.3. Idiom 1: Parameterised Operations

ProB's partial order reduction and the POR analysis identifies operations by their name. However, there may be several operation instances, i.e., combinations of a name and concrete parameter values. A trivial example is part of listing 7.1.

From a high-level point of view, this machine has three operations where only `add` and `lock` can be enabled simultaneously but are dependent. Thus, the state space cannot be reduced. Yet, the operation instances `add(1)` and `add(2)` satisfy exactly our definition of independence (fig. 7.1), as `add(1)` and `add(2)` commute (see fig. 7.2)!

In this example, the independence of some operation instances within the same operation is not exploited. In many cases, certain operation instances of *one* operation are independent of certain operation instances of *another* operation. An example is described based on our grand challenge in section 7.5.2.

7.3.1. Solution: Unrolling of Operations

The example above has one important property: for the considered operation `add`, we can statically determine a finite set of possible values for the parameters (i.e., either $yy = 1$ or $yy = 2$). In this case we can replace the *operation* with all its *operation instances*, by hardwiring the parameter values. For the example above, this gives rise to two operations `add_1` and `add_2` in listing 7.3.

```
1 OPERATIONS
2 add_1  = SELECT locked = ⊥ ∧ 1 ∉ xx THEN xx := xx ∪ {1} END;
3 add_2  = SELECT locked = ⊥ ∧ 2 ∉ xx THEN xx := xx ∪ {2} END;
4 lock   = SELECT locked = ⊥ THEN locked := ⊤  END;
5 unlock = SELECT locked = ⊤  THEN locked := ⊥ END
```

Listing 7.3: Unrolled `add` Operation

Advantage: Necessary Preprocessing

This technique is the bare minimum to locate independence between operations that share at least one variable. Thus, it is the foundation for the techniques below.

Drawback: Infinite Sets

This unrolling technique is not always applicable given that parameter choices for all states have to be considered. Indeed, the calculation of all possible parameter values may be expensive and yield a large or infinite number of values (due to an overapproximation by the static analysis).

Drawback: Multiple Evaluations

While unrolling an operation may be suitable for POR analysis, it duplicates the majority of sub-expressions. Each operation is considered individually in ProB, and shared sub-expressions have to be re-evaluated which results in a slow-down during model checking.

Below, we assume that all operation instances are unrolled. Thus, there is no difference between the concepts of operation and operation instance and their independence. In case an operation cannot be unrolled, it is retained as-is and syntactic independence can still be determined.

7.4. Idiom 2: Usage of Compound Values (Sets, etc.)

With the simple unrolling technique above, we have established that the POR analysis could now in principle spot the independence between operation instances. In practice, the POR analysis in ProB will, however, *not* determine the independence if two operations write to the same variable.

For performance reasons, the POR analysis focuses mostly on syntactic aspects in order to yield a fast approximation[8]. It considers the (**a**ction) **r**ead and **w**rite sets of two operations (AR_1, AR_2, R_1, R_2, W_1 and W_2). A variable is contained in the action read set AR of an operation, iff the substitution reads it; in the read set R iff the guard or the substitution reads it; and in the write set W iff the variable is written to. The

[8]Which is precise enough for some formalisms (at least using LTSmin's POR), but not for others [LPVDPH16].

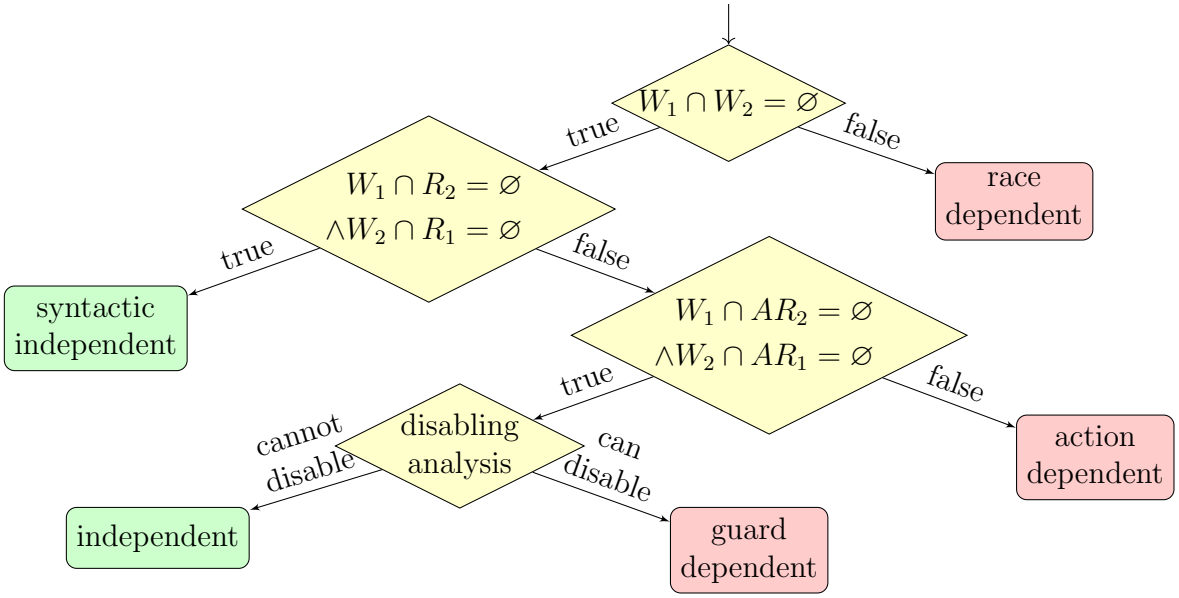


Figure 7.3.: Syntactically Determining the Independence Relation of Two Operations

POR analysis then follows the flowchart depicted in fig. 7.3, where only the disabling analysis uses semantic aspects.

If we re-consider the operations in listing 7.3, we can observe that both `add_1` and `add_2` write to the same variable `xx`. Obviously, the intersection of the two write sets $W_1 \cap W_2$ is not empty and a syntactic POR analysis yields that the two operations are (race) dependent. Yet, set union is associative and commutative and the operations *should* be classified as independent because $(xx \cup \{1\}) \cup \{2\} = (xx \cup \{2\}) \cup \{1\}$.

A summary of a combined approach can be found in Appendix 7.A (listing 7.5).

7.4.1. Solution 1: Constraint-Based POR Analysis

Since the original syntactic approach depicted in fig. 7.3 does not suffice, we added a new constraint-based semantic approach. Instead of syntactically classifying a pair of operations as race or action dependent (see fig. 7.3), we use a constraint solver (ProB, Kodkod or Z3) during the POR analysis. Below, we present how we determine operations to be independent by considering non-disabling and commutativity constraints separately (see Def. 1). Further, in order to be able to check (A 2.2) on the fly, we also use constraints to determine which other operations may (not) be enabled by a specific operation. Finally, again for (A 2.2), one also has to determine which operations may be co-enabled. For the overall approach, we use the notion of before-after predicates and enabling predicates:

Notation (Before-After Predicate)**.** *For an operation instance e, we define $BA_e(s, s')$ to be the before-after predicate. It is a conjunction of the guard of operation $op(e)$ and the predicate whose solutions s' form the successor states of s using e.*

As an example, the before-after predicate for the operation `add_1` is[9]:

$$BA_{add_1}(s, s') \equiv \underbrace{locked = \bot \wedge 1 \notin xx}_{en_{add_1}(s)} \wedge \underbrace{xx' = xx \cup \{1\} \wedge locked' = locked}_{\text{substitution of } add_1}$$

Before-after predicates do not exist for all operations, e.g., those containing a WHILE-loop.

Non-Disabling Constraint

Independent operations must not disable each other and commute. The constraint below checks whether operation α can disable the operation β. The conjunct *Info* might contain additional information, such as the values of constants, proven theorems or (parts of) the state invariant. Also note that the states s and s' may not be reachable in the state space, and, thus the following computes a (safe) approximation of disabling:

$$\exists s, s'.(Info \wedge en_\beta(s) \wedge BA_\alpha(s, s') \wedge \neg en_\beta(s'))$$

For example, to check whether `add_1` may disable `add_2`, we have to consider the constraint:

$$\begin{aligned}\exists s, s'.(Info \wedge & \underbrace{locked = \bot \wedge 2 \in xx}_{en_{add_2}(s)} \\ & \wedge \underbrace{locked = \bot \wedge 1 \notin xx \wedge xx' = xx \cup \{1\} \wedge locked' = locked}_{BA_{add_1}(s,s')} \\ & \wedge \underbrace{\neg(locked' = \bot \wedge 2 \in xx')}_{\neg en_{add_2}(s')})\end{aligned}$$

As this constraint is a contradiction, we can conclude that `add_1` cannot disable `add_2` (and, analogously, vice versa). This does not suffice for independence, and we have to continue to check the commutativity of the operations (see below). However, `lock` can (and will) disable `add_1` and the operations cannot be independent. The same holds for `lock` and `add_2`.

Commuting Constraint

The next constraint below encodes counter examples to commutativity in Def. 1. egain, if a solution is found, a timeout occurs or unknown is returned by the solver, we conclude that the operations might be non-commuting and thus dependent:

$$\exists s, s_1, s_2, s_3, s_4.(Info \wedge BA_\alpha(s, s_1) \wedge BA_\beta(s, s_2) \wedge BA_\alpha(s_2, s_3) \wedge BA_\beta(s_1, s_4) \wedge s_3 \neq s_4)$$

E.g., to find that `add_1` and `add_2` commute, the following constraint is used:

[9] We will directly refer to the state variables by their name; e.g., xx is part of state s, and xx' is a variable of s'.

$$\exists s, s_1, s_2, s_3, s_4.(\underbrace{locked = \bot \wedge 1 \notin xx \wedge xx_1 = xx \cup \{1\} \wedge locked_1 = locked}_{BA_{add_1}(s,s_1)}$$
$$\wedge \underbrace{locked = \bot \wedge 2 \notin xx \wedge xx_2 = xx \cup \{2\} \wedge locked_2 = locked}_{BA_{add_2}(s,s_2)}$$
$$\wedge \underbrace{locked_2 = \bot \wedge 1 \notin xx_2 \wedge xx_3 = xx_2 \cup \{1\} \wedge locked_3 = locked_2}_{BA_{add_1}(s_2,s_3)}$$
$$\wedge \underbrace{locked_1 = \bot \wedge 2 \notin xx_1 \wedge xx_4 = xx_1 \cup \{2\} \wedge locked_4 = locked_1}_{BA_{add_2}(s_1,s_4)}$$
$$\wedge \underbrace{\neg(xx_3 = xx_4 \wedge locked_3 = locked_4)}_{s_3 \neq s_4})$$

Due to the associativity and commutativity of the set union, the two operations will commute. Further, as they do not disable each other, the constraint can be found to be unsatisfiable. Hence, we know for certain that for all states Def. 1 holds and the operations are independent of each other.

Non-Enabling Constraint

For condition (A 2.2), we also have to know which operations can *enable* each other. In order to determine whether operation α can enable β, we need a constraint similar to the non-disabling constraint:

$$\exists s, s'.(Info \wedge \neg en_\beta(s) \wedge BA_\alpha(s, s') \wedge en_\beta(s'))$$

As an example, `add_1` cannot enable `add_2` and vice versa. However, both these operations can be enabled by `unlock`.

Co-Enabledness Constraint

Again, for condition (A 2.2), we need to know which operations are potentially co-enabled. The constraint below is true if the operations α and β are co-enabled in some state:

$$\exists s.(Info \wedge en_\alpha(s) \wedge en_\beta(s))$$

For example, `add_1` and `add_2` are both enabled in the initial state. However, `lock` and `unlock` are never co-enabled as their guards form a contradiction.

Advantage: Precision

Overall, such a constraint-based analysis is very precise and, in an optimal world, would obtain all necessary information for POR.

