Simulation and Code Generation for Validation and Verification of Formal B Models

Inaugural-Dissertation

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

vorgelegt von

Fabian Vu

aus Solingen

Düsseldorf, Oktober 2025

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

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

Berichtererstatter:

1. Prof. Dr. Michael Leuschel
   Heinrich-Heine-Universität Düsseldorf
2. Prof. Dr. Angelo Gargantini
   Università degli Studi di Bergamo

Tag der mündlichen Prüfung: 15. Mai 2025

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

- Fabian Vu, Michael Leuschel, and Atif Mashkoor. Validation of Formal Models by Timed Probabilistic Simulation. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12709 of *Lecture Notes of Computer Science*, pages 81–96, Springer, 2021. doi: 10.1007/978-3-030-77543-8_6

- Fabian Vu and Michael Leuschel. Validation of Formal Models by Interactive Simulation. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 14010 of *Lecture Notes of Computer Science*, pages 59–69, Springer, 2023. doi: 10.1007/978-3-031-33163-3_5

- David Geleßus, Sebastian Stock, Fabian Vu, Michael Leuschel, and Atif Mashkoor. Modeling and Analysis of a Safety-Critical Interactive System Through Validation Obligations. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 14010 of *Lecture Notes of Computer Science*, pages 284–302, Springer, 2023. doi: 10.1007/978-3-031-33163-3_22

- Fabian Vu, Jannik Dunkelau, and Michael Leuschel. Validation of Reinforcement Learning Agents and Safety Shields with ProB. In *Proceedings NFM (International Symposium on NASA Formal Methods)*, volume 14627 of *Lecture Notes of Computer Science*, pages 279-297, Springer, 2024. doi: 10.1007/978-3-031-60698-4_16

- 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)*, volume 13478 of *Lecture Notes of Computer Science*, pages 334–351, Springer, 2022. doi: 10.1007/978-3-031-17244-1_20

- Fabian Vu, Christopher Happe, and Michael Leuschel. Generating Domain-Specific Interactive Validation Documents. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 13487 of *Lecture Notes of Computer Science*, pages 32–49, Springer, 2022. doi: 10.1007/978-3-031-15008-1_4

- Fabian Vu, Christopher Happe, and Michael Leuschel. Generating interactive documents for domain-specific validation of formal models. In *STTT Journal (International Journal on Software Tools for Technology Transfer)*, 26(2):147–168, 2024. doi: 10.1007/s10009-024-00739-0

Other peer-reviewed publications

- 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 of Computer Science*, pages 456–473, Springer, 2019. doi: 10.1007/978-3-030-34968-4_25

- Jens Bendisposto, David Geleßus, Yumiko Jansing, Michael Leuschel, Antonia Pütz, Fabian Vu, and Michelle Werth. ProB2-UI: A Java-Based User Interface for ProB. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 12863 of *Lecture Notes of Computer Science*, pages 193–201, Springer, 2021. doi: 10.1007/978-3-030-85248-1_12

- Sebastian Stock, Fabian Vu, David Geleßus, Michael Leuschel, Atif Mashkoor, and Alexander Egyed. Validation by Abstraction and Refinement. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 14010 of *Lecture Notes of Computer Science*, pages 160–178, Springer, 2023. doi: 10.1007/978-3-031-33163-3_12

- Jan Gruteser, David Geleßus, Michael Leuschel, Jan Roßbach, and Fabian Vu. A Formal Model of Train Control with AI-Based Obstacle Detection. In *Proceedings RSSRail (International Conference on Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification)*, volume 14198 of *Lecture Notes of Computer Science*, pages 128-145, Springer, 2023. doi: 10.1007/978-3-031-43366-5_8

- Jan Gruteser, Jan Roßbach, Fabian Vu, and Michael Leuschel. Using Formal Models, Safety Shields and Certified Control to Validate AI-Based Train Systems. In *Proceedings FMAS (International Workshop on Formal Methods for Autonomous Systems)*, volume 411 of *Electronic Proceedings in Theoretical Computer Science*, pages 151–159, Open Publishing Association, 2024. doi: 10.4204/EPTCS.411.10

Other publications

- Michael Leuschel, Fabian Vu, and Kristin Rutenkolk. Case Study: Safety Controller for Autonomous Driving on Highways. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 15728 of *Lecture Notes of Computer Science*, pages 203–211. Springer, 2025. doi: 10.1007/978-3-031-94533-5_12

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 18. Dezember 2024

Fabian Vu

The pilot looked at his cues of attitude and speed and orientation and so on and responded as he would from the same cues in an airplane, but there was no way it flew the same. The simulators had shown us that.

Alan Shepard

Abstract

This thesis addresses simulation and code generation for high-level formal models and contains manuscripts I co-authored on the two main topics.

The first part presents a technique called *timed probabilistic simulation*, which allows formal models to be simulated with timing and probabilistic behavior. The underlying concept is based on activations, which describe how events trigger one another. To this end, we implement the SimB simulator built on ProB's animator. SimB supports Real-Time Simulation, Monte Carlo Simulation, and statistical validation techniques, such as hypothesis testing and estimation of likelihood and values.

In the first use case, we use SimB with domain-specific visualization in VisB to create real-time prototypes for safety-critical systems that involve human interactions and automatic system events. Therefore, we present a technique called *interactive simulation* that allows us to simulate system reactions in response to user interactions. As case studies, we present real-time prototypes for a vehicle's exterior light system and an air traffic control system.

In the second use case, we use SimB to simulate and validate systems with artificial intelligence (AI). In this approach, a real AI runs simulations via SimB, while a formal model serves as a safety shield at runtime, i.e., as a runtime monitor. The safety shield defines which actions are safe, ensuring that the AI always executes safe actions. SimB's validation techniques help to identify vulnerabilities in the AI and the safety shield. We evaluate this approach on a highway AI trained with reinforcement learning, demonstrating that an established technique called Responsibility-Sensitive Safety improves safety. We also discuss broader applications of this approach for AI systems.

In the second part, we evaluate code generation of high-level B models via B2Program.

First, we extend B2Program to generate model checking code, aiming for high performance. To achieve this goal, we implement multiple techniques, including parallelization and caching. We implement ProB's operation reuse technique for B2Program and show that the technique also improves the model checking performance for B2Program. For most models, B2Program achieves a better performance than ProB and a similar performance to TLC. In this thesis, we considered the target languages Java, JavaScript/TypeScript, and C++.

Second, we use B2Program to generate HTML documents that support animation, simulation, domain-specific visualization in VisB, and feedback for validating B models. This approach enables domain experts to be involved in the validation process of the formal model. As case studies, we evaluate a vehicle's exterior light system and a landing gear system.

Zusammenfassung

Diese Arbeit befasst sich mit Simulation und Codegenerierung von high-level formalen Modellen und enthält Manuskripte, die von mir mitverfasst worden sind.

Im ersten Teil präsentieren wir eine Technik der *Zeitgesteuerten Probabilistischen Simulation* von formalen Modellen. Das Konzept basiert auf Aktivierungen, die beschreiben, wie Ereignisse sich gegenseitig auslösen. Wir implementieren den neuen Simulator SimB, welcher auf ProBs Animator aufsetzt. SimB unterstützt Echtzeitsimulation, Monte-Carlo Simulation, und statistische Techniken wie Hypothesentest und Schätzung von Wahrscheinlichkeiten und Werten.

Als Erstes verwenden wir SimB mit domänenspezifischer Visualisierung in VisB, um Echtzeit-Prototypen für sicherheitskritische Systeme mit menschlichen Interaktionen und automatischen Ereignissen zu erstellen. Wir präsentieren die Technik der *Interaktiven Simulation*, die es ermöglicht, Systemreaktionen auf Nutzerinteraktionen zu simulieren. Als Fallstudien präsentieren wir Echtzeit-Prototypen für ein Fahrzeugbeleuchtungssystem und ein Flugsicherungssystem.

Als Zweites verwenden wir SimB für die Simulation und Validierung von Systemen mit künstlicher Intelligenz (KI). In diesem Ansatz führt eine echte KI die Simulation über SimB aus, während ein formales Modell als Safety Shield zur Laufzeit, d.h. als Runtime-Monitor, dient. Das Safety Shield definiert, welche Aktionen sicher sind, sodass die KI nur sichere Aktionen ausführt. SimBs Validierungstechniken ermöglichen die Erkennung von Schwachstellen in der KI und im Safety Shield. Wir evaluieren diesen Ansatz an einer Autobahn-KI, die mit bestärkendem Lernen trainiert wurde, und zeigen, dass Responsibility-Sensitive Safety, ein bewährtes formales Modell für autonomes Fahren, die Sicherheit verbessert. Außerdem diskutieren wir weitere Anwendungen für KI-Systeme.

Der zweite Teil behandelt die Codegenerierung von high-level B-Modellen mit B2Program.

Als Erstes erweitern wir B2Program so, dass es Model-Checking Code generiert und damit eine leistungsstarke Performance erreicht. Um dieses Ziel zu erreichen, implementieren wir verschiedene Techniken, inklusive Parallelisierung und Caching. Wir implementieren ProBs Operation Reuse Technik für B2Program und zeigen, dass diese Technik auch für B2Program die Model-Checking Performance verbessert. Für die meisten Modelle erreicht B2Program eine bessere Performance als ProB und eine ähnliche Performance wie TLC. Wir betrachten die Zielsprachen Java, JavaScript/TypeScript, und C++.

Als Zweites verwenden wir B2Program, um HTML-Dokumente zu generieren, die Animation, Simulation, domänenspezifische Visualisierung in VisB, und Rückmeldung für die Validierung von B-Modellen unterstützen. Dieser Ansatz ermöglicht es, Domänenexperten in den Validierungsprozess des formalen Modells einzubinden. Als Fallstudien evaluieren wir ein Fahrzeugbeleuchtungssystem und ein Flugzeugfahrwerkssystem.

Acknowledgments

First, I would like to thank my parents, Thi Cam Phuong Vu and Van Minh Vu, for their support throughout my bachelor's, master's, and doctoral studies.

I have been part of the STUPS department at Heinrich-Heine-Universität since 2016, first as a research assistant and later as a doctoral student.

My sincere thanks go to Michael Leuschel and Jens Bendisposto, who gave me the opportunity to work in the STUPS department during my bachelor's and master's studies.