Drawback: Required Information

In practice, (proven) invariants often are important to determine independence (i.e., they should be part of the *Info* predicate above). E.g., if $x > 0 \Rightarrow x = y$ is known, we can infer that the guards $x > 0$ and $y \leq 0$ are mutually exclusive. However, adding conjuncts to the *Info* predicate can also make a constraint solver time out. We were not able to find a heuristic that selects additional information for the solver and consistently succeeds for more complex models.

Drawback: Analysis Overhead

For many constraints the solvers time out, which vastly increases the POR analysis time. We found that for many models, such an analysis surpasses the actual model checking time for the full state space. The issue is further discussed regarding the interlocking example in section 7.5.2.

Drawback: Instability of Solver Integrations

ProB's own constraint solver does not perform well in finding unsatisfiability of the commuting constraints. Other integrated solvers on the other hand, i.e., Kodkod and Z3 fit extraordinarily well. However, for some constraints Kodkod and Z3 will occupy all available memory (including swap space), leading to crashes during POR analysis.

7.4.2. Solution 2: SAT Encoding of Finite Sets

While the constraint-based approach above works well for smaller models, the blow up of analysis time renders it less favourable for larger ones. Thus, we have implemented a prototype[10] that aims to expose syntactic independence by automatically re-writing finite set variables (as well as finite relations) into a series of boolean variables. This is technique often refered to as "bit blasting", or "data refinement" in the context of modelling and refinement. It also is used in Kodkod's translation to SAT, and similar re-writes are required when encoding such a model in lower-level formalisms, such as Promela. In listing 7.2, an example encoding is given for the machine in listing 7.3.

One can see that the (original) set variable `xx` can contain at most two values that can be determined statically (i.e., 1 and 2). Then, the original set `xx` is replaced by a group of boolean variables, here `xx_1` (that equals `TRUE` iff $1 \in xx$) and `xx_2` (that equals `TRUE` iff $2 \in xx$). Finally, a membership check is a comparison with `TRUE` (or `FALSE` for non-membership, e.g., in the guard of `add_1`), and the set union with a singleton set just sets the according boolean to true (e.g., in the body of `add_1`). Most operators concerning sets, functions and relations can be re-written (though some translations are rather involved [TJ07], and are omitted here).

[10] Available at: `https://github.com/JanRossbach/fset`

Advantage: Faster Analysis

The POR analysis yields a pretty precise result even if the original, fast syntactical analysis in fig. 7.3 is re-used. For example, `add_1` reads and writes only `xx_1` and does not require `xx_2`, and vice versa for `add_2`, resulting in independent operations on a syntactical level. Further, as the behaviour of the machine is not altered, one could also verify that this is a valid refinement in order to ensure correctness.

Drawbacks

Performance There are several aspects of performance overheads to consider here: first, the translation itself requires some time, especially if all operations are unrolled and if complicated invariants are used. For larger models, our prototype of the translation may take several minutes. Second, the translated model does not perform as well during model checking with ProB, and may be several times slower. Thus, a sensible option would be to use the translated model for POR analysis only and map the results to the original model.

Translatable Subset Unfortunately, not all operators in the B language have a straightforward mapping to a SAT encoding. As a fallback, one may re-calculate the original set by combining all boolean values it is spliced into. Yet, in these instances, one loses all syntactic independence again.

7.5. Case Study & Challenge: Railway Interlocking System

In his book on Event-B [Abr10], Abrial presents a model[11] of a railway interlocking system. The role of an interlocking is to safely operate signals and points within an area of the train network. This means that the interlocking controller has to ensure that trains do not collide and that points are not moved while a train is driving over them.

In this section, we investigate the impact of the POR analysis techniques we presented above with this interlocking system by Abrial [Abr10, Chapter 17] (cf. listing 7.4). Although it is an academic model intended for teaching, we chose it because *(i)* it shares several features with real-world models, *(ii)* while SAT-based approaches are able to verify small to medium-sized interlockings [Bor18, PFB19], the verification of larger interlockings is still an active research area and challenge, *(iii)* applying ProB's *POR yields no state space reduction,(iv)* it requires vast resources for model checking — its state space for the simple topology from Fig. 7.4 consists of 61 648 077 states and invariant checking with ProB would take about six days (based on estimates [KB18] — without distributed model checking, the process ran out of memory and crashed), *(v)* one can identify that partial-order reduction is in principle possible because the `route_freeing` operation is independent of all other operations. One can hand-code

[11] `https://github.com/pkoerner/train-por/blob/main/Train_1_beebook_TLC.mch`

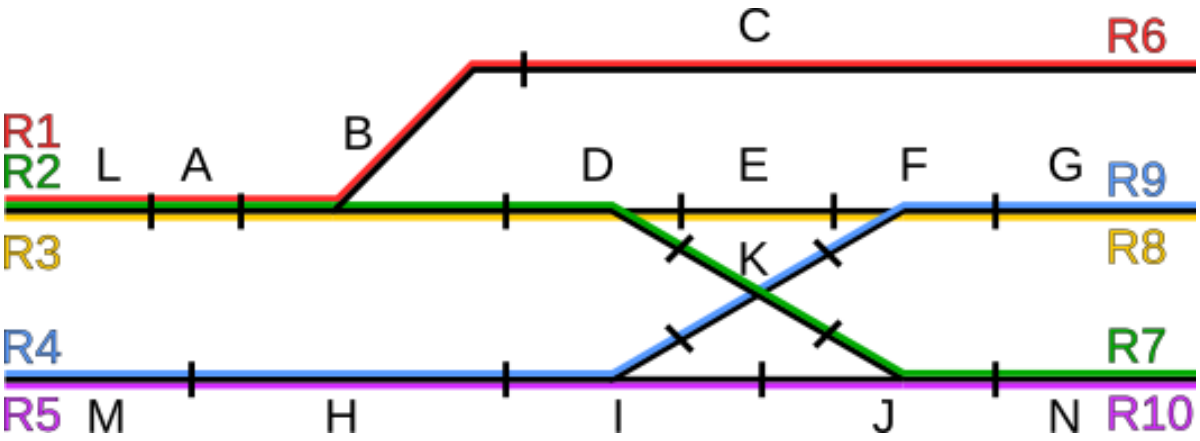


Figure 7.4.: Example interlocking track layout based on page 524 of [Abr10] with 5 signals, 5 points, one crossing and 14 tracks segments

this insight into the model [LBH14] by forcing this operation (`route_freeing`) to be taken as soon as it is enabled[12], thereby reducing the state space to 672 174 states. Our challenge for the last years has been to identify why our current approach fails and to obtain this two-order of magnitude reduction by (an improved) POR.

7.5.1. Interlocking Model Overview

The rail network is divided into individual *blocks*; the blocks in Fig. 7.4 are named A – N. The interlocking allows trains to follow a fixed number of statically determined *routes* through the network. Fig. 7.4 contains 10 routes, named R1 – R10. For example, route R1 goes through blocks L, A, B, C, while route R2 goes through L, A, B, D, E, F, G and route R6 is the reversed route of R1, going through C, B, A, L (analogously for R7 – R10).

The model also contains the following constants and variables: `fst` and `lst` are functions that map a route to its first and last block, respectively. `nxt` is a function that — given a route — returns a function mapping a block to its successor. `rtbl` is a relation storing the routes for each block.

`resbl` (reserved blocks) `resrt` (reserved routes) and `rsrtbl` (blocks reserved for routes) store information about reservations. `OCC` keeps track of blocks that are occupied. `frm` stores which routes are formed on the physical track (`TRK`). `LBT` maps a route to the last block of the train.

Operations are usually called within a certain order: first, a route has to be reserved (`route_reservation`) and the points need to be positioned to match the route (`point_positionning`). Then, these points are locked as the route is formed (`route_formation`). On formed routes, trains may enter and leave blocks in the corresponding order (via the operations `FRONT_MOVE_1`, `FRONT_MOVE_2`, `BACK_MOVE_1` and `BACK_MOVE_2`). Once a train finishes its route, the route is freed again (`route_freeing`).

[12] `https://github.com/pkoerner/train-por/blob/main/Train_1_beebook_tlc_POR.mch`

```
SETS BLOCKS={A,B,C,D,E,F,G,H,I,J,K,L,M,N};
     ROUTES={R1,R2,R3,R4,R5,R6,R7,R8,R9,R10}
CONSTANTS fst, lst, nxt, rtbl
VARIABLES LBT, TRK, frm, OCC, resbl, resrt, rsrtbl
INITIALISATION
resrt := ∅ || resbl := ∅ || rsrtbl := ∅ || OCC := ∅ || TRK := ∅ ||
frm   := ∅ || LBT   := ∅
OPERATIONS
route_reservation(r) =
  SELECT r ∉ resrt ∧ (rtbl⁻¹)[{r}] ∩ resbl = ∅
  THEN resrt := resrt ∪ {r} ||
       rsrtbl := rsrtbl ∪ (rtbl ▷ {r}) ||
       resbl := resbl ∪ (rtbl⁻¹)[{r}] END;
route_freeing(r)
  SELECT r ∈ resrt \ ran(rsrtbl)
  THEN resrt := resrt \ {r} || frm := frm \ {r} END;
FRONT_MOVE_1(r) =
  SELECT r ∈ frm ∧ fst(r) ∈ resbl \ OCC ∧ rsrtbl(fst(r)) = r
  THEN OCC := OCC ∪ {fst(r)} || LBT := LBT ∪ {fst(r)} END;
FRONT_MOVE_2(b) =
  SELECT b ∈ OCC ∧ b ∈ dom(TRK) ∧ TRK(b) ∉ OCC
  THEN OCC := OCC ∪ {TRK(b)} END;
BACK_MOVE_1(B) =
  SELECT b ∈ LBT ∧ b ∉ dom(TRK)
  THEN OCC := OCC \ {b} || rsrtbl := {b} ⩤ rsrtbl ||
       resbl := resbl \ {b} || LBT := LBT \ {b} END;
BACK_MOVE_2(b) =
  SELECT b ∈ LBT ∧ b ∈ dom(TRK) ∧ TRK(b) ∈ OCC
  THEN OCC := OCC \ {b} || rsrtbl := {b} ⩤ rsrtbl ||
       resbl := resbl \ {b} || LBT := LBT \ {b} ∪ {TRK(b)} END;
point_positionning(r) =
  SELECT r ∈ resrt \ frm
  THEN TRK := ((dom(nxt(r)) ⩤ TRK)
               ⩥ ran(nxt(r))) ∪ nxt(r) END;
route_formation(r) =
  SELECT r ∈ resrt \ frm ∧
        (rsrtbl⁻¹)[{r}] ◁ nxt(r) = (rsrtbl⁻¹)[{r}] ◁ TRK
  THEN frm := frm ∪ {r} END
END
```

Listing 7.4: Grand Challenge: Abrial's Interlocking System (Excerpt)

Since only some routes share blocks, several routes can be reserved, formed and several trains may be on the tracks at the same time. For example, route R1 does not share any block with route R4 or R5. On the other hand, route R3 and R4 both include the blocks F and G.

7.5.2. Insights

Operation Unrolling

As previously mentioned, this is the key technique for the POR analysis that avoids rewriting the POR implementation itself. In our case study, one can unroll all operations, as parameters are either one of the ten routes or fourteen blocks. Then, the unrolled model has 92 operations. If the operations were not unrolled, one could not exploit that some pairs of routes do not overlap (and the corresponding operation instances are, thus, *independent*). One consequence is that the POR analysis cannot infer the independence of, e.g., the route reservation of the disjoint routes R1 and R5. Another consequence is that, e.g., `route_reservation` and `route_formation` are overapproximated as dependent, even though *some* pairs of routes do not overlap (and the corresponding operation instances are, thus, *independent*).