I would also like to thank Michael Leuschel for supervising my doctoral thesis and my master's thesis. He taught me a lot about formal methods, programming, and scientific work and inspired me to continue working in research after my master's thesis. I am grateful to him for giving me the opportunity to write my first paper and present it at iFM 2019 in Bergen before the beginning of my doctoral studies.

During my bachelor's studies, Jens Bendisposto initiated the ProB2-UI project and gave me the opportunity to work as a research assistant and software developer for ProB2-UI. I would also like to thank Jens Bendisposto for mentoring me during my bachelor's and master's studies. I gained a lot of experience in software engineering from him and had the opportunity to attend many meetups, including rheinJUG meetups.

I would also like to thank Angelo Gargantini, who was the second reviewer of my doctoral thesis, and Stefan Conrad, who was my mentor during my doctoral studies.

I would also like to thank Dominik Hansen for supervising my bachelor's thesis, which is the foundation for my first paper.

I would also like to thank Claudia Kiometzis for managing the administrative tasks in the STUPS group. Her support took a lot of bureaucratic work off my hands and made it possible for me to dedicate more time to research and teaching.

Also, I would like to thank all other members from the STUPS group I worked with: Ina Backes-Schulz, Carl-Friedrich Bolz-Tereick, Ivaylo Dobrikov, Jannik Dunkelau, Alexandros Efremidis, David Geleßus, Jan Gruteser, Yumiko Jansing, Philipp Körner, Sebastian Krings, Lukas Ladenberger, Mareike Mutz, Antonia Pütz, Max Ried, Jan Roßbach, Kristin Rutenkolk, Joshua Schmidt, David Schneider, Jonas Schneider, Sherin Schneider, Miles Vella, Michelle Werth, and John Witulski.

During my doctoral studies, I worked on the IVOIRE and KI-LOK projects. Working on the IVOIRE project, I would like to thank Atif Mashkoor and Sebastian Stock for constructive discussions and their feedback.

I would also like to thank all my co-authors for their work and all the fruitful discussions.

I would also like to thank the "Stiftung Familie Claus gGmbH" for supporting me with the "Deutschlandstipendium" during my bachelor's and master's studies.

Through my scientific work, I am also grateful for the opportunity to attend many conferences and present at most of them: iFM 2019 in Bergen, ABZ 2021 and FMICS

2021 online, iFM 2022 in Lugano, FMICS 2022 in Warsaw, ICFEM 2022 in Madrid, ABZ 2023 in Nancy, RSSRail 2023 in Berlin, NFM 2024 at NASA Ames in Silicon Valley, and ABZ 2024 in Bergamo. I also had the opportunity to help organize some workshops and conferences, namely, the IVOIRE workshops 2022 in Lugano, 2023 in Nancy, and 2024 in Bergamo, the Rodin workshop 2023 in Nancy, ABZ 2024 in Bergamo, and ABZ 2025 in Düsseldorf. I would also like to thank my reviewers and many people I met at conferences for their constructive feedback on my research.

Finally, I would like to thank my friends for their support.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Part I.

Introduction

1. Introduction

Software is becoming increasingly important in everyday life, including safety-critical domains such as automotive, aviation, and railway. With the rise of artificial intelligence (AI), safety-critical applications are increasingly using AI. Software errors in safety-critical, security-critical, and business-critical domains can be financially costly or result in loss of human lives. A prominent case with fatal consequences in recent years was the software errors of the Boeing 737 MAX in connection with the plane crashes of Lion Air Flight 610 and Ethiopian Airlines Flight 302, with over 300 deaths in total [105]. Misunderstandings in human-machine interaction might also lead to fatalities, such as with Air France Flight 447 in 2009, with over 200 deaths [60]. Errors in AI systems, e.g., autonomous driving systems, might lead to fatal accidents with human casualties as well [1].

Formal methods are used to specify, validate, and verify software systems, including those used for the previously mentioned domains. In particular, formal methods ensure the quality and safety of software systems.

This thesis explores simulation and code generation for validating and verifying formal models. We mainly focus on Classical B and Event-B as formalisms. In particular, this thesis presents and implements (1) new simulation techniques incorporating timing and probabilistic behavior to simulate formal models and (2) code generation techniques to verify formal models through model checking and involve domain experts early in the validation process. Evaluated case studies include interactive systems and AI systems.

1.1. Formal Methods

Formal methods are rigorous techniques for specifying and verifying software systems. In the context of formal methods, there are formal modeling languages with precise semantics. There are different types of formal modeling languages: State-based formal methods include B [4], Event-B [5], ASM [40], Z [214], and TLA+ [138]. Declarative formalisms include formal languages such as Alloy [117]. There are also process modeling languages based on process algebras, such as CCS [173], CSP [107], LOTOS [41], or the π-calculus [174]. Petri nets [191] and UML activity diagrams [68] are also suitable for process modeling. Other formalisms are timed automata [13] supported by tools such as UPPAAL [29] and Markov chains [50] supported by tools such as PRISM [131].

Since formal languages have precise semantics, formal models can be verified rigorously, i.e., in a formal way. Those verification techniques include model checking [18], proving [6, 113], or SMT/SAT solving [59, 15], all supported by various tools. One can use other logics to formulate temporal properties, such as Linear-Time Logic (LTL) [18] or Computational Tree Logic (CTL) [18]. As artificial intelligence (AI) is increasingly

becoming important in software systems, formal methods must also be able to validate and verify AI systems. Various approaches attempt to verify neural networks [121, 86, 201]. Furthermore, there are runtime monitoring approaches for AI systems with formal methods, such as robustness checks [86, 92], certified control [118], and safety shielding [12, 208, 193].

Formal methods are used for safety-critical applications such as aviation [177, 195, 242], aerospace [62, 42], automotive [158, 247], and railway [101, 71]. Moreover, formal methods also gain more relevance in business-critical and security-critical applications. Business-critical applications include Amazon's AWS [179] and Microsoft's Static Driver Verifier [19], Hypervisor [52], and Azure Cosmos DB [96]. Security-critical applications include credit cards, where formal methods are used to detect security issues [20, 21, 22]. Another security-critical application involves verifying security properties in the seL4 OS kernel [123]. Ter Beek et al. [222] provide a detailed overview of industrial applications of formal methods.

1.1.1. B and Event-B

Classical B [4] and Event-B [5] are state-based formal modeling languages that rely on *set theory* and *first-order logic*.

Within Classical B, each component is called a *machine*. The current state of a machine is represented by its *variables* and *constants*. A machine contains an *invariant*, a predicate that must be true in every state of the machine. This predicate is relevant for verification, e.g., proving and model checking. B supports the scalar data types booleans and integers, and compound types such as *tuples*, *sets*, and *records*. One can introduce new carrier sets as new data types in the `SETS` clause. A machine also contains an `INITIALISATION` clause to define the values of the variables in the initial state. These values might change by executing *operations*. An operation usually consists of a *guard*, which describes whether the operation is enabled, and *substitutions* [1] that change the values of the variables.

Event-B, the successor of Classical B, is used for system modeling, while Classical B is used for software modeling. A difference in Event-B is that the static parts, consisting of constants and sets, are stored in *contexts*, and the dynamic parts are stored in *machines*. Some keywords are also different. For example, Event-B contains *events* instead of *operations*, which work similarly. While Classical B supports *guards* and *preconditions* in the operations, Event-B only supports *guards* in the events. Unlike Classical B, Event-B does not support (imperative) constructs such as *sequential compositions*, *while-loops*, or *if-then-else* substitutions. Furthermore, there are many differences between Classical B and Event-B concerning refinement [149]. Leuschel [149] presents a detailed comparison of Classical B and Event-B.

Classical B and Event-B follow a refinement-based development approach, i.e., the components are refined step-by-step during the development process, with each step introducing more details. Each refinement step must be verified to be consistent with

[1] Similar to statements in imperative programming languages

the previous one. This incremental modeling and verification approach helps to detect errors in the design early. In practice, low-level code generators are applied to the final refinement to generate code for embedded systems. In Classical B, the final refinement used for code generation only contains constructs from a low-level subset of B, called B0, which is close to imperative programming languages. For instance, the AtelierB code generators [51] perform code generation from B0. Another code generator is B2Program [233], which I introduced in my bachelor's thesis [229]. B2Program generates code from a high-level specification and does not require refinement to B0. For more details on code generation, see Section 1.3.5.

The following chapters provide more background, including example codes for B and Event-B. More details on both formalisms are provided in [4] and [5], respectively.

There are also tools for Classical B and Event-B. ProB [153] is an animator, model checker, and constraint solver for both formalisms. Respective proving environments for Classical B and Event-B are AtelierB [51] and Rodin [8]. Code generation is supported for both formalisms, as outlined in Section 1.3.5.

1.2. Validation and Verification

In the following, we introduce and discuss the terminologies of *validation* and *verification*. Stock et al. [218] present a comparable discussion.

Validation [192] is a task to check whether a system meets the stakeholders' requirements, i.e., validation checks whether the correct system is implemented. In contrast, verification [192] checks whether the system is implemented correctly. In particular, verification checks for invariant violations, deadlocks, or WD errors.

Validation techniques include animation (e.g. as supported in ProB [153] or AsmetaA [34]), trace replay, and simulation (e.g. [167, 223]), which explore parts of the state space. Here, one could evaluate coverage criteria (e.g. operation coverage or MC/DC coverage), which are not as strict as full coverage. There are also less formal techniques, such as the inspection of visualizations, e.g., state space projections [136] or domain-specific visualizations (like BMotionStudio [134] or VisB [243]). Regarding validation, feedback from domain experts and stakeholders is crucial to confirm if the system meets the desired requirements.

More rigorous techniques are model checking [18] and proving [9], which cover all possible execution paths in the formal model. Both techniques are classified as verification. However, we also consider them for validation as they can check whether the stakeholders' requirements are met (similar argument in [218]).

In this thesis, we explore how *simulation* and *code generation* help to validate and verify formal models (with a stronger focus on validation).

1.3. Validation Techniques

In the following, we describe validation techniques relevant to this thesis.