Constrained-Based Analysis

The constraint-based approach is able to yield a precise independence analysis. This, however, comes with a cost: if operations *are* dependent on each other, solvers usually time out rather than returning a counterexample or unknown. As many operations do not commute (or may enable or disable each other), this drastically increases POR analysis time. As 4186 (unordered) pairs of operations exist, a full analysis that checks the non-disabling, commutativity (for independence) as well as non-enabling and co-enabledness constraints (for (A 2.2)) takes several hours even on modern hardware due to the amount of timeouts. Finally, even though the obtained information was pretty precise, we did not achieve any reduction with this approach. The POR analysis was not able to determine that a crucial pair of operations cannot be co-enabled (cf. (A 2.2)), and was not precise enough concerning the enabling relation. In particular, for the same parameter route `R`, the operation instance `route_freeing(R)` may disable both `point_positionning(R)` and `route_formation(R)` and, thus, is not independent of them. However, the operations are never enabled at the same time. If this co-enabledness was disproven, the reduction would occur as expected.

SAT Encoding

Finally, the SAT encoding of the original model[13] *in combination with* the constraint-based analysis yielded the most precise POR analysis results. In consequence, the technique also allowed the POR algorithm to achieve the same reduction as the hand-written

[13] `https://github.com/pkoerner/train-por/blob/main/train_auto4.mch`

version. Analysis and model checking takes about 30 minutes (1881 seconds) and requires 5048 MB of memory. In comparison, the hand-written version without ProB's POR takes around 7 minutes (397 seconds) and uses 2038 MB of memory. The faster runtime is due to the overhead of the POR as well as the less efficient encoding of the refinement. Reasons for the additional memory usage include a larger refined model and larger states, storage of POR analysis results, etc.

7.6. Conclusions and Future Work

In this chapter, we have identified two idioms in B and Event-B — operation abstraction by parameters and usage of high-level data types — that often hinder the POR analysis and, henceforth, successful state space reduction. Certainly, there are further patterns that may be uncovered in the future. Thus, our main conclusion is that the usage of high-level constructs prevalent in B are indeed the root cause for our previous unsatisfying experiences with POR and, thus, deeper analysis is required.

We have described three techniques in sections 7.3 and 7.4, (i.e. unrolling of operations, constraint-based POR analysis of operations based on before-after predicates and/or a precise SAT encoding of finite set variables). Individually, each technique is no universal remedy and brings its own drawbacks to the table. In combination, however, one can exploit their individual advantages and, indeed, we were able to match the two order of magnitude state space reduction of the hand-written version for deadlock checking of the interlocking case study.

Related work is dynamic POR [FG05] which is especially useful for model checking of concurrent software systems, where possible parameter values are drawn from large or infinite sets such as integer values. It avoids static analysis altogether, tracks information dynamically during execution traces and backtracks later if alternative paths that need to be explored are identified. One main benefit is that one does not need to keep the entire state space in memory but only the execution that is currently considered. While this is quite different from our approach, it still requires precise information on the dependence relation and, thus, cannot yield better reduction alone. Yet, evaluating the dependency relation *lazily* — i.e., considering only combinations of operation instances which are actually encountered — can help where our improvements in sections 7.3 and 7.4 currently fail, i.e., when parameters are drawn from infinite sets or when sets are statically unbounded.

The constraint-based analysis still has room for improvement: for one, there might be useful heuristics for similar operation pairs to avoid timeouts. If missing information was made more transparent to the user, one might also assist the POR analysis by providing (proven) theorems. Yet, our implementation of SAT encoding is not mature enough for large-scale benchmarking. In the future, we aim to evaluate our new approach in the large.

Finally, the focus of this study lies on deadlock checking — invariant or LTL model checking may require different or additional techniques. In particular, it is often hard to prove that operations preserve the invariant (which is required for operations to be stut-

ter events, which in turn is required for successful reduction during LTL model checking). Thus, work in this direction might benefit from integrating provers to obtain information about invariants that are guaranteed to be preserved by individual operations.

Acknowledgments

The authors would like to thank Joshua Schmidt for his patience and relentless work on the Z3 interface and Jan Roßbach for his implementation of the SAT encoding of finite sets.

7.A. Pseudo-Code Overview of the POR Analysis

```
non_disabling    = set()
independent      = set() ## symmetric
may_enable       = set()
may_be_coenabled = set() ## symmetric


def por_analysis():
  determine_non_disabling()
  determine_independence() # execute after non_disabling is determined
  determine_may_enable()
  determine_coenabledness()

def determine_non_disabling():
  for op1 in ops:
      for op2 in ops:
          if op1 == op2:
              ## not relevant for POR
          ## phase 1: syntactic check (fast):
          if empty(intersection(write(op1), read(op2))):
              non_disabling.add(pair(op1, op2))
          else:
              ## optional phase 2: fallback to constraint solver
              if solve(non_disabling_constraint(op1, op2)) == contradiction:
                  non_disabling.add(pair(op1, op2))

def determine_independence():
  ## requires non_disabling relation to be calculated
  for op1 in ops:
      for op2 in ops:
          if op1 == op2:
              pass ## assume dependent
          else if op1 >= op2:
              pass ## performance optimisation: symmetric relation
          else if pair(op1, op2), pair(op2, op1) in non_disabling:
              ## if one op might disable the other, ops are dependent.
              ## phase 1: syntactic check (fast):
              if empty(intersection(write_set(op1), write_set(op2))):
                  independent.add(pair(op1, op2))
                  independent.add(pair(op2, op1)) # symmetry reduction
              else:
                  ## optional phase 2: fallback to constraint solver
                  if solve(commuting_constraint(op1, op2)) == contradiction:
                      independent.add(pair(op1, op2))
                      independent.add(pair(op2, op1)) # symmetry reduction
```

```
45
46 def determine_may_enable():
47   for op1 in ops:
48       for op2 in ops:
49           if op1 = op2:
50               continue ## an op cannot enable itself
51           ## phase 1: syntactic check (fast):
52           if empty(intersection(write_set(op1), guard_read_set(op2)):
53               pass
54           else:
55               ## optional phase 2: fallback to constraint solver
56               if solve(non_enabling_constraint(op1, op2)) != contradiction:
57                   may_enable.add(pair(op1, op2))
58
59 def determine_coenabledness():
60   for op1 in ops:
61       for op2 in ops:
62           ## no reasonable syntactical approximation feasible
63           ## must assume true without further information
64           if solve(coenabledness_constraint(op1, op2)) != contradiction:
65               may_be_coenabled.add(pair(op1, op2))
```

Listing 7.5: Pseudo-Code of POR Analysis

Conclusions

8. Conclusions and Future Work

In this final chapter, we re-visit the research questions from chapter 1 and try to answer them based on the conclusions from each article.

8.1. Integrating formal specifications into applications: the ProB Java API

We found that the ProB Java API renders it easy to embed high-level specifications directly into programs. This raises the questions:

- RQ 1: In what circumstances should (high-level) specifications be executed?
- RQ 2: What kinds of applications can reasonably interact with a formal methods tool?
- RQ 3: What are benefits and drawbacks wrt. generated code?

Re RQ 1: When to execute high-level specifications?

Executability of formal specifications — e.g., in the form of animation — is always desirable in order to verify not only the absence of errors but also that the specification actually fulfils the requirements. Naturally, this should happen as soon as possible in the development process.

Re RQ 2: When to embed FM tools?

It makes sense to embed a specification in order to simulate an environment that follows known rules. One example is the ETCS HL3 case study, where existing components were hooked into the application and triggered certain events; another is Pac-Man, where the ghosts have several movement options at a junction, but the actual behaviour is described by an algorithm.

Less suitable are applications with heavy real-time requirements; for example, a Pac-Man GUI that actually feels smooth seems still out of reach due to the communication overhead with the ProB kernel. If the requirements are more lax, as with the ETCS HL3 demonstrator, embedding the specification is certainly an option. Real-time constraints can also be acted upon in two ways: first, direct integration of the formal methods tool; in the case of ProB, implementation in Prolog that avoids communication and translation overhead makes more applications feasible at the cost of leaving the Java

eco-system behind. Second, refinement to lower levels of abstraction that require less or even avoid constraint solving, which should speed up the calculation of successor states.

Re RQ 3: Executing Specifications vs Code Generation

In some instances, deriving an implementation is hard or cumbersome, and, often, too many of refinement steps are necessary to actually reach B0. Some steps can be done automatically [Lec14b], using tools like BART [BM99]. If the required performance is provided by ProB and its Java API, one can certainly embed the entire tool in an application. Nota bene: this approach should be used for rapid prototyping or demonstrators, not for actual safety-critical systems.

Code generation from more high-level specifications has been explored [Vu18, VHKL19]. However, this approach also faces limitations: first, a sufficient number of refinements is still required — e.g., comprehension sets must follow a certain pattern where the n-th conjunct of the predicate must constrain the n-th variable to a finite type. Second, large or infinite sets (e.g., set of integers) are still enumerated entirely. Here, the performance of constraint solvers makes a significant difference [VBL22]. Thus, we conclude, that for applications that work on small and finite types, generated code easily outperforms interpretation; however, once efficient constraint solving is required, embedding the entire ProB interpreter may be beneficial.

So What? Overall, developing formal specifications similar to a "model-in-the-loop testing" approach seems to be a sensible next step in the evolution of formal methods and should be made more accessible to practitioners. Similarly, the Functional Mock-Up Interface (FMI) standard [BOA+12] allows co-simulation of hybrid models. Using FMI, a B model could simulate discrete state changes only, while other tools simulate a continuous environment. Especially with recent experiments with floating-point and real numbers, an implementation of the FMI is worth pursuing.

Another interesting idea could be to combine code generators, e.g., B2Program [VHKL19], with the Java API for faster execution of (parts of) specifications. This could, again, enhance the range of applications where the performance of the presented approach is not sufficient.

8.2. A Verified Low-Level Implementation and Visualization of the Adaptive Exterior Light and Speed Control System

Our case study in chapter 3 was designed to address the following questions:

- RQ 4: How does verification after a non-formal, test-driven workflow ("correcting-the-construction") compare to applying formal methods from the get-go "correct-by-construction"?

- RQ 5: What classes of properties of C code are verifiable by existing tools?

Re RQ 4: Formal vs Non-Formal Development Approaches

The implementation rigorously followed a test-driven approach; the number of lines of code for testing is about four times the number of lines of code of the implementation. At the early implementation phase, we thus felt confident in the correctness of the implementation, and our tests were able to catch the most severe bugs. However, the more implementation work was done, the more we felt that we lost understanding of our code and changes triggered failing tests more frequently. Additionally, the test scenarios required more adaptations of our code.

Addressing located issues is far from trivial. For example, the implementation of the light system is rather small with about 400 lines of C code. However, the main function that reads all sensors and calculates the signals to be sent to the lights contains 35 if-statements on the top-level, with additional nested ifs. The conditions of the if-statements are in many cases not trivial to understand, which makes it hard to reason about conditions that are contradictory or what requirement has precedence over others.

The design of our implementation allowed us to interact with the code easily and account for missing tools: for example, graphics libraries could be linked with the code in order to achieve visualisations of real-time executions. Further, we found that the state changes in the trace provided by CBMC were too fine-grained. However, we could quickly account for this issue with an ad-hoc visualisation tool.

Merging several execution step might also make sense for animation of higher-level formalism: e.g., the CODA framework [BCE+13] (based on Event-B and UML-B [SB06]) automatically executes "internal" events (usually, a skip event in the abstract machine) using ProB. State visualisation tools such as BMotionWeb [LL16] and VisB [WL20] also can execute several events at once (as, e.g., in the Pac-Man case study section 2.3.1).

Re RQ 5: Verifiable Properties on Implementation

With CBMC, one can verify properties such as invariants in B (depending on where an assertion is placed in the code). However, the actual verification requires unwieldy injection of additional information for the tool in the code. While there are other tools that work on C code, CBMC does not support temporal logics such as LTL or CTL, yet, some properties (e.g., the next operator) can be emulated by injecting additional state variables.

Attempts of formal verification using CBMC immediately raised counterexamples, in some cases due to bugs, in others due to our assumptions on possible program states, or due to missing information from the specification. Sound verification, however, is not as easy as it seems at first glance and more training with the CBMC tool is required than expected. Indeed, our first attempts of verification were incorrect and significant changes to the code are required. Here, the claimed benefits of a more structured approach prove true.

So What? In summary, this case study will add to the body of evidence that formal methods are beneficial. At the very least, a model checking tool exposed errors we did not locate before. In order to organise the code, a notion of guarded substitutions, such as operations in B, as well as related analysis tools would certainly have been helpful. Yet, looking back after more than two years since the submission of our case study, it is likely that changes to code and a formal specification will be painful and error-prone. However, we would have more trust in correctness of formal model here. So, overall, the approach of the article should not be adopted for real projects.

On the other hand, it is worth highlighting the benefits of our approach: the code can be directly linked into real sensors and be executed. Also, a broad range of off-the-shelf libraries can be applied. Typically, this is not achievable with formal method tools, though one can achieve similar results using, e.g., the ProB Java API.

Finally, the experiment showed that, indeed, it is possible to locate errors in C code. Verification of invariants is also efficient: due to the nondeterministic sensor reads and user interactions, the code is translated to SMT predicates that form an inductive proof and only a few iterations of the main loop have to be unwound. Thus, for code generators, such an additional validation step on the emitted code certainly is feasible and only requires that the emitted code contains CBMC's annotations and adheres to its naming conventions.

8.3. Treating Specifications as Data

With our implementation of *lisb* in chapter 4, we examined the research questions below:

- RQ 6: What kind of issues of B's DEFINITIONS can be addressed using a Lisp-style macro system?
- RQ 7: What is a favourable way to integrate external data sources with ProB's constraint solver?
- RQ 8: How does meta-programming of B models elevate DSL and tool development?

Re RQ 6: Alternative Definition System

The expressiveness of Clojure's macro system is a great fit for generating B machines. However, getting the right combination of quoting and unquoting reader macros right is notoriously intricate; from my experience teaching Clojure, this is the topic students struggle the most with. Thus, the difficulty of mastering this kind of macro system may outweigh its benefits, especially in the context of safety-critical where correctness is required.

We have also presented *lisb*'s `defpred` mechanism, which is similar to ProB's interpretation of definitions. It inlines the predicate or expression by directly inserting a sub-tree in the AST. The difference is that *lisb* is not able to capture variables from an

outer scope. If another variable is used, it must be passed as an argument. We deem this a sensible middle ground of the alternatives, as it eliminates the most common error sources and is still easy to use.

Re RQ 7: Interaction with External Data Sources

The embedding of B in Clojure has similar advantages to the embedding of Alloy in Ruby (αRby) [MJ14]: no additional parser is needed, the language can be easily extended, solutions that the solver provides can be processed for further constraint solving tasks, and, relevant for RQ 7, external data from disk, network, etc. can be transformed into a format suitable for the formal language and its tooling.

The goals of *lisb* and αRby differ: where αRby seems to be mostly motivated by mixed execution, usage of partial solutions and stages model finding, *lisb* aims at processing input data, solutions and the opportunity of meta-programming entire specifications. Addresses these goals, we differ in our judgement concerning the programming language the formalism is embedded into: we deem a functional language more suitable for data transformation that an imperative language.

Re RQ 8: Meta-Programming Specifications

This approach facilitates tool and DSL development. As B machines are treated as plain data, stateless transformation is easily possible. As examples, first, we considered the algorithm description language implemented by Clark in her master's thesis [Cla16, CBH+16]. In a single afternoon, we were able to design a DSL in *lisb* that generates B operations for all statements contained in the proposed language. Second, as a bachelor student with no prior knowledge of *lisb* or B, Roßbach was able to implement a rather complex automatic refinement tool as part of his bachelor's thesis [Roß22]. Thus, we conclude that the infrastructure *lisb* for meta-programming is satisfactory, and even complex tools and DSLs can be implemented.

So What? Meta-programming of formal specifications are a preferable way of interacting with external data sources. It also enables easier development of tools that support (e.g., a refinement tool) or use (e.g., providing a DSL) the formalism. Certainly, it would be an interesting experiment to compose B specifications using the B language itself in order to obtain transformations that can be proven correct.

Many features prevalent in programming languages raise issues and questions on how to express them in B. Examples include, recursion, inheritance, interfaces or polymorphic functions in general, which require significant translation effort. It is not clear whether such constructs can be expressed at all with the current B language [Leu21]. Thus, future experiments on DSLs can drive the evolution of the B language or certain dialects.

8.4. Towards a Shared Specification Repository

In order to accommodate different possible use cases, we asked:

- RQ 9: How can information about specifications be organised in an open, extensible way?
- RQ 10: How should sets of benchmarks be handled?
- RQ 11: How should changes to specifications be incorporated into the repository?
- RQ 12: What kind of specifications are available in ProB's public examples?

Re RQ 9: Extensible Information Organisation

The B machines in the repository may be used by different applications. We simply used maps to organise the data. One would expect certain keys always to be present, as certain information is application- and tool-agnostic. Examples include whether the machine includes errors that should be determined on load (e.g., parse or type errors), how many variables and operations are present, and whether it contains a deadlock or invariant violation.

Other information depends on tooling and used techniques: for example, applying partial order reduction might yield a lower number of explored states and transitions, and, thus, improve run-time and memory consumption significantly. One would expect corresponding entries to be optional, e.g., if a technique cannot be applied to machines with certain features. Using Clojure's spec library[1], one can also automatically check conformance to this format.

Re RQ 10: Benchmark Sets

Benchmark sets and results can be included in the repository itself, as for example done for well-definedness checking [Leu20]. An alternative approach was recently presented in the context of machine repair [CSD+22], where the authors describe the filters they applied in order to reach the benchmark set. However, as machines can be added at any time, it is important to cite the revision that is used as a foundation.

Re RQ 11: Changes to Benchmarks

In the manuscript, we advocated that changes to the specifications should not occur and new version should be added instead. While immutability of benchmarks certainly is a desirable property, it may also hinder changes due to tool development. As an example, ProB introduced a new keyword `floor` for experiments with floating-point and real numbers. Consequently, all examples of lift machines that contained a state variable named *floor* to specify the position of the lift were broken and updated. Though it may be useful to document that certain machines once worked with ProB, in this instance, adding copies introduces semantic clones of the machines which may in turn lead to a bias for benchmarks. Thus, a more pragmatic approach would be to let the version control system git take care of tracking changes.

[1] `https://clojure.org/guides/spec`

Re RQ 12: Overview of Available Machines

Regarding the available machines, we can classify (at the time of writing) 4091 machines roughly as:

- 818 machines that do not parse properly,
- 659 B and 178 Event-B machines without operations (or events), which are most likely used for constraint programming or data validation,
- 1284 B and 288 Event-B machines with less than 1000 reachable states,
- 211 B and 95 Event-B machines with more than (or equal to) 1000 reachable states,
- 384 B and 174 Event-B machines of unknown size.

We can conclude that the repository contains many "small" machines with only few states. For 653 specifications, this may be due to the usage of deferred sets, which by default are treated as sets with only two elements. In such cases, one could probably scale the state space arbitrarily by changing ProB's preferences.

So What? Gathering information about ProB's public examples was long overdue and it is good to see that first research articles make use of it for larger-scale evaluation efforts [Leu20, CSD+22]. However, some further organisation is necessary in order to separate toy examples from specifications stemming from industry and academic case studies with similar features. Then, one can propose default benchmark sets for established applications such as model checking.

8.5. Empirical Evaluation of POR for B

Regarding partial order reduction in B, we asked ourselves:

- RQ 13: How well does the current implementation of POR in ProB perform?

Judging from our evaluation in chapter 6, the current implementation of POR in ProB does not achieve reduction for the majority of machines. While POR typically is associated with reduction of several orders of magnitude, we mostly observed less than an order of magnitude, with a few outliers that were very susceptible to POR. However, the empirical evaluation mostly considered smaller models, and POR might perform better for larger models. Yet, there is little substance supporting this theory: in chapter 7, we found that certain widespread modelling patterns, that are especially used to structure larger models, interfere with the POR analysis.

So What? While we assessed that, currently, POR does not perform well, we also found some strategies that enable more precise POR analysis. One can imagine that, in the future, certain automatic re-writes of machines may boost POR effectiveness significantly.

8.6. Towards Practical Partial Order Reduction for High-Level Formalisms

Finally, we are able to answer the following question:

- RQ 14: Why is the application of POR techniques in ProB unsuccessful in most cases?

The use of high-level data structures and abstraction is a good indicator that POR will not perform well. However, it is neither a sufficient nor a necessary condition; Many low-level specifications are not suitable for POR as their ordering is important, and some high-level specifications feature syntactical independence which allows ProB's POR to reduce the state space.

However, the usage of higher-level constructs is widespread in the B language, as they are more expressive and, as such, more comfortable to use. In chapter 7, we identified two idioms — operation abstraction by parameter and usage of sets — that interfere with ProB's POR analysis. We also presented strategies that improve the precision of the POR analysis.

This does not mean that these are the only two idioms. Careful evaluation and better visualisation why a POR technique fails is necessary to identify or counteract further idioms. In particular, a large-scale evaluation similar to the one in chapter 6 that includes new techniques is still outstanding.

One of the main issue concerning solvers in the POR analysis is that in the current implementation, they are called too rarely to be effective. In practice, most of the required analysis result can be obtained by solvers such as Kodkod (if applicable) as well as Z3. Here, these solvers are often able to produce counterexamples quickly. However, if the constraint holds true, the solvers usually time out. This renders the usage to their best capabilities difficult and unlikely.

ProB's own solver does not work particularly well with commuting constraints that include set operators; e.g., $(s \cup \{1\}) \cup \{2\} = (s \cup \{2\}) \cup \{1\}$ can not be proven to be true as s is quantified over all possible set values. Improved constraint handling rules that encode commutative operators may help here.

So What? Overall, the situation is not as dire as chapter 6 suggests. The two identified idioms that hinder POR are widely used in B and addressing them can improve the performance of POR significantly. On the other hand, the current state still is not very satisfactory: the bitvector encoding does not work for all operators. Especially function calls are hard to translate, and n-ary relations for $n > 2$ can blow up significantly. The solvers that are currently available are also not sufficient, as timeouts are too costly during analysis of larger models.

Thus, an approach that seems to be more hopeful would be to create specialised solvers for each constraint type. For example, many operators $\triangle$ (e.g., the set union $\cup$) are known to be commutative and associative, so that $(a \triangle b) \triangle c = a \triangle (b \triangle c) = a \triangle (c \triangle b) = (a \triangle c) \triangle b$ holds, which is the form of the independence constraints. Using constraint

handling rules (CHR) [Frü09], one could specify such rules in Prolog. Then, a mixture of solvers and bitvector encoding might be a combination that works well for the static analysis.

Alternatively, one could also explore the dynamic partial order reduction techniques and see what benefits are transferable from software model checking to high-level specifications.

Bibliography

[ABG+20] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Modelling an Automotive Software-Intensive System with Adaptive Features Using ASMETA. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2020.

[ABH+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[ABHV06] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An Open Extensible Tool Environment for Event-B. In *Proceedings ICFEM (International Conference on Formal Engineering Methods)*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr06] Jean-Raymond Abrial. Formal Methods in Industry: Achievements, Problems, Future. In *Proceedings ICSE (International Conference on Software Engineering)*, pages 761–768. ACM, 2006.

[Abr07] Jean-Raymond Abrial. Formal Methods: Theory Becoming Practice. *Journal of Universal Computer Science*, 13(5):619–628, 2007.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[AGRS11] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41(2):155–166, 2011.

[ALN+91] Jean-Raymond Abrial, Matthew Lee, Dave Neilson, P.N. Scharbach, and Ib Holm Sørensen. The B-Method. In *Proceedings VDM (International Symposium of VDM Europe)*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 1991.

[ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification language. In *On the Construction of Programs: An Advanced Course*. Cambridge University Press, 1980.

[aut19] *General Specification of Basic Software Modules*. AUTOSAR, Munich, 2019.