1.3.1. Animation

Animation is a technique that enables humans to execute operations in a formal model interactively/manually. In particular, a human can animate a trace, step through the animated trace, and inspect the reached states in between to grasp the operations' effect step-by-step. Animators such as ProB [153] compute enabled transitions based on the semantics of the formal language. Those transitions are then suggested to the user to ease interaction with the formal model. ProB also allows one to step back and explore alternate execution paths. AsmetaA [34] is an animator for ASM specifications. In AsmetaA, a user can execute one *interactive animation* step interactively or provide a number of multiple steps that are executed randomly with *random simulation*.

In general, the main difference between *simulation* and *animation* is that a simulator automatically executes operations in *simulation*, while *animation* is controlled interactively by a human.

1.3.2. Model Checking

Explicit-state model checking (see Section 3–6 of [18]) is a technique that uses the formal language's semantics to explore the entire state space. While exploring the state space, the explicit-state model checking algorithm (Algorithm 4 in [18]) checks for state-based properties, such as invariants (Algorithm 4 in [18]) or deadlock-freedom [24]. Starting from the initial state, model checking computes and executes all enabled transitions to reach the successor states. The algorithm applies this step iteratively to the successor states until exploring the complete state space. There are also temporal model checking techniques, such as LTL model checking (see Section 5 of [18]) or CTL model checking (see Section 6 of [18]). Both temporal model checking techniques enable checking safety and liveness properties over traces.

There are search strategies for model checking, e.g., depth-first search, breadth-first search, mixed breadth/depth-first search, or heuristic-based techniques for more precise control.

More details on model checking, including algorithms for explicit-state and temporal model checking, are provided in [18].

1.3.3. Visualization

Visualization is crucial for gaining a better overview of a formal model and uncovering new insights. There are techniques for visualizing state spaces [152, 153, 156] and state space projections [136]. The latter is helpful for domain experts to validate behaviors in a formal model.

Another technique is to create domain-specific visualizations for formal models. There are/were several tools for ProB, supporting domain-specific visualizations, such as ProB's animation function [155], BMotionStudio [134], BMotionWeb [137], and VisB [243]. Other visualization tools are AnimB [2] and Brama [206] for (Event-)B, PVSio-Web [241] for PVS

[2] `https://wiki.event-b.org/index.php/AnimB`

models, and *formal* MVC [31] for ASM specifications. In particular, the domain-specific visualization graphically represents the current state of a formal model and provides interface elements for interaction with the formal model. Those interface elements are usually graphical elements that trigger an event in the formal model. Thus, the domain-specific visualization can be used as a prototype for the system, making it easier for domain experts to interact with the modeled system and validate desired behavior. Bombarda et al. [31] explicitly refer to this technique as the *formal model-view-controller pattern*, which aligns with the design of other visualization tools, e.g., VisB.

1.3.4. Simulation

Simulation techniques execute events in a system automatically, which is the main difference from animation, where events are performed manually by a human. In particular, simulation techniques include Monte Carlo simulation [176] and Co-simulation [223]. Monte Carlo simulation runs a random experiment multiple times and usually applies statistical validation techniques, such as *hypothesis testing* [122] or *estimation* [75]. Co-simulation is a simulation technique that allows many components to run in parallel. The subcomponents might also share data and use them for simulation. Well-known simulators in the context of formal methods are JeB [245], AsmetaS [83], and INTO-CPS [223].

1.3.5. Code Generation

In formal methods, code generation is a technique for generating executable code from a formal model. Formal models were initially designed for specifying and proving algorithms and systems, and not for execution [104]. Körner et al. [126] outline many benefits of executing formal models. For example, animating and simulating a formal model enable more precise validation of the system modeled.

In refinement-based formalisms, formal models are refined gradually until they reach an implementable subset of the modeling language. This subset is close to imperative programming languages and is often used to apply code generators. Classical B contains an implementable subset called B0 [51], which allows constructs such as *if-then-else* substitutions and *while-loops*. However, high-level constructs, such as set/relational operators and set comprehensions, are not allowed at that level. Code generators operating at the implementation level usually target embedded systems. At this level, the generated code only uses constructs with static memory allocation, eliminating the risk of running out of memory. Well-known low-level code generators for Classical B are the AtelierB code generators [51] and B2LLVM [38]. EventB2ALL [172] is a code generator that supports an implementable subset of Event-B.

However, many code generators are unverified. As mentioned in [233], the practice is to use at least two code generators developed by different teams to validate them against each other. The translations are also run on distinct hardware to protect against hardware errors (discussed in [233]). Such an approach is followed in the LCHIP project [143] in the CLEARSY Safety Platform [142].

In contrast, CompCert [146] is a formally verified compiler that ensures semantic equivalence between the generated executable code and the source program. That approach prevents the compiler from introducing additional bugs. CompCert [146] works on a small subset of C and targets embedded systems.

Some code generators apply to a high-level formal language. B2Program [233] is a high-level B code generator that targets Java [233], JavaScript/TypeScript [103], C++ [233], and Rust [66]. B2Program supports high-level B constructs including sets, relations, and set comprehensions. EventB2Java [49, 200] is another high-level code generator that generates Java code from Event-B models. Asmeta2Java [32] and Asm2C++ [36] are code generators for ASM models, targeting Java and C++, respectively. Both Asmeta2Java and Asm2C++ handle high-level ASM constructs, such as parallel execution of rules, non-determinism, and abstract domains. Asm2C++ supports generating C++ tests to validate the generated code [35, 36] and generating C++ code from domain-specific AValLa scenarios [36]. Moreover, Asm2C++ supports code generation for Arduino hardware [33].

1.4. Relevant Tools

So far, we have described validation techniques and tools that support them. This section provides a detailed overview of the tools relevant to this thesis, namely, ProB [153], ProB2-UI [25], VisB [243], and B2Program [233].

ProB, ProB2-UI, VisB. ProB [153] is an animator, constraint solver, and model checker for formal methods, including B, Event-B, TLA+ [99], CSP [43], Z, Alloy [129], and Lustre [230]. ProB supports explicit-state model checking, temporal model checking with LTL [194] and CTL, and symbolic model checking [128]. Furthermore, ProB supports various other validation techniques, including visualization and test case generation. ProB's visualization features include the visualization of state spaces [152, 153, 156] and state space projections [136], and domain-specific visualization with VisB [243].

VisB [243] is a tool within ProB for creating domain-specific visualizations. Using VisB, one can represent the current state of a formal model with graphical objects and define click listeners on them to trigger events. The appearances of the graphical objects change based on the current state. In a new feature of VisB, it is now possible to create graphical objects.

ProB2-UI [25] is a graphical user interface for ProB that supports all the ProB and VisB features mentioned before.

B2Program. B2Program [233] is a high-level code generator for formal B models. I introduced B2Program in my bachelor's thesis [229]. B2Program only targeted Java and C++ in [233]. Christopher Happe then extends B2Program to support Python in his project's work and TypeScript/JavaScript in his master's thesis [103]. Christopher Happe [103] also implemented an HTML export with domain-specific visualizations in VisB, which provides the basis for Chapter 8 in this thesis. Lucas Döring implemented

Rust for B2Program [66] by following an approach that generates low-level code from high-level B models. This thesis focuses on the target languages Java, JavaScript/TypeScript, and C++. The contributions of this thesis for B2Program are (1) extending B2Program by model checking code generation (mainly Chapter 7, Chapter 8 for JavaScript) and (2) demonstrating B2Program's HTML export for validation by domain experts and modelers (Chapter 8). Chapter 9 presents improvements to Chapter 7 and Chapter 8.

In contrast to low-level code generators (e.g. AtelierB code generators [51]), B2Program supports high-level constructs such as sets and relations, including high-level operations such as set operations (e.g., union, intersection, set difference, etc.), relation operations (e.g. domain, range, domain restriction, domain subtraction, etc.), and quantified expressions and predicates (e.g. set comprehensions, lambdas, existential/universal quantifiers, etc.). Thus, B2Program does not require refinement of a B model to B0.

The code generated by B2Program and the underlying external libraries allocate memory dynamically. Consequently, B2Program is not suitable for embedded systems. Furthermore, B2Program is not formally proven. However, this also applies to many other (even low-level) code generators. The primary motivation is to use B2Program for the development of prototypes, validation, and verification.

1.5. Reinforcement Learning

Reinforcement Learning (RL) [219] is a machine learning [248] technique where an agent learns to act in an environment based on *rewards/penalties*. While interacting in an environment, the agent performs an action a at each step, for which it receives a reward (penalty) r and observes the resulting state s. Figure 1.1 shows an overview of this process.

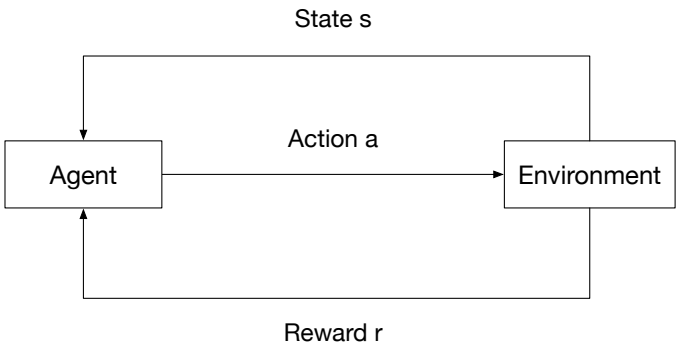


Figure 1.1.: Diagram of Reinforcement Learning, similar to presentation in [248]

The reward is computed based on a *reward function*. While training, the agent tries to learn a *policy* that maximizes the *long-term reward*. Reinforcement learning follows a *trial-and-error* approach, i.e., the agent performs various actions in the environment and uses the feedback as *rewards/penalties* to improve future decisions.

However, gaining insights into the agent's internal decision-making process remains challenging. While the reward function might consider safety constraints important, the agent could still execute actions that lead to dangerous situations. To tackle this problem, there are runtime monitoring and verification techniques which monitor RL agents and intervene in safety-critical situations. Below, we discuss runtime monitoring techniques with a focus on safety shielding.

1.6. Safety Shielding

Safety shielding [12] is a runtime monitoring technique. The original concept was introduced by Sha [208] and consists of two components: a complex unverified system and a simple verified controlling component. The simple system contains safety rules and enforces them on the complex system. When the complex system wants to execute an action which might lead to a dangerous situation, the controlling system intervenes to correct or avoid this action.