[AV14] Robert Abo and Laurent Voisin. Formal implementation of data validation for railway safety-related systems with OVADO. In *Proceedings SEFM (International Conference on Software Engineering and Formal Methods) 2013*, volume 8368 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2014.

[Bac81] Ralph-Johan Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981.

[BAPM83] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The Temporal Logic of Branching Time. *Acta informatica*, 20(3):207–226, 1983.

[Bau04] Hubert Baumeister. Combining Formal Specifications with Test Driven Development. In *Proceedings XP/Agile Universe*, volume 3134 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2004.

[BBH18] Roberto Bagnara, Abramo Bagnara, and Patricia M. Hill. The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software. In *Proceedings SAS (International Static Analysis Symposium)*, volume 11002 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2018.

[BC10] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer, 2010.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings TACAS (International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCE+13] Michael Butler, John Colley, Andrew Edmunds, Colin Snook, Neil Evans, Neil Grant, and Helen Marshall. Modelling and refinement in CODA. In *Proceedings Refine (International Refinement Workshop)*, volume 115 of *EPTCS*, pages 36–51, 2013.

[BCM+92] Jerry R. Burch, Edmund Clarke, Kenneth L. McMillan, David L. Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and computation*, 98(2):142–170, 1992.

[BDLM14] Richard Bonichon, David Déharbe, Thierry Lecomte, and Valério Medeiros. LLVM-Based Code Generation for B. In *Formal Methods: Foundations and Applications*, volume 8941 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014.

[BDMS05] Clark Barrett, Leonardo De Moura, and Aaron Stump. SMT-COMP: Satisfiability modulo theories competition. In *Proceedings CAV (International Conference on Computer Aided Verification)*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23. Springer, 2005.

[Bec03] Kent Beck. *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley, 2003.

[BGJ+21] Jens Bendisposto, David Geleßus, Yumiko Jansing, Michael Leuschel, Antonia Pütz, Fabian Vu, and Michelle Werth. ProB 2-UI: A Java-Based User Interface for ProB. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 12863 of *Lecture Notes in Computer Science*, pages 193–201. Springer, 2021.

[BGR03] Egon Börger, Angelo Gargantini, and Elvinia Riccobene. *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003. Proceedings*, volume 2589. Springer, 2003.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, 1995.

[BJL+19] Frederik Meyer Bønneland, Peter Gjøl Jensen, Kim G. Larsen, Marco Muñiz, and Jirí Srba. Partial Order Reduction for Reachability Games. In *Proceedings CONCUR (International Conference on Concurrency Theory)*, volume 140 of *LIPIcs*, pages 23:1–23:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BKK+20] Michael Butler, Philipp Körner, Sebastian Krings, Thierry Lecomte, Michael Leuschel, Luis-Fernando Mejia, and Laurent Voisin. The First Twenty-Five Years of Industrial Use of the B-Method. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 12327 of *Lecture Notes in Computer Science*, pages 189–209. Springer, 2020.

[BKL+16] Jens Bendisposto, Philipp Körner, Michael Leuschel, Jeroen Meijer, Jaco van de Pol, Helen Treharne, and Jorden Whitefield. Symbolic Reachability Analysis of B through ProB and LTSmin. In *Proceedings iFM (International Conference on integrated Formal Methods)*, volume 9681 of *Lecture Notes in Computer Science*, pages 275–291. Springer, 2016.

[BL05] Michael Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In *Proceedings FM (International Symposium on Formal Methods)*, volume 3582 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2005.

[BL09] Jens Bendisposto and Michael Leuschel. Proof assisted model checking for B. In *Proceedings ICFEM (International Conference on Formal Engineering Methods)*, volume 5885 of *Lecture Notes in Computer Science*, pages 504–520. Springer, 2009.

[BM99] Lilian Burdy and Jean-Marc Meynadier. Automatic refinement. In *Proceedings BUGM (B User Group Meeting) at FM'99*, 1999. `https://www-sop.inria.fr/everest/Lilian.Burdy/ug020003.pdf`.

[BM08] Cathal Boogerd and Leon Moonen. Assessing the Value of Coding Standards: An Empirical Study. In *Proceedings ICSM (International Conference on Software Maintenance)*, pages 277–286. IEEE, 2008.

[BOA+12] Torsten Blochwitz, Martin Otter, Johan Akesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings MODELICA*, pages 173–184. Linköping University Electronic Press, 2012.

[Bor18] Arne Borälv. Interlocking Design Automation Using Prover Trident. In *Proceedings FM (International Symposium on Formal Methods)*, volume 10951 of *Lecture Notes in Computer Science*, pages 653–656. Springer, 2018.

[BS03] Egon Börger and Robert F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard – Version 2.0. In *Proceedings SMT (International Workshop on Satisfiability Modulo Theories)*, 2010.

[BTJHL17] Victor Bandur, Peter Würtz Vinther Tran-Jørgensen, Miran Hasanagic, and Kenneth Lausdahl. Code-generating VDM for Embedded Devices. In *Proceedings of the 15th Overture Workshop*, volume 1513 of *School*

of Computing Science Technical Report Series, pages 1–15. School of Computing Science, University of Newcastle upon Tyne, 2017.

[BvdPW10] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and Symbolic Reachability. In *Proceedings CAV (International Conference on Computer Aided Verification)*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer, 2010.

[BW10] Raymond P.L. Buse and Westley R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[BW12] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 2012.

[CBH+16] Joy Clark, Jens Bendisposto, Stefan Hallerstede, Dominik Hansen, and Michael Leuschel. Generating Event-B Specifications from Algorithm Descriptions. In *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM and Z)*, volume 9675 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2016.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings OSDI (Conference on Operating Systems Design and Implementation)*, volume 8, pages 209–224. USENIX Association, 2008.

[CEN11] CENELEC. Railway Applications – Communication, signalling and processing systems – Software for railway control and protection systems. Technical Report EN50128, European Standard, 2011.

[CES86] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS (Transactions on Programming Languages and Systems)*, 8(2):244–263, 1986.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings TACAS (International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKY03] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings DAC (Design Automation Conference)*, pages 368–371. IEEE, 2003.

[Cla16] Joy Clark. An Algorithm Description Language of Event-B. Master's thesis, Heinrich Heine Universität Düsseldorf, January 2016.

[Cle16] ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2016. Available at `http://www.atelierb.eu/`.

[CLM+19] 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)*, volume 11495 of *Lecture Notes in Computer Science*, pages 202–212, 2019.

[Clo] Clojure Spec Guide. `https://clojure.org/guides/spec`. Accessed: 2020-03-12.

[CM12a] Dominique Cansell and Dominique Méry. Foundations of the B method. *Computing and informatics*, 22(3-4):221–256, 2012.

[CM12b] Mats Carlsson and Per Mildner. SICStus Prolog—the first 25 years. *Theory and Practice of Logic Programming*, 12:35–66, 2012.

[CML20] Alcino Cunha, Nuno Macedo, and Chong Liu. Validating Multiple Variants of an Automotive Light System with Electrum. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 2020.

[CMR07] Dominique Cansell, Dominique Méry, and Joris Rehm. Time Constraint Patterns for Event B Development. In *Proceedings B (International Conference of B Users)*, volume 4355 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2007.

[CN02] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley, 2002.

[COC97] 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)*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.

[CR16] Néstor Cataño and Victor Rivera. EventB2Java: A Code Generator for Event-B. In *Proceedings NFM (NASA Formal Methods Symposium)*, volume 9690 of *Lecture Notes in Computer Science*, pages 166–171. Springer, 2016.

[CRD+21] Antonio Cerone, Markus Roggenbach, James Davenport, Casey Denner, Marie Farrell, Magne Haveraaen, Faron Moller, Philipp Körner, Sebastian Krings, Peter Olveczky, Bernd-Holger Schlingloff, Nikolay Shilov, and Rustam Zhumagambetov. Rooting Formal Methods within Higher Education Curricula for Computer Science and Software Engineering – A White Paper. In *Proceedings FMFun (International Workshop on Formal Methods - Fun for Everybody) 2019*, volume 1301 of *CCIS*. Springer, 2021.

[CRW+17] Mingshuai Chen, Anders P. Ravn, Shuling Wang, Mengfei Yang, and Naijun Zhan. A Two-Way Path Between Formal and Informal Design of Embedded Systems. In *Proceedings UTP (International Symposium on Unifying Theories of Programming)*, volume 10134 of *Lecture Notes in Computer Science*, pages 65–92. Springer, 2017.

[CSD+22] Cheng-Hao Cai, Jing Sun, Gillian Dobbie, Zhé Hóu, Hadrien Bride, Jin Song Dong, and Scott Uk-Jin Lee. Fast Automated Abstract Machine Repair Using Simultaneous Modifications and Refactoring. *Formal Aspects of Computing*, 2022.

[CVS18] Marek Chalupa, Martina Vitovská, and Jan Strejček. Symbiotic 5: Boosted Instrumentation. In *Proceedings TACAS (International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 10806 of *Lecture Notes in Computer Science*, pages 442–446. Springer, 2018.

[CWA+88] Mats Carlsson, Johan Widen, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog user's manual*, volume 3. Swedish Institute of Computer Science, 1988.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings FMSP (Workshop on Formal Methods in Software Practice)*, pages 7–15. ACM, 1998.

[Dan17] Chris Dannen. *Introducing Ethereum and Solidity*, volume 1. Springer, 2017.

[dAOMDM19] Diego de Azevedo Oliveira, Valério Medeiros, David Déharbe, and Martin A. Musicante. BTestBox: A Tool for Testing B Translators and Coverage of B Models. In *Proceedings TAP (International Conference on*

Tests and Proofs), volume 11823 of *Lecture Notes in Computer Science*, pages 83–92. Springer, 2019.

[DKS19] Jannik Dunkelau, Sebastian Krings, and Joshua Schmidt. Automated Backend Selection for ProB Using Deep Learning. In *Proceedings NFM (NASA Formal Methods Symposium)*, volume 11460 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2019.

[DL14] Ivaylo Dobrikov and Michael Leuschel. Optimising the ProB Model Checker for B using Partial Order Reduction. In *Proceedings SEFM (International Conference on Software Engineering and Formal Methods)*, volume 8702 of *Lecture Notes in Computer Science*, pages 220–234. Springer, 2014.

[DL16] 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.

[dMB08] Leonardo Mendonça 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.

[Dob17] Ivaylo Miroslavov Dobrikov. *Improving Explicit-State Model Checking for B and Event-B*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2017.

[ED07] Didier Essamé and Daniel Dollé. B in Large-Scale Projects: The Canarsie Line CBTC Experience. In *Proceedings B (International Conference of B Users)*, volume 4355 of *Lecture Notes in Computer Science*, pages 252–254. Springer, 2007.

[ESKK19] Alexandros Efremidis, Joshua Schmidt, Sebastian Krings, and Philipp Körner. Measuring Coverage of Prolog Programs Using Mutation Testing. In *Proceedings WFLP (International Workshop on Functional and Constraint Logic Programming) 2018*, volume 11285 of *Lecture Notes in Computer Science*. Springer, 2019.

[FBHL73] Abraham Adolf Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of set theory*, volume 67. Elsevier, 1973.

[FD19] Tomas Fischer and Dana Dghaym. Formal Model Validation Through Acceptance Tests. In *Proceedings RSSRail (International Conference on Reliability, Safety, and Security of Railway Systems)*, volume 11495 of *Lecture Notes in Computer Science*, pages 159–169, 2019.

[FFHS06] Hosam K. Fathy, Zoran S. Filipi, Jonathan Hagena, and Jeffrey L. Stein. Review of hardware-in-the-loop simulation and its prospects in the automotive area. In *Modeling and simulation for military applications*, volume 6228. SPIE, 2006.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings POPL (Symposium on Principles of Programming Languages)*, pages 110–121. ACM, 2005.

[FGG07] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77(1-2):71–103, 2007.

[FLP19] John S. Fitzgerald, Peter Gorm Larsen, and Ken Pierce. Multi-modelling and Co-simulation in the Engineering of Cyber-Physical Systems: Towards the Digital Twin. In *From Software Engineering to Formal Methods and Tools, and Back*, pages 40–55. Springer, 2019.

[Fra22] Adolf Fraenkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86(3):230–237, 1922.

[Frü09] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[Fuc92] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.

[FW86] Philip J. Fleming and John J. Wallace. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Communications of the ACM*, 29(3):218–221, 1986.

[GH96] Andrew Gravell and Peter Henderson. Executing formal specifications need not be harmful. *Software Engineering Journal*, 11(2):104–110, 1996.

[GH98] Etienne M. Gagnon and Laurie J. Hendren. *SableCC: An Object-Oriented Compiler Framework*. IEEE, 1998.

[GIL12] Gudmund Grov, Andrew Ireland, and Maria Teresa Llano. Refinement Plans for Informed Formal Design. In *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, VDM, and Z)*, volume 7316 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2012.

[GK91] Carlo Ghezzi and Richard A. Kennerer. Executing Formal Specifications: The ASTRAL to TRIO Translation Approach. In *Proceedings TAV (Symposium on Testing, Analysis, and Verification)*, pages 112–122. ACM, 1991.

[GL20] David Gelessus and Michael Leuschel. ProB and Jupyter for logic, set theory, theoretical computer science and formal methods. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2020.

[God90] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proceedings CAV (International Conference on Computer Aided Verification)*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.

[Goo] Google Guice Repository. `https://github.com/google/guice`. Accessed: 2020-02-27.

[GRHRW15] Thomas Gibson-Robinson, Henri Hansen, A. William Roscoe, and Xu Wang. Practical partial order reduction for CSP. In *Proceedings NFM (NASA Formal Methods Symposium)*, volume 9058 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2015.

[GTB+17] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: State of the art. *arXiv preprint arXiv:1702.00686*, 2017.

[Hal90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

[Hat07] Les Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. *Information and Software Technology*, 49(5):475–482, 2007.

[HE10] Bernhard G. Humm and Ralf S. Engelschall. Language-Oriented Programming Via DSL Stacking. In *Proceedings ICSOFT (International Conference on Software and Data Technologies)*, pages 279–287, 2010.

[Hen00] Thomas A. Henzinger. The theory of hybrid automata. In *Verification of digital and hybrid systems*, pages 265–292. Springer, 2000.

[Hic20] Rich Hickey. A History of Clojure. In *Proceedings HOPL (History of Programming Languages)*, pages 1–46. ACM, 2020.

[HJ89] Ian James Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–339, 1989.

[HL12] 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.

[HLK+20] Dominik Hansen, Michael Leuschel, Philipp Körner, Sebastian Krings, Thomas Naulin, Nader Nayeri, David Schneider, and Frank Skowron. Validation and real-life demonstration of ETCS hybrid level 3 principles using a formal B model. *Software Tools for Technology Transfer*, 22, 2020.

[HLS+18] 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 *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z)*, volume 10817 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2018.

[HLW+15] Dominik Hansen, Lukas Ladenberger, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. Validation of the ABZ Landing Gear System using ProB. In *ABZ 2014: The Landing Gear Case Study*, volume 433 of *CCIS*, pages 1–17. Springer, 2015.

[HMR+19] David Harel, Assaf Marron, Ariel Rosenfeld, Moshe Vardi, and Gera Weiss. Labor Division with Movable Walls: Composing Executable Specifications with Machine Learning and Search (Blue Sky Idea). In *Proceedings AAAI (Conference on Artificial Intelligence)*, volume 33, pages 9770–9774, 2019.

[Hoa69] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HR20] Frank Houdek and Alexander Raschke. Adaptive Exterior Light and Speed Control System. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 281–301. Springer, 2020.

[HS00] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In *SAT2000: Highlights of Satisfiability Research in the Year 2000 (Frontiers in Artificial Intelligence and Applications)*, pages 283–292. IOS Press, 2000.

[HSL16] Dominik Hansen, David Schneider, and Michael Leuschel. Using B and ProB for Data Validation Projects. In *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and*

Z), volume 9675 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2016.

[Ida20] Akram Idani. Meeduse: A Tool to Build and Run Proved DSLs. In *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 12546 of *Lecture Notes in Computer Science*, pages 349–367. Springer, 2020.

[ILR13] Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky. The SafeCap Platform for Modelling Railway Safety and Capacity. In *Proceedings SAFECOMP (International Conference on Computer Safety, Reliability, and Security)*, volume 8153 of *Lecture Notes in Computer Science*, pages 130–137. Springer, 2013.

[ILW+19a] Akram Idani, Yves Ledru, Abderrahim Ait Wakrime, Rahma Ben Ayed, and Philippe Bon. Towards a Tool-Based Domain Specific Approach for Railway Systems Modeling and Validation. In *Proceedings RSSRail (International Conference on Reliability, Safety, and Security of Railway Systems)*, volume 11495 of *Lecture Notes in Computer Science*, pages 23–40, 2019.

[ILW+19b] Akram Idani, Yves Ledru, Abderrahim Ait Wakrime, Rahma Ben Ayed, and Simon Collart-Dutilleul. Incremental Development of a Safety Critical System Combining Formal Methods and DSMLs. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 11687 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2019.

[ISO11] ISO. Road vehicles – Functional safety, 2011.

[Jac03] Daniel Jackson. Alloy: A Logical Modelling Language. In *Proceedings ZB (International Conference of B and Z Users)*, volume 2651 of *Lecture Notes in Computer Science*, pages 1–1. Springer, 2003.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

[JKSSS90] H. Jarvinen, Reino Kurki-Suonio, Markku Sakkinen, and Kari Systa. Object-oriented specification of reactive systems. In *Proceedings ICSE (International Conference on Software Engineering)*, pages 63–71. IEEE, 1990.

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *Ai Magazine*, 33(1):89–92, 2012.

[JLC15] Peter Würtz Vinther Jørgensen, Morten Larsen, and Luis Diogo Monteiro Duarte Couto. A code generation platform for VDM. In *Proceedings of the 12th Overture Workshop*, volume 1446 of *School of Computing Science Technical Report Series*, pages 21–35. School of Computing Science, University of Newcastle upon Tyne, 2015.

[JLY+19] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. An Empirical Study Assessing Source Code Readability in Comprehension. In *Proceedings ICSME (International Conference on Software Maintenance and Evolution)*, pages 513–523. IEEE, 2019.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.

[KB18] Philipp Körner and Jens Bendisposto. Distributed Model Checking Using ProB. In *Proceedings NFM (NASA Formal Methods Symposium)*, volume 10811 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2018.

[KBD+19] Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. Embedding High-Level Formal Specifications into Applications. In *Proceedings FM (International Symposium on Formal Methods)*, volume 11800 of *Lecture Notes in Computer Science*, pages 519–535. Springer, 2019.

[KBD+20] Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. Integrating formal specifications into applications: the ProB Java API. *Formal Methods in System Design*, 57:160–187, 2020.

[KD06] Timo Käköla and Juan Carlos Duenas. *Software product lines*. Springer, 2006.

[KHDB06] Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In *Proceedings SPIN (International SPIN Workshop on Model Checking of Software)*, volume 3925 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2006.

[KK18] Philipp Körner and Sebastian Krings. plspec - A Specification Language for Prolog Data. In *Proceedings Declare (International Workshop on Functional and Constraint Logic Programming) 2017*, volume 10997 of *LNAI*. Springer, 2018.

[KK21a] Philipp Körner and Sebastian Krings. Increasing Student Self-Reliance and Engagement in Model-Checking Courses. In *Proceedings FMTea (Formal Methods Teaching)*, pages 60–74. Springer, 2021.

[KK21b] Sebastian Krings and Philipp Körner. Prototyping Games Using Formal Methods. In *Proceedings FMFun (International Workshop on Formal Methods - Fun for Everybody) 2019*, volume 1301 of *CCIS*. Springer, 2021.

[KKDR20] Sebastian Krings, Philipp Körner, Jannik Dunkelau, and Chris Rutenkolk. A Verified Low-Level Implementation of the Adaptive Exterior Light and Speed Control System. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 382–397. Springer, 2020.

[KL16] Sebastian Krings and Michael Leuschel. SMT solvers for validation of B and Event-B models. In *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 9681 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 2016.

[KL18] Philipp Körner and Michael Leuschel. Embedding Formal Specifications as Libraries into Applications. Technical Report cs-tr-1525, School of Computing, Newcastle University, December 2018.

[KL23] Philipp Körner and Michael Leuschel. Towards Practical Partial Order Reduction for High-Level Formalisms. In *Proceedings VSTTE (International Conference on Verified Software: Theories, Tools, and Experiments) 2022*, volume 13800 of *Lecture Notes in Computer Science*. Springer, 2023. To appear.

[KLB+22] 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.

[KLD20] Philipp Körner, Michael Leuschel, and Jannik Dunkelau. Towards a Shared Specification Repository. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2020.

[KLK+18] Sebastian Krings, Michael Leuschel, Philipp Körner, Stefan Hallerstede, and Miran Hasanagić. Three Is a Crowd: SAT, SMT and CLP on a Chessboard. In *Proceedings PADL (International Symposium on Practical Aspects of Declarative Languages)*, volume 10702 of *Lecture Notes in Computer Science*. Springer, 2018.

[KLS+20] Sebastian Krings, Michael Leuschel, Joshua Schmidt, David Schneider, and Marc Frappier. Translating Alloy and extensions to classical B. *Science of Computer Programming*, 188:1–25, 2020.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[KM22] Philipp Körner and Florian Mager. An Embedding of B in Clojure. In *Companion Proceedings MODELS (International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings)*, page 598–606. ACM, 2022.

[KML18] Philipp Körner, Jeroen Meijer, and Michael Leuschel. State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin. In *Proceedings iFM (International Conference on integrated Formal Methods)*, volume 11023 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2018.

[KNP12] Marta Kwiatkowska, Gethin Norman, and David Parker. The PRISM benchmark suite. In *Proceedings QEST (International Conference on the Quantitative Evaluation of Systems)*, pages 203–204. IEEE, 2012.

[Kör14] Philipp Körner. Improving Distributed Model Checking in ProB. Bachelor's thesis, Heinrich Heine Universität Düsseldorf, August 2014.

[Kör17] Philipp Körner. An Integration of ProB and LTSmin. Master's thesis, Heinrich Heine Universität Düsseldorf, February 2017.

[Kri17] Sebastian Krings. *Towards Infinite-State Symbolic Model Checking for B and Event-B*. PhD thesis, Universitäts-und Landesbibliothek der HHU Düsseldorf, 2017.

[KSB+18] Sebastian Krings, Joshua Schmidt, Carola Brings, Marc Frappier, and Michael Leuschel. A Translation from Alloy to B. In *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z)*, volume 10817 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2018.

[KSL21] Philipp Körner, David Schneider, and Michael Leuschel. On the Performance of Bytecode Interpreters in Prolog. In *Proceedings WFLP (International Workshop on Functional and Constraint Logic Programming) 2020*, volume 12560 of *Lecture Notes in Computer Science*. Springer, 2021.

[Kup18] Markus A. Kuppe. Let TLA+ RiSE. RiSE group all-hands meeting, August 2018.

[Lad17] Lukas Ladenberger. *Rapid creation of interactive formal prototypes for validating safety-critical systems*. PhD thesis, HHU Düsseldorf, 2017.

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002.

[Lam09] Leslie Lamport. The PlusCal Algorithm Language. In *Proceedings ICTAC (International Colloquium on Theoretical Aspects of Computing)*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009.

[LB03] 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.

[LB08] Michael Leuschel and Michael Butler. ProB: an automated analysis toolset for the B method. *Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[LB11] Michael Leuschel and Jens Bendisposto. Directed Model Checking for B: An Evaluation and New Techniques. In *Formal Methods: Foundations and Applications*, volume 6527 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2011.