The *Neural Simplex Architecture* (NSA) [193] adapts this concept to RL systems. The RL agent represents a complex unverified system, while another simple and verified system monitors the RL agent. Additionally, there is a certified decision module that switches between both components. More generally, safety shielding employs a safety box around the AI. That means the AI itself might act unsafely, but the safety shield then corrects its decision.

Other runtime monitoring approaches for AI systems are robustness checks [86, 92] and certified control [118]. Robustness checking [92] verifies that a neural network's output remains stable under perturbations of the input. Certified control [118] aims to ensure the safety of an AI perception system. Certified control contains a main controller that generates a certificate to provide evidence for correct perception. An independent runtime monitor verifies this certificate to confirm the perception is correct.

1.7. Overview of Chapters

This thesis focuses on two topics, namely simulation and code generation. The first part focuses on simulation (see Part II), exploring how to design and apply simulation techniques to validate formal models. The second part focuses on code generation (see Part III), investigating the use of code generation for validation and verification. Figure 1.2 shows an overview of the chapters in this thesis.

The first part consists of the following chapters:

- **Validation of Formal Models by Timed Probabilistic Simulation** (Chapter 2): This chapter introduces a simulation technique called *timed probabilistic simulation*. We implement this technique in the simulator SimB in ProB2-UI [25]. The underlying concept is based on activations, which describe how events trigger one another with timing and probabilistic behavior. Timed probabilistic simulation can be run as Real-Time simulation or Monte Carlo simulation. SimB also enables statistical validation techniques, such as hypothesis testing and estimation of likelihood and values.

- **Validation of Formal Models by Interactive Simulation** (Chapter 3): In this chapter, we extend the SimB simulator in Chapter 2 by adding interactive elements, specifically listeners on user interactions performed via VisB or the ProB animator. By combining SimB with VisB, one can create real-time prototypes where system reactions are responsive to user interactions.

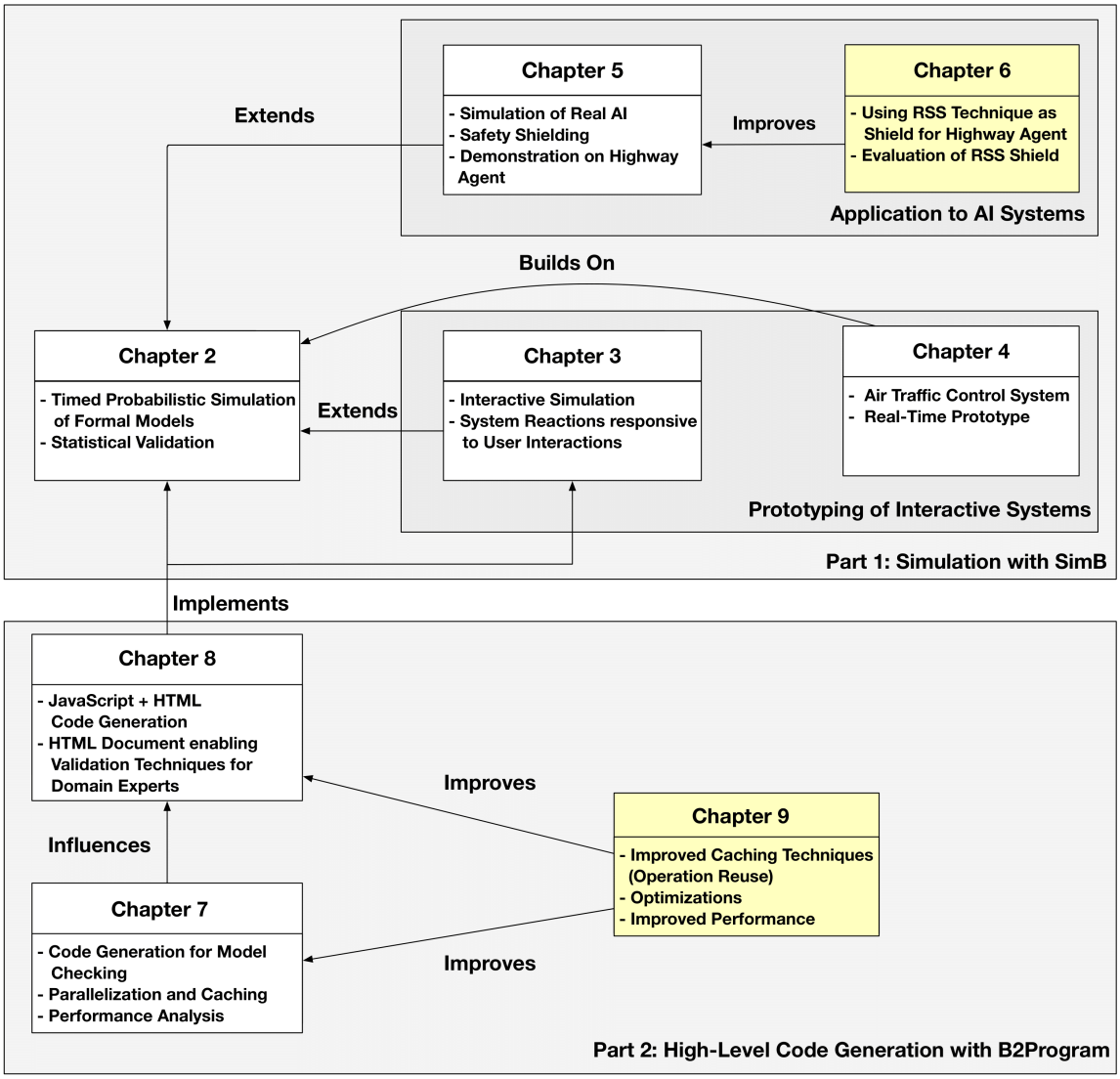


Figure 1.2.: Overview of Chapters in this Thesis: Simulation with SimB (Chapter 2 – Chapter 6) and High-Level Code Generation with B2Program (Chapter 7 – Chapter 9), Chapters with Additional Improvements and Evaluations in yellow (Chapter 6 and Chapter 9)

- **Development and Validation of a Formal Model and Prototype for an Air Traffic Control System** (Chapter 4): This chapter presents a formal model for an air traffic control system, namely Arrival Manager (AMAN). The system consists of a manual/interactive part controlled by an air traffic controller and an autonomous part that detects and schedules airplanes. Furthermore, Chapter 4 presents a real-time prototype for AMAN, developed from the formal model by utilizing VisB and SimB (presented in Chapter 2).

- **Validation of Reinforcement Learning Agents and Safety Shields with ProB** (Chapter 5): In this chapter, we present an approach to simulate a reinforcement learning (RL) agent in its environment with ProB and SimB. For the RL agent to run the simulation, we encode a formal B model corresponding to the RL agent and its environment. The formal model acts as a safety shield, ProB as a runtime verification/monitoring tool, and SimB as a simulation and statistical validation tool. We demonstrate the approach on a highway environment AI.

- **Additional Improvements and Evaluations** (Chapter 6): This chapter presents additional improvements and evaluations for Chapter 5. Here, we train new agents using slightly modified parameters from those in Chapter 5 to improve safety. Afterward, we evaluate using (the first rule of) Responsibility-Sensitive Safety (RSS) [209] as a safety shield for the highway AI. The formal B model implementing the RSS rules was created and developed by Michael Leuschel. To demonstrate the efficiency of the RSS shield, we also apply it to an adversarial agent.

The second part of the thesis consists of the following chapters:

- **Model Checking B Models via High-Level Code Generation** (Chapter 7): This chapter extends the high-level B code generator B2Program [233] by model checking capabilities. A key goal of code generation is to achieve high performance for model checking. To this end, we evaluate the model checking performance via B2Program on various machines, comparing it to the state-of-the-art code generators ProB [153] and TLC [246]. We implement parallelization and caching techniques to improve the performance.

- **Generating Interactive Documents for Domain-Specific Validation of Formal Models** (Chapter 8): In this chapter, we implement two approaches for generating interactive documents that enable domain experts to inspect formal models. The first approach (implemented by Michael Leuschel in ProB) is the *static export*, which generates an HTML document from a single trace. The second approach (implemented by Christopher Happe [103], extended and improved by Fabian Vu in this thesis) is the *dynamic export*, which uses B2Program to generate TypeScript/JavaScript and HTML from a formal B model and a VisB visualization. The dynamic export enables domain experts to be involved in the validation process, supporting techniques such as animation, trace replay, domain-specific feedback, domain-specific VisB visualization, and timed probabilistic simulation (presented in Chapter 2) with interactive simulation (presented in Chapter 3). While the static export works for all formalisms supported by ProB, the dynamic export is limited to the subset of Classical B supported by B2Program. The implementation of Chapter 7 also influences Chapter 8 because the animator in the dynamic export evaluates invariants and computes outgoing transitions.

- **Additional Improvements and Benchmarks** (Chapter 9): This chapter presents additional improvements and benchmarks on B2Program (improved implementation

and results for Chapter 7 and Chapter 8). In particular, Chapter 9 addresses many limitations discussed in Chapter 7 and Chapter 8. We also implement several optimizations, aiming to improve the performance of the code generated by B2Program. Those optimizations include rewriting certain predicates and expressions, and Leuschel's operation reuse technique [150]. Operation reuse significantly improves the performance of ProB for most models, as shown in [150] and Chapter 7. In this chapter, we evaluate whether Leuschel's operation reuse works efficiently for the code generated by B2Program.

1.8. Research Questions and Methodologies to Answer Them

In this section, we formulate the research questions of this thesis and outline the methodologies to answer them.

1.8.1. Validation of Formal Models by Timed Probabilistic Simulation

Animation enables humans to execute operations in a formal model interactively. Explicit-state model checking is fully automated and aims to explore the complete state space. Simulation is a technique to automatically run a formal model (see Section 1.3.4), i.e., without interaction, resulting in one or multiple traces. In contrast to animation, simulation executes each animation step automatically. In contrast to model checking, simulation only aims to cover parts of the state space.

One goal of this thesis (see Chapter 2) is to develop a technique which allows the formulation and execution of realistic simulations on formal models. We consider the following aspects to achieve this goal:

1. It should be possible to describe how events trigger one another.
2. When executing multiple events, there is usually a time delay between two successive events. Thus, one should be able to define the time elapsed until an event is executed in order to simulate realistic timing behavior.
3. The user should be able to formulate probabilistic behavior. Therefore, one should be able to probabilistically select between multiple events and values for variables and parameters.