[LB16] Michael Leuschel and Egon Börger. A compact encoding of sequential ASMs in Event-B. In *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z)*, volume 9675 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2016.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John S. Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The overture initiative integrating tools for VDM. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.

[LBH14] Michael Leuschel, Jens Bendisposto, and Dominik Hansen. Unlocking the Mysteries of a Formal Model of an Interlocking System. In *Proceedings Rodin Workshop 2014*, 2014.

[LBL12] Thierry Lecomte, Lilian Burdy, and Michael Leuschel. Formally Checking Large Data Sets in the Railways. *CoRR*, abs/1210.6815, 2012. Proceedings of DS-Event-B.

[LCB09] Michael Leuschel, Dominique Cansell, and Michael Butler. Validating and Animating Higher-Order Recursive Functions in B. In Jean-Raymond Abrial and Uwe Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, pages 78–92. Springer, 2009.

[Lec14a] Thierry Lecomte. *Atelier B*, chapter 2, pages 35–46. Wiley, 2014.

[Lec14b] Thierry Lecomte. Return of Experience on Automating Refinement in B. In *Proceedings SETS (Workshop about Sets and Tool)*, 2014.

[Leu08] Michael Leuschel. The high road to formal validation. In *Proceedings ABZ (International Conference on Abstract State Machines, B and Z)*, volume 5238 of *Lecture Notes in Computer Science*, pages 4–23. Springer, 2008.

[Leu20] Michael Leuschel. Fast and effective well-definedness checking. In *Proceedings IFM (International Conference on Integrated Formal Methods)*, volume 12546 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2020.

[Leu21] Michael Leuschel. Spot the Difference: A Detailed Comparison Between B and Event-B. In *Logic, Computation and Rigorous Methods*, volume 12750 of *Lecture Notes in Computer Science*, pages 147–172. Springer, 2021.

[LFW+16] Peter Gorm Larsen, John S. Fitzgerald, Jim Woodcock, Peter Fritzson, Jörg Brauer, Christian Kleijn, Thierry Lecomte, Markus Pfeil, Ole Green, Stylianos Basagiannis, and Andrey Sadovykh. Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project. In *Proceedings CPS Data (International Workshop on Modelling, Analysis, and Control of Complex CPS)*, pages 1–6. IEEE, 2016.

[LIL15] Kenneth Lausdahl, Hiroshi Ishikawa, and Peter Gorm Larsen. Interpreting implicit VDM specifications using ProB. In *Proceedings of the 12th Overture Workshop*, volume 1446 of *School of Computing Science Technical Report Series*, pages 6–20. School of Computing Science, University of Newcastle upon Tyne, 2015.

[LL16] Lukas Ladenberger and Michael Leuschel. BMotionWeb: A Tool for Rapid Creation of Formal Prototypes. In *Proceedings SEFM (International Conference on Software Engineering and Formal Methods)*, volume 9763 of *Lecture Notes in Computer Science*, pages 403–417. Springer, 2016.

[LMW20] Michael Leuschel, Mareike Mutz, and Michelle Werth. Modelling and Validating an Automotive System in Classical B and Event-B. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 335–350. Springer, 2020.

[Log] Logic Calculators. `https://web.archive.org/web/20120418155039/http://research.microsoft.com/en-us/um/people/lamport/tla/logic-calculators.html`. Accessed: 2020-02-27.

[LPVDPH16] Alfons Laarman, Elwin Pater, Jaco Van De Pol, and Henri Hansen. Guard-based partial-order reduction. *Software Tools for Technology Transfer*, 18(4):427–448, 2016.

[MBC+16] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *Proceedings FSE (International Symposium on Foundations of Software Engineering)*, pages 373–383. ACM, 2016.

[MF20] Amel Mammar and Marc Frappier. Modeling of a Speed Control System Using Event-B. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2020.

[MFL20] Amel Mammar, Marc Frappier, and Régine Laleau. An Event-B Model of an Automotive Adaptive Exterior Light System. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2020.

[mis13] *MISRA C:2012 – Guidelines for the use of the C language in critical systems*. MISRA, 2013.

[MJ14] Ido Milicevic, Aleksandar Erfrati and Daniel Jackson. aRby—An Embedding of Alloy in Ruby. In *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM and Z)*, volume 8477 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2014.

[MS11] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-B models. In *Proceedings SoICT (Symposium on Information and Communication Technology)*, pages 179–188. ACM, 2011.

[NFM17] Elisa Negri, Luca Fumagalli, and Marco Macchi. A Review of the Roles of Digital Twin in CPS-based Production Systems. *Procedia Manufacturing*, 11:939–948, 2017.

[NKK19] Falco Nogatz, Philipp Körner, and Sebastian Krings. Prolog Coding Guidelines: Status and Tool Support. In *Proceedings ICLP (International Conference on Logic Programming) (Technical Communications)*, volume 306 of *EPTCS*, 2019.

[NLL12] Claus Ballegaard Nielsen, Kenneth Lausdahl, and Peter Gorm Larsen. Combining VDM with executable code. In *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, VDM and Z)*, volume 7316 of *Lecture Notes in Computer Science*, pages 266–279. Springer, 2012.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 284–295. IEEE, 2005.

[Num13] Timo Nummenmaa. *Executable formal specifications in game development: Design, validation and evolution*. PhD thesis, University of Tampere, 2013.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Proceedings CAV (International Conference on Computer Aided Verification)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings CAV (International Conference on Computer Aided Verification)*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.

[Pel07] Radek Pelánek. BEEM: benchmarks for explicit model checkers. In *Proceedings SPIN (International SPIN Workshop on Model Checking of Software)*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2007.

[PF15] Simon Poulding and Robert Feldt. Heuristic model checking using a monte-carlo tree search algorithm. In *Proceedings GECCO (Conference on Genetic and Evolutionary Computation)*, pages 1359–1366. ACM, 2015.

[PFB19] Camille Parillaud, Yoann Fonteneau, and Fabien Belmonte. Interlocking Formal Verification at Alstom Signalling. In *Proceedings RSSRail (International Conference on Reliability, Safety, and Security of Railway Systems)*, volume 11495 of *Lecture Notes in Computer Science*, pages 215–225. Springer, 2019.

[PL07] 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.

[PL12] 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.

[PlU] PlüS. `https://plues.github.io/en/index/`. Accessed: 2020-02-27.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings SFCS (Symposium on Foundations of Computer Science)*, pages 46–57. IEEE, 1977.

[Proa] ProB Java API Source Code. `https://github.com/hhu-stups/prob2_kernel`. Accessed: 2020-03-11.

[Prob] The ProB Logic Calculator. `https://github.com/hhu-stups/prob-logic-calculator`. Accessed: 2020-07-10.

[Proc] ProB Java API Example Source Code. `https://github.com/hhu-stups/executable_spec_example`. Accessed: 2020-03-11.

[Prod] ProB Maven Artifacts. `https://search.maven.org/artifact/de.hhu.stups/de.prob2.kernel`. Accessed: 2020-03-11.

[RC07] Joris Rehm and Dominique Cansell. Proved Development of the Real-Time Properties of the IEEE 1394 Root Contention Protocol with the Event B Method. In *Proceedings ISoLA (International Symposium on Leveraging Applications of Formal Methods, Verification and Validation)*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*, pages 179–190. Cépaduès-Éditions, 2007.

[RCWR17] Victor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. Code generation for Event-B. *Software Tools for Technology Transfer*, 19(1):31–52, 2017.

[RGS19] Daniel Ratiu, Marco Gario, and Hannes Schoenhaar. FASTEN: An Open Extensible Framework to Experiment With Formal Specification Approaches. In *Proceedings FormaliSE (Workshop on Formal Methods in Software Engineering)*, pages 41–50. IEEE, 2019.

[RNM+21] Daniel Ratiu, Arne Nordmann, Peter Munk, Carmen Carlan, and Markus Voelter. FASTEN: An Extensible Platform to Experiment with Rigorous Modeling of Safety-Critical Systems. In *Domain-Specific Languages in Practice*, pages 131–164. Springer, 2021.

[Rod13] Maria Teresa Llano Rodriguez. *Invariant discovery and refinement plans for formal modelling in Event-B*. PhD thesis, Heriot-Watt University, UK, 2013.

[Roß22] Jan Roßbach. Boolean Encoding of Statically Finite Sets in B Machines. Bachelor's thesis, Heinrich Heine Universität Düsseldorf, March 2022.

[SA92] Michael Spivey and Jean-Raymond Abrial. *The Z Notation*. Prentice-Hall, 1992.

[SB06] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM TOSE (Transactions on Software Engineering and Methodology)*, 15(1):92–122, 2006.

[Sch17] David Schneider. *Constraint Modelling and Data Validation Using Formal Specification Languages*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2017.

[SH09] Jan-Georg Smaus and Jörg Hoffmann. Relaxation refinement: A new method to generate heuristic functions. In *Proceedings MoChArt (International Workshop on Model Checking and Artificial Intelligence) 2008*, volume 5348 of *LNAI*, pages 147–165. Springer, 2009.

[Sie19] Stephen F. Siegel. What's Wrong with On-the-Fly Partial Order Reduction. In *Proceedings CAV (International Conference on Computer Aided Verification)*, volume 11562 of *Lecture Notes in Computer Science*, pages 478–495. Springer, 2019.

[SKL18] 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*, pages 346–366. Springer, 2018.

[SL21] Joshua Schmidt and Michael Leuschel. Improving SMT Solver Integrations for the Validation of B and Event-B Models. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 12863 of *Lecture Notes in Computer Science*, pages 107–125. Springer, 2021.

[SLF+12] Aymerick Savary, Jean-Louis Lanet, Marc Frappier, Tiana Razafindralambo, and Josselin Dolhen. VTG - Vulnerability Test Generator, a Plug-in for Rodin. In *Proceedings Workshop Deploy 2012*, Fontainebleau, France, 2012.

[SLW15] David Schneider, Michael Leuschel, and Tobias Witt. Model-Based Problem Solving for University Timetable Validation and Improvement. In *Proceedings FM (International Symposium on Formal Methods)*, volume 9109 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015.

[SLW18] David Schneider, Michael Leuschel, and Tobias Witt. Model-based Problem Solving for University Timetable Validation and Improvement. *Formal Aspects of Computing*, pages 545–569, 2018.

[SP08] Michael Short and Michael J. Pont. Assessment of high-integrity embedded automotive control systems using hardware in the loop simulation. *Journal of Systems and Software*, 81(7):1163–1183, 2008.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *Proceedings IJCAI (International Joint Conference on Automated Reasoning)*, volume 8562 of *Lecture Notes in Computer Science*, pages 367–373. Springer, 2014.

[Sut17] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[TB13] Emina Torlak and Rastislav Bodik. Growing Solver-Aided Languages With Rosette. In *Proceedings Onward! (International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software)*, pages 135–152. ACM, 2013.

[TJ07] 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.

[TJLL18] Peter Würtz Vinther Tran-Jørgensen, Peter Gorm Larsen, and Gary T. Leavens. Automated translation of VDM to JML-annotated Java. *Software Tools for Technology Transfer*, 20(2):211–235, 2018.

[TLG+19] Casper Thule, Kenneth Lausdahl, Cláudio Gomes, Gerd Meisl, and Peter Gorm Larsen. Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory*, 92:45–61, 2019.

[TLL18] Casper Thule, Kenneth Lausdahl, and Peter Gorm Larsen. Overture FMU: Export VDM-RT Models as Tool-Wrapper FMUs. In *Proceedings of the 16th Overture Workshop*, volume 1524 of *School of Computing Science Technical Report Series*, pages 23–38. School of Computing Science, University of Newcastle upon Tyne, 2018.

[TN16] Casper Thule and René Nilsson. Considering Abstraction Levels on a Case Study. In *Proceedings of the 14th Overture Workshop: Towards Analytical Tool Chains*, pages 16–31, 2016.

[Val89] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings ICATPN (International Conference on Application and Theory of Petri Nets)*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[VBL22] Fabian Vu, Dominik Brandt, and Michael Leuschel. Model Checking B Models via High-level Code Generation. In *Proceedings ICFEM (International Conference on Formal Engineering Methods)*, Lecture Notes in Computer Science, 2022. To appear.