This work presents a concept to annotate events with timing and probabilistic elements for simulation. We refer to the simulation technique as *timed probabilistic simulation* and implement it in a simulator called SimB. SimB is built on top of the ProB animator [153], i.e., it executes operations in the formal model via the ProB animator. Here, we ask the question:

- **Q1**: How can we annotate events in formal models with timing and probabilistic elements for simulation?

Finally, we also aim to validate timing and probabilistic properties. Here, we ask the research question:

- **Q2**: When is it beneficial to use timed probabilistic simulation, and how does this technique help modelers validate formal models?

1.8.2. Validation of Formal Models by Interactive Simulation

The results from Section 1.8.1 (see Chapter 2) enable us to simulate and validate formal models, taking into account timing and probabilistic behavior. While applying SimB to some case studies, we discovered use cases where an intermediate technique between animation and simulation might be beneficial and more realistic.

In particular, the systems in these use cases include both human interactions and automatic events. For instance, a pilot can control the landing gear of an airplane [135] by changing the handle's position in the cockpit. In response, one expects the landing gear system to perform a sequence of events automatically, resulting in the landing gear finally being extended or retracted. Another example is a vehicle's exterior light system [154], where a driver can change the position of the pitman controller or press the warning lights button. In response, one expects the corresponding vehicle's lights to flash every 500 ms.

Chapter 3 extends the simulator SimB by interactive elements to simulate system responses to user interactions. We refer to this feature as *interactive simulation*. Here, we ask the research question:

- **Q3**: How can we simulate system reactions in response to user interactions?
- **Q4**: When should we use *interactive simulation*, and how can we validate user interactions and system reactions?

1.8.3. Development and Validation of a Formal Model and Prototype for an Air Traffic Control System

Chapter 4 presents a formal Event-B model of an air traffic control system, namely Arrival Manager (AMAN). The requirements document for AMAN is provided by Palanque and Campos [186]. Within the AMAN system, an automatic component schedules airplanes arriving at an airport for landing, i.e., the airplanes are assigned a time slot on the timeline. These automatic events are called *AMAN updates* and occur every 10 seconds. The timeline is called the *landing sequence*, which contains a time slot for each minute that may be assigned to an airplane for landing. The AMAN system features a graphical user interface (GUI) that enables an Air Traffic Controller (ATCo) to interact with the timeline and the airplanes. For example, an ATCo can move airplanes or block time slots within the landing sequence, e.g., when the runway is occupied.

Thus, AMAN contains interactive parts performed by an ATCo via a GUI and autonomous parts which schedule airplanes.

Within ProB2-UI [25], there are tools for domain-specific visualization, namely VisB [243], and for timed probabilistic simulation with interactive components (presented in Chapter 2 and Chapter 3 of this thesis), namely SimB. Since AMAN consists of a GUI with interactive components and an autonomous simulation in the formal model, we evaluate whether the combination of VisB and SimB is suitable for creating a real-time prototype. Here, we ask the following research question:

- **Q5**: How can we convert a formal model into a prototype for a real-time human-machine interface?

1.8.4. Validation of Reinforcement Learning Agents and Safety Shields with ProB

In Chapter 5, we tackled the challenge of simulating and validating an AI system. The case study is a reinforcement learning (RL) agent in a highway environment [147].

A similar challenge emerged when using SimB for an AI-based railway system [93]. Gruteser et al. [93] presented a formal model with operations for object detection, which an AI with image recognition should perform. The approach there was to encode SimB activations with probabilities for object detection.

Another approach is to generate traces based on the AI's simulation runs outside formal method tools and validate them with SimB. Davin Holten [108] demonstrated the feasibility of this approach in his bachelor's thesis in the same highway environment [147]. Due to the simulation outside of the formal method context, the formal model did not influence the AI.

Here, we ask the research questions:

- **Q6**: How suitable are SimB's simulation and validation capabilities to check the safety and evaluate the quality of AI systems?
- **Q7**: How can we simulate real AI with SimB?

Once we can simulate real AI with SimB, we aim to use the formal model to ensure safety. A reinforcement learning-specific goal is to increase the reward as well. In particular, the approach followed in Chapter 5 is about encoding rules in the formal model to enforce them on the AI. Consequently, the formal model would act as a safety box surrounding the AI. Chapter 5 is related to Sha's approach [208] of using a simple verified controller to control a complex unverified system, and the Neural Simplex Architecture [193], which extends Sha's approach for AI systems. Furthermore, Chapter 5 is related to shield synthesis [125]. Here, we ask the research question:

- **Q8**: How can we use the formal model as a runtime monitor, i.e., a safety shield for AI systems?

Chapter 6 presents improved results and additional evaluations for Chapter 5. Furthermore, the results concerning **Q6 – Q8** also influence future work on runtime monitoring/verification and AI in general. Consequently, the results are also relevant for the AI-based railway system presented by Gruteser et al. [93]

1.8.5. Model Checking B Models via High-Level Code Generation

The initial results [233] of B2Program show that (1) code generation from many high-level constructs is feasible and (2) the code generated by B2Program is around two magnitudes faster than ProB for simulation for many models. Those results were particularly promising towards generating model checking code from high-level B models.

To the best of our knowledge, generating code for model checking is only done by the SPIN model checker [109] for the specification language Promela. Unlike Classical B, most Promela constructs are low-level and close to the C programming language.

Chapter 7 explores the generation of model checking code for high-level B models. Our implemented algorithm explores the state space explicitly. Invariants and deadlock-freedom are checked online for each visited state. One challenge of high-level B constructs is to deal with non-determinism. Here, we ask the research questions:

- **Q9**: How can we generate model checking code with B2Program from a Classical B model, targeting imperative programming languages?
- **Q10**: What high-level constructs does B2Program support, and what are the limitations?

High-level code generation for model checking is only beneficial if the performance can compete with state-of-the-art tools such as ProB [153] or TLC [246]. We benchmark ProB using the ProB CLI, and TLC by translating Classical B models to TLA with TLC4B [100], comparing the results with the code generated by B2Program.

ProB could be particularly strong in some high-level constructs due to constraint-solving capabilities. With B2Program, high-level code generation could improve the performance compared to interpreting formal models. We also evaluate techniques to improve performance, including parallelization and caching. Here we ask the research question:

- **Q11**: How does the generated model checking code by B2Program perform compared to state-of-the-art tools such as ProB or TLC, and which techniques improve the performance?

Chapter 9 presents additional improvements and benchmarks on B2Program, which also affects Chapter 7.

1.8.6. Generating Interactive Documents for Domain-Specific Validation of Formal Models

In Chapter 8, we implemented two approaches to generate interactive documents for domain experts to inspect formal models. The goal is to improve communication between modelers and domain experts.

The first approach (implemented by Michael Leuschel in ProB) implements a *static export* of a single trace featuring a domain-specific visualization. The *static export* shows the trace with the executed operations and the states reached (both formally and graphically). The values are hard-coded in the static export, meaning the domain expert can only inspect the exported trace.

In the second approach, the domain expert can dynamically interact with the formal model, i.e., animate and export their scenarios instead of inspecting a single scenario only. We refer to the second approach as the *dynamic export*. The dynamic export helps domain experts to validate a formal model using a domain-specific visualization. Christopher Happe [103] implemented B2Program's support to generate TypeScript/JavaScript code with an HTML document containing a domain-specific VisB visualization from a formal B model in his master's thesis [103], which provides the basis for Chapter 8. Here, we ask the following research question:

- **Q12**: How can we generate code for validation, and how is it beneficial for validating formal models?

The research questions in Section 1.8.5 are also relevant here. We evaluate the performance of "classical simulation" and model checking. As explained earlier, the model checking algorithm explores the complete state space by computing all transitions in each state. The animation feature in the dynamic export also computes all outgoing transitions for a visited state. Therefore, one animation step behaves similarly to one model checking step. Thus, we evaluate the model checking performance to represent animation. Here, we ask the research question:

- **Q13**: How does the code generated by B2Program perform for "classical simulation" and animation?

Chapter 9 presents additional improvements and benchmarks on B2Program, which also affects Chapter 8.

Part II.

Simulation

2. Validation of Formal Models by Timed Probabilistic Simulation

Abstract. The validation of a formal model consists of checking its conformance with actual requirements. In the context of (Event-) B, some temporal aspects can typically be validated by LTL or CTL model checking, while other properties can be validated via interactive animation or trace replay. In this paper, we present a new simulation-based validation technique for (Event-) B models called SimB. The proposed technique uses annotations to construct simulations, taking probabilistic and real-time aspects of the models into account. In this fashion, statistical properties of a single simulation run or a series of runs can be checked (e.g., Monte Carlo estimation or hypothesis tests). SimB complements animation and model checking, and its usability has been assessed via several case studies.

Funding. This research presented in this paper has been conducted within the IVOIRE project, which is funded by "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF) grant # I 4744-N. The work of Atif Mashkoor has been partly funded by the LIT Secure and Correct Systems Lab sponsored by the province of Upper Austria.

2.1. Introduction

A typical modeling approach in B [4] and Event-B [5] is to have a generic model for proof, and various instances of the generic model for animation or model checking. The generic model can be *verified* using provers, such as AtelierB[1] or Rodin [8], while the instances can be *verified* or *validated* with ProB [153] using animation and model checking. These two techniques are complementary: proof gives strong guarantees under the assumption of a correct model and can scale to large or infinite-state systems. But it provides limited feedback and typically cannot be used to ensure the presence of a desired real-world behavior. Animation and model checking provide more intuitive user feedback (e.g., in the form of domain-specific visualizations), but typically cannot be applied to generic models and usually cannot be used for exhaustive verification.

In this paper, we focus on validation, i.e., checking that a formal model is realistic and meets user expectations. Currently, in (Event-) B temporal properties (e.g., liveness) can be validated with LTL model checking, while the presence of features or desired behaviors

[1] `https://www.atelierb.eu/en/`

can be validated via animation and trace replay [164]. However, what is currently missing is the validation via more realistic simulations. In this work, we want to enable validation based on simulation taking into account real-time and probabilistic aspects. Our goal is to develop a lightweight and flexible validation approach, which can also be used for other formalisms (e.g., Z, TLA+, or CSP), and which is capable to accommodate various modelling styles and ways to encode time. Our approach builds on annotations of the respective formal models, processed by a simulator called SimB built on top of ProB.

As we show later, SimB can be used for a variety of complimentary validation tasks. Here we sketch one example. Suppose we have a generic Event-B model of a safety-critical component of train movement. This model has an abstraction of the environment, with just the features needed for proofs (e.g., maximal acceleration of other trains). The model may only incorporate a limited, abstract notion of time and may not include information about the likelihood of enabled events. This is where our new technique and tool get involved: we can associate time and probabilities with events of the model, enabling us to conduct realistic simulations as well as to collect statistical information about the formal model, e.g., the likelihood of enabled events or the likelihood of a behavior (within a certain time). Information about timing, probabilities, and interactions between events are not mined from true system executions. So, the challenge for the user is to define simulation parameters in SimB such that realistic simulations are created. Here it is possible to vary the simulation parameters for the same model to see how it behaves afterwards. This makes it possible to get a better picture of the model's behavior in the real world.

The rest of the paper is organized as follows. Section 2.2 describes how we encode timing and probabilistic behavior. Its realization in the form of SimB with the corresponding scheduling algorithm is explained in Section 2.3. Section 2.4 introduces a class of validation techniques using the presented simulation approach. Section 2.5 then demonstrates how SimB is applied to existing examples for validation purposes. In Section 2.6, we compare our approach with published works in the context of simulation and formal methods with probabilistic and timing behaviors. The paper is concluded in Section 2.7.

2.2. Timed Probabilistic Simulation Principles

This section explains the principles of encoding timing and probabilistic behavior in this work. To make the idea easier to understand, we recall the notion of operations and events in (Event-) B first, which will be referred to as events for the rest of the paper. Events consist of a guard and an action. An event can be fired if it is enabled, i.e., its guard is true. Firing an event executes the corresponding action modifying the current state. Note that events may use parameters and the action may itself be non-deterministic.

(Event-) B is a discrete-time language and events are always executed instantaneously. As shown in Figure 2.1, the model switches instantaneously from $c2{=}0$ to $c2{=}2$ without violating the invariant $c2 \in \{0, 2\}$ (i.e., not taking on the intermediate value 1). The (Event-) B method neither provides any guideline about how much time passes between

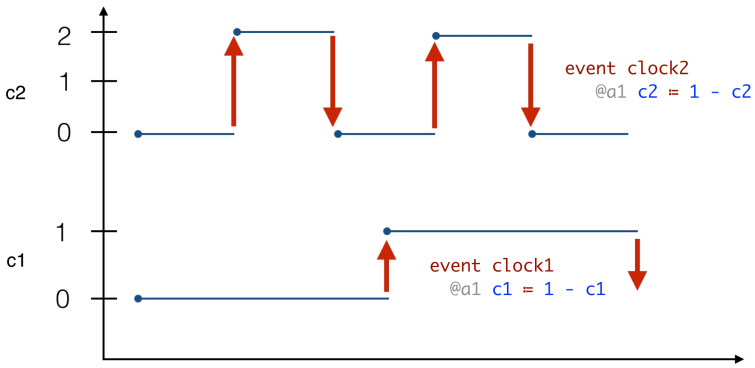


Figure 2.1.: Clocks Example

two event executions[2] nor imposes any restriction on how to choose which enabled event is executed.

```
invariants c1 : {0,1} & c2 : {0,2}
initialisation c1,c2 := 0,0
event clock1 = begin c1 := 1-c1 end
event clock2 = begin c2 := 2-c2 end
```

Listing 2.1. Examples with Two Independent Clocks

2.2.1. Encoding Simulation Time

Adding and Adapting Timing Behavior. Our approach works independently of whether time is already part of a model (e.g., in the style as suggested by Rehm et al. [198]) or not. The task of SimB is to add timing behavior in case time is not already part of the model. Otherwise, SimB annotations adapt to timing constructs that are part of the model. Furthermore, SimB simulates many processes in parallel.

Let us first have a look at the small example presented in Listing 2.1 and Figure 2.1. Here, time is not a part of the model. Suppose `clock1` and `clock2` are independent; one ticks every second, the other every 300 ms. Naively, one would increment the simulation time after executing an event. But following this naive approach, it is not possible to model the parallelism of both clocks. However, we can model it if we allow the execution of an event, which triggers other events. So in this example, `clock1` activates itself every second and `clock2` activates itself every 300 ms. This also makes it possible to encode sequential cyclic or acyclic processes like CSP [107].

SimB also intends to cater for models managing time explicitly, e.g., the models of automotive systems [154]. An important task here is to synchronize SimB annotations with the model's time, i.e., to adapt the timing behavior. This is enabled by the fact that SimB activation deadlines do not have to be static but can be computed from the model's constants and variables. Hence, event activation can take an explicit time or

[2]The fact that invariant proof obligations encode an induction proof, the B method implicitly assumes that there are no Zeno runs (i.e., there are no infinitely many events during any given finite time period).

deadline variable from the model into account. In conclusion, timing behavior is encoded such that events activate each other to be executed at a specific time in the future.

Event Activation. There are two issues regarding the activation of events with timing behavior:

1. Which event is selected first when many are activated at the same time?
2. How is a new event activation processed if the same event is already queued for execution in the future?

To tackle the first problem, the modeler can assign a priority when annotating an event to control which event is selected first. By default, their priorities are defined by the order they appear in the SimB annotations. The second problem is solved by adding another event activation to the queue by default, i.e., there are *multiple* activations for the same event. To make our encoding of timing behavior more flexible, it is possible to force SimB to keep just a single activation. Here, it must be specified whether the new activation should be ignored, or whether the maximum or minimum time of both activations should be taken.

2.2.2. Encoding Simulation Probabilities

There are three possibilities where probability can be applied to an (Event-) B model:

1. Probabilistic choice in non-deterministic assignments (e.g., x :: S)
2. Probabilistic choice between parameters
3. Probabilistic choice between events

As explained by Hallerstede et al. [97], a model becomes very difficult to understand when it features probabilities besides non-deterministic assignments. Consider different versions to model a coin toss as portrayed in Listing 2.2. The designer could model it either with non-determinism (1), with a parameter (2), or with two different events (3). In our approach, probabilities are not encoded in the model as we do not extend the B language. Instead, probabilities are encoded in SimB annotations, which will be explained in Section 2.3 more in detail.

```
toss = BEGIN lastToss :: {{Heads}, {Tails}} END // (1)

toss(c) = PRE c : CoinSide THEN lastToss := {c} END // (2)

toss_heads = BEGIN lastToss := {Heads} END;  // (3)
toss_tails = BEGIN lastToss := {Tails} END
```

Listing 2.2. Possibilities to Model a Coin Toss

To cover simulation for a wide range of models, it is thus necessary to enable the simulator to control all of the three encodings above. Otherwise, the existing models have to be rewritten to a given format to make the simulator feasible.

2.3. Simulation Infrastructure

Based on the aforementioned ideas, we now introduce our concept of activating events via annotations combining both timing and probabilistic behavior. An important issue is keeping the syntax and the semantics of the SimB annotations understandable. Even though both timing and probabilistic behavior are part of SimB, they should never be mixed up together at the same level. It becomes even more complicated when the modeler has to foresee that an event might be disabled.

After loading a formal model into ProB, corresponding annotations containing probabilistic and timing elements are loaded into the SimB simulator. Figure 2.2 shows the architecture of the interaction of SimB with ProB. SimB uses ProB to evaluate formulas and to execute events. Again, SimB manages a scheduling table to store the event activation's scheduled time. An event is executed if these two conditions are met: it is activated for now and it is enabled together with the chosen values for parameters and non-deterministic variables.

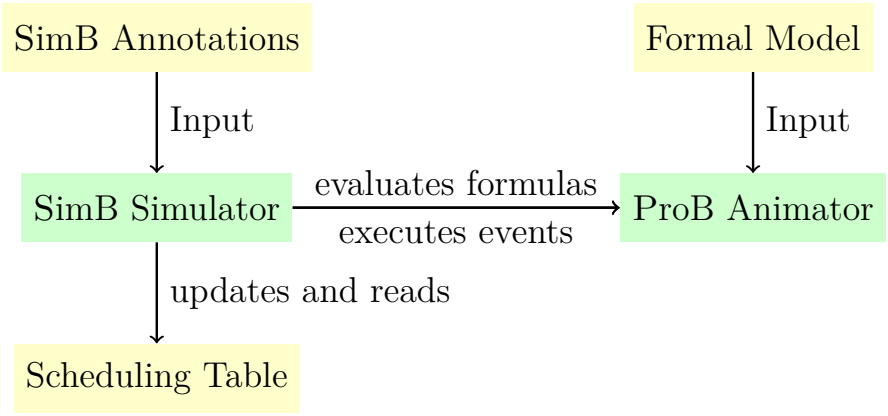


Figure 2.2.: Interaction of SimB Simulator with ProB using Annotations

Concept of Activation. Initially, SimB activates `SETUP_CONSTANTS` and `INITIALISATION`, whereupon the other events are activated. For each event executed by SimB, the modeler can annotate (multiple) events for activation in the future. There are activations of two kinds:

1. *direct activations* which execute an event after some delay,
2. *probabilistic choices*, which again lead to other activations, each labeled with a probability. The sum of the probabilities must be equal to 1. It is possible to chain multiple such activations together, but eventually, a direct activation must be reached.

Each activation is associated with an `id`. A *direct activation* always stores the activated event's name. Optionally, it also contains information to control the scheduled time, the

(probabilistic) choice between parameters and non-deterministic variables, the priority, additional guards, activation kind, and (multiple) activations to activate other events. In contrast, *probabilistic choice* activations contain the `id`s of the other activations with the corresponding probabilities.

Those simulation parameters are not mined from true system executions. So, it is the modeler's responsibility to define them such that realistic simulations are created. Using SimB, the user can vary the simulation parameters for the same model to see how it behaves afterwards. Regarding time and probabilities, it is not only possible to specify constant values, but also to use B formulas which are evaluated in the current state. Thus, it is also possible to vary the simulation parameters within a single simulation.