[VHKL19] Fabian Vu, Dominik Hansen, Philipp Körner, and Michael Leuschel. A Multi-Target Code Generator for High-Level B. In *Proceedings iFM (International Conference on integrated Formal Methods)*, volume 11918 of *Lecture Notes in Computer Science*, pages 456–473. Springer, 2019.

[Vu18] Fabian Vu. A High-Level Code Generator for Safety Critical B Models. Bachelor's thesis, Heinrich Heine Universität Düsseldorf, August 2018.

[VW86] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings LICS (Symposium in Logic in Computer Science)*. IEEE, 1986.

[Weg72] Peter Wegner. The Vienna definition language. *ACM Computing Surveys*, 4(1):5–63, 1972.

[WK21] Isabel Wingen and Philipp Körner. Effectiveness of Annotation-Based Static Type Inference. In *Proceedings WFLP (International Workshop on Functional and Constraint Logic Programming) 2020*, volume 12560 of *Lecture Notes in Computer Science*, pages 74–93. Springer, 2021.

[WL20] Michelle Werth and Michael Leuschel. VisB: A lightweight tool to visualize formal models with SVG graphics. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2020.

[WLB00] Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing formal specifications with concurrent constraint programming. *Automated Software Engineering*, 7(4):315–343, 2000.

[WRM18] Nathaniel Watson, Steve Reeves, and Paolo Masci. Integrating User Design and Formal Models within PVSio-Web. In *Proceedings F-IDE (Formal Integrated Development Environment)*, pages 95–104, 2018.

[YICD20] Asfand Yar, Akram Idani, and Simon Collart-Dutilleul. Merging Railway Standard Notations in a Formal DSL-Based Framework. In *Proceedings ECSA (European Conference on Software Architecture)*, volume 1269 of *CCIS*, pages 411–419. Springer, 2020.

[YJS13] Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières. JeB: Safe simulation of Event-B models in javascript. In *Proceedings APSEC (Asia-Pacific Conference on Software Engineering), Volume 1*, pages 571–576. IEEE, 2013.

[YML99] 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)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

[YSAA97] Jun Yuan, Jian Shen, Jacob Abraham, and Adnan Aziz. On combining formal and informal verification. In *Proceedings CAV (International Conference on Computer Aided Verification)*, volume 1254 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 1997.

[YZ16] Mengfei Yang and Naijun Zhan. Combining Formal and Informal Methods in the Design of Spacecrafts. In *Engineering Trustworthy Software Systems*, volume 9506 of *Lecture Notes in Computer Science*, pages 290–323. Springer, 2016.

[ZGS14] Simone Zenzaro, Vincenzo Gervasi, and Jacopo Soldani. WebASM: an abstract state machine execution environment for the web. In *Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM and Z)*, volume 8477 of *Lecture Notes in Computer Science*, pages 216–221. Springer, 2014.

Information About Included Manuscripts

While I can honestly claim major contributions to all articles presented in this thesis, parts of the articles have been written by Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, Michael Leuschel, Florian Mager and Kristin Rutenkolk. Behind the scenes, many more people were involved with implementation work, especially several students writing their theses at the STUPS group.

In the following, I try to acknowledge everyone's contribution as fairly as possible. Special thanks to everyone mentioned here for supporting my research.

Integrating formal specifications into applications: the ProB Java API

The article "Integrating formal specifications into applications: the ProB Java API" is published in the journal "Formal Methods in System Design" [KBD+20]. It is part of the special issue containing extended versions of the "Best Papers from FM 2019".

A short version was informally published as part of the Event-B Day 2018 [KL18]. The first peer-reviewed publication was part of the proceedings of the Third World Congress on Formal Methods (sub-titled "The Next 30 Years") (FM 2019) [KBD+19].

The informal manuscript of the Event-B Day was written by Philipp Körner and Michael Leuschel. Both published articles are co-authored by Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings and Michael Leuschel.

Design and architecture of ProB's Java API was mainly done by Jens Bendisposto. Significant implementation work was done by Joy Clark.

The Pac-Man case study has been created by Christoph Heinzen and performance improvements were made by David Geleßus. The chess case study has been created by Philip Höfges. The ProB logic calculator presented in the article has been implemented by Jens Bendisposto. A prior version written in PHP and the Prolog evaluation logic was developed by Michael Leuschel. *lisb* was developed by Philipp Körner. Main contributors of the PlüS case study are David Schneider, Joshua Schmidt, Tobias Witt and Philip Höfges. Many people were involved in the ETCS HL3 project: the embedded B model was written by Dominik Hansen and Michael Leuschel.

The idea to use B models at runtime was promoted by Michael Leuschel, in particular for HL3 and PlüS [SLW15]; an early antecedent is to use B for functional programming [LCB09]. This is very much related to the concept of "Digital Twins" yet is not identical (though it matches some definitions of the term in literature [NFM17, Table 1]).

Philipp Körner's contributions to the article are:

- creation of an initial draft,
- code contributions to the ProB Java API and ETCS HL3 case study,
- description of the ProB Java API,
- presentation of the Pac-Man, Chess, *lisb* and PlüS case studies,
- discussion regarding executability,
- discussion of related VDM-based approaches, FMUs and hybrid systems.

Jens Bendisposto's, Jannik Dunkelau's, Sebastian Krings' and Michael Leuschel's contributions to the article are:

- implementation, maintenance of and example code for the ProB Java API,
- implementation of and text about the ProB logic calculator and the ETCS HL3 case study,
- details on B and ProB,
- discussion of lessons learned,
- comparison with code generation and related tools,
- discussion of potential of AI integration.

Full bibliographic references:

- Philipp Körner and Michael Leuschel. Embedding Formal Specifications as Libraries into Applications. Technical Report cs-tr-1525, School of Computing, Newcastle University, December 2018
- Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. Embedding High-Level Formal Specifications into Applications. In *Proceedings FM (International Symposium on Formal Methods)*, volume 11800 of *Lecture Notes in Computer Science*, pages 519–535. Springer, 2019
- Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. Integrating formal specifications into applications: the ProB Java API. *Formal Methods in System Design*, 57:160–187, 2020

A Verified Low-Level Implementation and Visualization of the Adaptive Exterior Light and Speed Control System

The manuscript in chapter 3 is based on an article that is part of the ABZ 2020 conference proceedings [KKDR20] and is an extended version that has been submitted to

the International Journal on Software Tools for Technology Transfer (STTT). The currently submitted version describes validation efforts that later proved to be incomplete. Thus, clearly marked corrigenda are inserted to highlight any changes I made to the manuscript later on.

The new manuscript is submitted under the same title as above. Both versions have been co-authored by Sebastian Krings, Philipp Körner, Jannik Dunkelau and Kristin Rutenkolk. The initial idea for this article was proposed by Sebastian Krings.

Philipp Körner's contributions to the article are:

- design of the development approach (equally with Sebastian Krings),
- implementation and testing of the adaptive exterior light system (ELS) (equally with Sebastian Krings),
- initial verification attempts using CBMC (equally with Sebastian Krings).
- verification using CBMC.

Sebastian Krings', Jannik Dunkelau's and Kristin Rutenkolk's contributions to the article are:

- implementation and testing of the speed control system (SCS),
- additional testing of the ELS for American cars,
- visualisation of the program state,
- comparison with the other case studies.

Full bibliographic reference: Sebastian Krings, Philipp Körner, Jannik Dunkelau, and Chris Rutenkolk. A Verified Low-Level Implementation of the Adaptive Exterior Light and Speed Control System. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 382–397. Springer, 2020

Treating Specifications as Data

The chapter "Treating Specifications as Data" was published under the title "An Embedding of B in Clojure" in the proceedings of the 19th Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa 2022). It is part of the Companion Proceedings of the 25th International Conference on Model Driven Engineering Languages and System (MODELS 2022). The manuscript is co-authored by Philipp Körner and Florian Mager. Recently, the article was invited to be extended for a special issue of Innovations in Systems and Software Engineering (a NASA Journal, Springer).

The first idea of embedding B in Clojure was suggested by David Schneider.

Philipp Körner's contributions to the article are:

- implementation and design of *lisb*,
- idea to cover the entirety of B,
- idea to use meta-programming techniques for B tools and DSLs,
- implementation of the algorithm DSL in *lisb* (with Florian Mager).

Florian Mager's contributions to the article are:

- additional implementation work covering B machines in *lisb*,
- details on the current internals of *lisb*.

The automatic refinement tool discussed in the article was developed by Jan Roßbach under the supervision of Philipp Körner.

Full bibliographic reference: Philipp Körner and Florian Mager. An Embedding of B in Clojure. In *Companion Proceedings MODELS (International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings)*, page 598–606. ACM, 2022

Towards a Shared Specification Repository

This article was published as a short paper at the 7th International Conference on Rigorous State-Based Methods (ABZ 2020) [KLD20].

It was co-authored by Philipp Körner, Michael Leuschel and Jannik Dunkelau.

Philipp Körner's contributions to the article are:

- benchmarking, organisation and presentation of the ProB examples.

Michael Leuschel's and Jannik Dunkelau's contributions to the article are:

- initial collection of the machine files,
- addition to the features to collect data about,
- additional benchmark execution,
- additions to related work.

Full bibliographic reference: Philipp Körner, Michael Leuschel, and Jannik Dunkelau. Towards a Shared Specification Repository. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12071 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2020

Empirical Evaluation of POR for B

The short manuscript in chapter 6 is included in this thesis for adequate background and motivation for the last article in chapter 7. Originally, these two chapters are part of the same (published) manuscript; Due to page limitations during an earlier submission, they had been split into two. Thus, this small chapter is an extended version of section 3 of Philipp Körner and Michael Leuschel. Towards Practical Partial Order Reduction for High-Level Formalisms. In *Proceedings VSTTE (International Conference on Verified Software: Theories, Tools, and Experiments) 2022*, volume 13800 of *Lecture Notes in Computer Science*. Springer, 2023. To appear..

The manuscript is co-authored by Philipp Körner and Michael Leuschel.

Philipp Körner's contributions to the manuscript are:

- creation of the initial draft,
- setup and execution of the empirical evaluation,
- presentation of the results.

Michael Leuschel's contributions to the manuscript are:

- improvements on the presentation,
- highlighting threats to validity.

Full bibliographic reference: Parts of the chapter have been published in: Philipp Körner and Michael Leuschel. Towards Practical Partial Order Reduction for High-Level Formalisms. In *Proceedings VSTTE (International Conference on Verified Software: Theories, Tools, and Experiments) 2022*, volume 13800 of *Lecture Notes in Computer Science*. Springer, 2023. To appear.

Towards Practical Partial Order Reduction for High-Level Formalisms

This chapter was presented at the 14th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2022) and was accepted for inclusion in the proceedings. It is a version of this article without its section 3 (which is a shorter version of the previous chapter of this thesis). As above, the manuscript is co-authored by Philipp Körner and Michael Leuschel.

Philipp Körner's contributions to the article are:

- creation of the initial draft,
- identification of the idioms hindering POR,
- proposal and evaluation of solutions,
- details on the effectiveness of dynamic POR.

Michael Leuschel's contributions to the article are:

- improvements wrt. correctness, consistency, and clarity,
- selection and description of the grand challenge (see [LBH14]),
- proving in Rodin that the operation `route_freeing(R)` cannot be co-enabled with `route_formation(R)` (and, thus, `point_positionning(R)`) (see section 7.5.2),
- initial discussion about dynamic POR.

Full bibliographic reference: A version of this chapter has been published in: Philipp Körner and Michael Leuschel. Towards Practical Partial Order Reduction for High-Level Formalisms. In *Proceedings VSTTE (International Conference on Verified Software: Theories, Tools, and Experiments) 2022*, volume 13800 of *Lecture Notes in Computer Science*. Springer, 2023. To appear.

List of Figures

List of Listings

List of Tables