An example for SimB annotations for (3) of Listing 2.2 specifying a coin toss each 500 ms is shown in Listing 2.3. This results in the corresponding activation diagram graph portrayed in Figure 2.3 (*direct activations* in yellow, *probabilistic choice* in red).

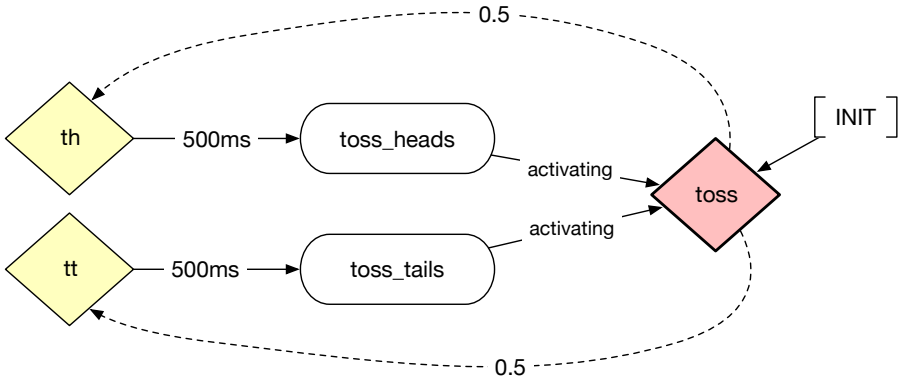


Figure 2.3.: Activation Diagram for Listing 2.3

When simulating coin tosses with the SimB annotations in Listing 2.3, SimB first activates the `INITIALISATION`. After initializing the model, SimB activates the *probabilistic choice* identified with `toss`. This again activates either the *direct activation* `tt` or `th`, each with a probability of 50%. Either `toss_heads` or `toss_tails` is then scheduled for execution in 500 ms. After executing one of the two events, the *probabilistic choice* `toss` is triggered again, which results in the next cycle simulating a coin toss.

```
{
 "activations": [
  {"id":"$initialise_machine", "execute":"$initialise_machine",
   "activating":"toss"},
  {"id":"toss", "chooseActivation":{"th": "0.5", "tt": "0.5"}},
  {"id":"th", "execute":"toss_heads", "after":500, "activating":"toss"},
  {"id":"tt", "execute":"toss_tails", "after":500, "activating":"toss"}
 ]
}
```

Listing 2.3. SimB Annotations for Coin Toss (3) in Listing 2.2

Scheduling Algorithm. The scheduling algorithm is separated into two parts: initialization and simulation loop.

Initially, `simTime` and `delay` are both assigned to 0. While `simTime` stores the simulation's current time, `delay` describes the time for the next scheduled events. The scheduling table `st` is initialized storing scheduled times for direct activations. They are identified by their `id`. Again, direct activations are stored in `annEvents`. In the beginning, the scheduling algorithm activates `INITIALISATION` and `SETUP_CONSTANTS` with `time(INITIALISATION) = 0` and `time(SETUP_CONSTANTS) = 0` respectively. To handle `SETUP_CONSTANTS` before the `INITIALISATION`, it is always assigned with a higher priority. For these two special cases, the user is not able to define the priority.

In the following, the simulation loop is described in Algorithm 2.1. The loop runs until reaching the ending condition, e.g., when the scheduling table is empty and thus no event can be fired anymore, or when the simulator is interrupted by the user. Within each simulation step, `simTime` is updated. Similarly, each scheduled activation's time is reduced by `delay`. Afterwards, activated events are processed by `executeActivatedEvents`. Finally, `delay` is updated to the time where the next events will be activated. Regarding real-time simulation, i.e., simulation with wall-clock time, this is the waiting time until the next step.

Algorithm 2.1: Algorithm for Simulation Loop

```
1  procedure simulationLoop()
2      while not endingConditionReached() // Finish at ending condition
3          simTime := simTime + delay // Update time
4          for each annEvent ∈ annEvents // Update scheduling table
5              for each activation ∈ st(id(annEvent))
6                  time(activation) := time(activation) - delay
7          executeActivatedEvents() // Execute activated events
8          delay := minimum(activationTimes(st)) // Update delay
9          wait delay // Wait delay (only in real-time simulation)
10 end procedure
```

Now, we refer to Algorithm 2.2 describing the execution of activated events. To execute scheduled events, the simulator iterates over the direct activations in order of their priority. When no priorities are specified the definition order in the file is used. An activation is scheduled for this step if it is activated now, i.e., its time is equal to 0. It is then removed from the scheduling table. When scheduling activations in the future, each activation is always inserted sorted after the time in the corresponding list. This makes it possible to iterate in each list until an activation is taken, which is not scheduled for this step. Afterwards, an enabled event matching the stored name, additional guards, and the (probabilistic) choice of parameters and non-deterministic variables is selected for execution if it exists. Executing an event activates other events following the *concept of activation*, which is realized by `activateEvents`.

Algorithm 2.2: Algorithm for Executing Activated Events

```
1  procedure executeActivatedEvents()
2      for each annEvent ∈ annEvents in order of priority
3          // Do not execute if ending condition reached
4          if endingConditionReached()
5              break
6          for each activation ∈ st(id(annEvent))
7              if time(activation) > 0 // Do not execute if not scheduled
8                  break
9              // Remove activation from scheduling table
10             st(id(annEvent)) := st(id(annEvent)) \ {activation}
11             // Select enabled event matching event name,
12             // additional guards, and (probabilistically) chosen
13             // values for parameters and non-deterministic variables
14             transition := selectTransition(activation)
15             if transition exists
16                 execute(transition) // Execute transition of activated event
17                 activateEvents(annEvent) // activate other events
18 end procedure
```

2.4. Applying SimB for Validation

Real-Time Simulation. Using SimB annotations and the underlying model, a modeler can play a single simulation in real-time, i.e., wall-clock time. This provides a feeling of how the model might behave in practice. The modeler can then manually check whether the model behaves as desired.

Monte Carlo Simulation. SimB also supports Monte Carlo simulations [176]. Here simulations can be performed faster than in real-time (i.e., SimB does not have to wait for the delay to expire, as long as it keeps track of the elapsed time in the model). In SimB, the modeler can specify a start predicate, a start time, or a number of steps that a single generated scenario must have reached. Furthermore, it is possible to define a number of steps, an end predicate, or an end time where the simulation for generating a single scenario should end. In addition to Monte Carlo simulation, the modeler can provide probabilistic and timing properties that are checked on the resulting simulations taking the start condition into account. Two validation techniques are considered here: hypothesis testing [122] and estimation [75]. During Monte Carlo simulation, the simulator also collects statistical information, e.g., the likelihood of enabled events or the likelihood of a behavior (within a certain time).

Given several simulations, a hypothesis, and a significance level, the modeler could ask a question whether to accept or reject the hypothesis. This is done by checking whether a certain property is fulfilled to a given probability. Given several simulations, one could

also ask a question about a certain value, which is then estimated. For example, let v_e be the estimated value, and v_d be the desired value, it is then possible to check whether $v_e \in [v_d - \epsilon, v_d + \epsilon]$ for a given ϵ.

Compared to probabilistic (temporal) model checking [144], SimB does not encode a Markov chain which is then used as a state space with probabilities. Furthermore, SimB does not validate probabilistic temporal properties expressed as PCTL [102] formulas. There are also statistical model checking techniques applying Monte Carlo simulation, hypothesis testing, and estimation. Scenarios are generated whereupon PB-LTL formulas [2], or BLTL formulas with a threshold [145] are checked. Since SimB does not check temporal formulas, it is not possible to validate properties over infinite paths. As mentioned before, one can define a start condition as well as an end condition between which a certain property is checked with a probability.

Timed Trace Replay. Given a single simulation run, one can also save it to a trace file with the particular timing encoded as SimB annotations. Afterwards, this timed trace can be re-played in real-time. It does not matter whether the simulation was generated from real-time simulation or Monte Carlo simulation. The resulting SimB annotations do not contain any probabilistic elements. Consider a simulation generating a trace with length n where the timestamp of the i-th event is $ts(i)$. It is then possible to generate SimB annotations where executing event i activates the event $i+1$ with annotation $i+1$ and with a scheduled time of $ts(i+1) - ts(i)$. Nevertheless, it is still challenging to check whether a timed trace can be re-generated from a modified model or SimB annotation. It might then be necessary to save more information about the simulator's history, e.g., which probabilistic choices have been taken into account or which activations have been discarded.

2.5. Case Studies

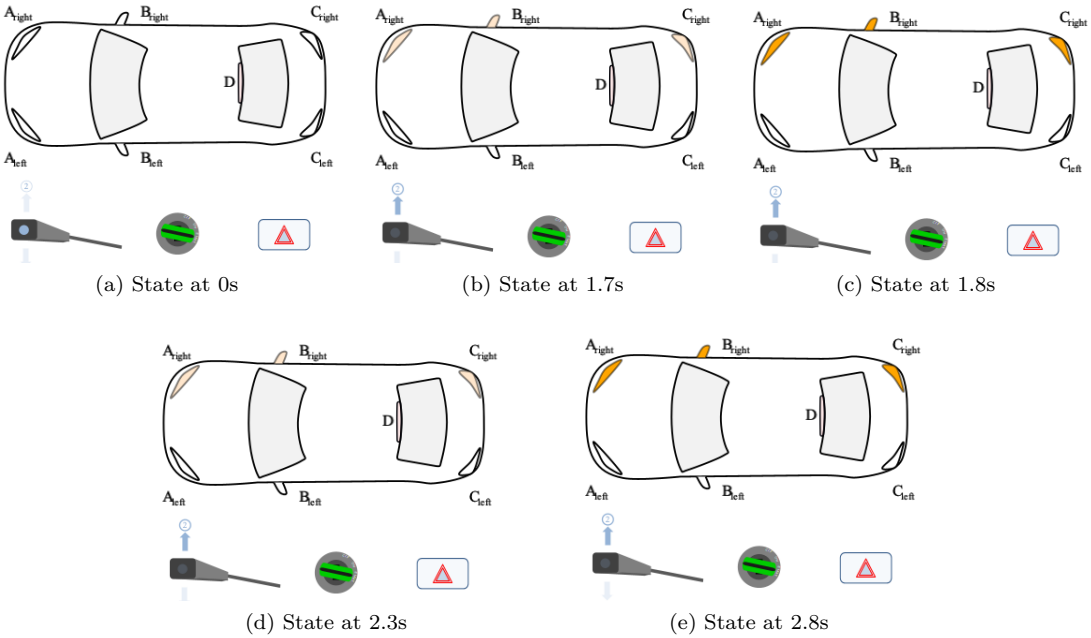
This section demonstrates the application of SimB to various case studies. See Table 2.1 for a complete list of applied case studies, which are accessible online[3].

Real-Time Simulation and Timed Trace Replay. In the context of real-time simulation, we only focus on the automotive case study [154]. As a case study, the driver's inputs on the pitman controller and the warning lights are simulated. Every time the driver activates the pitman controller or the warning lights, a sequence of events blinking the lights until the driver's next input is triggered.

A property to be validated is, e.g., that the corresponding lights are turned on with full intensity within a certain time after the driver makes an input on the pitman controller. Another property for validation is that the lights never turn on until the driver makes an input on the pitman controller or the warning lights button.

[3] Available at `https://github.com/favu100/SimB-examples`

Within the model, the time is modeled as a variable that is increased by events, which are responsible for passing the time, passing the time until the next deadline, blinking the lights and passing the time until the next deadline, and passing the time until the next deadline with a timeout. By following the principles of our simulation approach (see Section 2.2), it was possible to adapt to the model's timing specification.



(a) State at 0s (b) State at 1.7s (c) State at 1.8s

(d) State at 2.3s (e) State at 2.8s

Figure 2.4.: Simulation Example for the Automotive Case Study [154]

Figure 2.4 shows an actual simulation illustrated as VisB [243] visualizations. Using VisB alone, one can create SVG images and an associated VisB file for a model. Within the VisB file, it is possible to define which operation is triggered when clicking an SVG element. Here, one can also manipulate the style of the images by using B formulas which are evaluated in each state. In combination with SimB, a simulation can then be seen as an animated picture similar to a `GIF` picture. As shown in Figure 2.4a, the engine is turned on, the pitman controller is in a neutral position, the warning lights button is not pressed, and the lights are turned off. After 1.7 seconds have passed, the driver decides to move the pitman controller to `Upward7`, which activates the lights on the right-hand side (see Figure 2.4b). With a delay of 100 ms, the lights on the right-hand side turn on whereupon they blink every 500 ms (see Figure 2.4c - Figure 2.4e).

Timed traces are successfully captured from the real-time simulation as well as Monte Carlo simulation. As explained before, they are stored as SimB annotations. Thus, re-playing them works similar to real-time simulation.

Monte Carlo Simulation. SimB is also used for Monte Carlo simulation, together with hypothesis testing and estimation, to validate the timing and probabilistic behavior of formal models. The results are shown in Table 2.1.

Table 2.1.: Application of SimB Validation Techniques Based on Monte Carlo Simulation to Case Studies with Number of Runs, Number of Evaluated Transitions (ET), Runtime in Seconds (RT), and the Result of Validation

Model	Simulation	Property	Runs	ET	RT	Result
Coin Toss	Fair Tosses	Heads in 50% of all Tosses	1 000 000	7	8.19	✓(49.93%)
		Eventually Heads in 100 Tosses	10 000	7	3.43	✓(100%)
Rolling Dice	Fair Dices	6 in 16.67% of all Rolls	1 000 000	43	10.33	✓(16.66%)
		Eventually 6 in 100 Rolls	10 000	43	6.09	✓(100%)
Dueling Cowboys	100 Cowboys, 80% Accuracy	Termination in 125 Shots	100	1 720 854	1676.06	✗(56%)
		Termination in 250 Shots	100	1 723 302	1703.74	✓(100%)
Dueling Cowboys (abstract)	100 Cowboys, 80% Accuracy	Termination in 125 Shots	10 000	201	11.01	✗(63.13%)
		Termination in 250 Shots	10 000	201	12.51	✓(100%)
Tourists	100 Tourists	Termination in 125 Moves	100	956 468	2019.1	✗(0%)
		Termination in 300 Moves	100	1 081 099	3195.14	✓(100%)
Leader Election	10 Nodes	Termination in 250 Steps	10 000	37 917	201.6	✗(99.46%)
		Termination in 500 Steps	10 000	37 884	201.36	✓(100%)
Traffic Light (TL)	Cars TL from Red to Green	Red to Green in 0.5 s for Cars	1 000 000	5	9.61	✗(0%)
		Red to Green in 1 s for Cars	1 000 000	5	9.84	✓(100%)
Lift	Basement to 2nd floor	Reaching 2nd floor in 10 s	1 000 000	47	48.11	✗(0%)
		Reaching 2nd floor in 20 s	1 000 000	47	46.57	✓(100%)
Lift	Basement to 2nd floor with stop at 1st floor	Reaching 2nd floor in 20 s	1 000 000	70	78.36	✗(0%)
Automotive Case Study	Random Input on Pitman Controller and Hazard Warning Signal with Engine on	Left light blinks 100 ms with full intensity after moving pitman to Downward7	10 000	106	22.73	✗(99.17%)
		Left light blinks 500 ms with full intensity after moving pitman to Downward7	10 000	106	22.37	✓(100%)
		Lights never turn on until it is activated via pitman or warning lights	10 000	74	9.51	✓(100%)

Validated properties also include "almost-certain" properties, i.e., properties describing a random event to occur with probability 1. The examples Dueling Cowboys, Tourists (aka Rabin's Choice Coordination), and Leader Election (aka Herman's probabilistic self-stabilization) are taken from the work of Hallerstede et. al. [97] and Hoang [106]. All experiments are done with a fixed seed[4] in ProB2-UI[5] on a MacBook Air with 8 GB of RAM and a 1.6 GHz Intel i5 processor with two cores. A significance level and an ϵ-value are set for hypothesis testing and estimation respectively (as described in Section 2.4).

[4]Seed is a number used to initialize the random number generator to make the results are reproducible. We used 1000 as seed.

[5]`https://github.com/hhu-stups/prob2_ui`

Both values are set to 1% for 100 runs. Again, they are set to 0.1% for $\geq 10\,000$ runs.

Simulations with $\geq 10\,000$ runs were calculated within 4 minutes, with each of them executing more than 500 000 events. In contrast, those simulations with only 100 runs take up to 1 hour to terminate. Here, a significantly lower number of events ($\leq 65\,000$ events) are executed for each simulation. Currently, SimB evaluates all outgoing transitions before choosing one. This can obviously lead to performance issues. Particularly, a very large number of transitions are evaluated in the simulations with 100 runs. For the Dueling Cowboys we produced a more abstract version with a smaller state space, enabling us to simulate 10 000 runs in less than 13 seconds. In future, SimB could be improved such that it only evaluates a single transition given the probabilistic annotations.

2.6. Related Work

In this section, we compare our work with the state of the art in the field of modeling and simulation of probabilistic and timing behaviors.

Modeling and Verification of Probabilistic Behavior. Hallerstede et al. [97] introduce probabilistic events for Event-B in which the modeler can use probabilistic assignments in place of non-deterministic assignments (but not probabilistic choice between events nor for parameters, and there are no explicit values for the probabilities used). Based on this work, Hoang [106] presents an approach to verify almost-certain properties. In contrast, SimB simulates existing models by using additional annotations. This makes it possible to gain better insights on how the model would behave in a real-world application. SimB can use statistical techniques to validate the presence of a desired behavior with detailed feedback. We achieve this by building on top of the semantics of (Event-)B, not by changing it at the core. Our approach is more empirical than formal and proof-based.

Legay et al. [144, 145] provide an overview of statistical model checking including probabilistic model checking and numerical approaches. While the former is applied to a Markov chain (used as a state space with probabilities), the latter approximates certain values during validation. We do not encode a Markov chain in SimB and our work does not apply probabilistic model checking. Furthermore, SimB does not validate probabilistic temporal properties expressed as PCTL [102] formulas. There are also statistical model checking techniques applying Monte Carlo simulation, hypothesis testing, and estimation. This is done by generating simulations and checking timing properties expressed as BLTL formulas with a threshold. Abdellatif et. al. [2] present a simulation-based approach to generate attacking scenarios to validate probabilistic properties expressed as PB-LTL formulas in a model of smart contracts and the blockchain. Similar to our work, safety properties are also checked with fault tolerance and estimation of error probability. In contrast, we provide a property that is checked for each generated simulation, e.g., whether a predicate is eventually true between the starting condition and the ending condition of a simulation. Since SimB does not check temporal formulas, it is not possible to validate probabilistic properties over infinite paths.

Modeling and Verification of Timing Behavior. To model and verify real-time behavior, the modeler could use formalisms, such as timed automaton [13], with existing model checkers, e.g., Uppaal [29]. There are also approaches to verify both probabilistic and real-time behavior, e.g., by using the model checker PRISM which is applied to probabilistic timed automaton [132]. To check such properties in (probabilistic) timed automata, *reachability analysis* is applied. Its task is to check whether a state is reachable with the given timing (and probabilistic) properties. Our proposed approach is a lightweight solution to simulate existing models to gain additional insights into how they might behave in practice. SimB simulates a model until a certain condition, a certain time, or a certain number of steps is reached. Probabilistic and timing properties are then validated with statistical methods on the resulting traces without applying *reachability analysis*. Thus, SimB is not meant to replace other approaches based on timed automata. Abdellatif et al. [3] present a scheduling approach to check whether the modeled program is implementable, holding the defined timing properties. Again, one can also model concrete time in discrete-time formalisms, e.g., by following an approach presented by Leslie Lamport for TLA+ [139], or Event-B by following a timing constraint pattern discussed by Rehm et al. [198] and Mashkoor et al. [164]. Timing properties can then be verified with existing provers and model checkers in the corresponding languages. As aforementioned, our work simulates the underlying (Event-) B model by using annotations for timing and probabilistic behavior. SimB annotations can be used to match the modeled time, see, e.g., the automotive case study [154]. Furthermore, SimB can also simulate user behavior.

Simulators. JeB [167] is a framework, which translates Event-B models into JavaScript programs for simulation. Models are sometimes too abstract for animation tools such as ProB. To enable validation of these models anyway without refining, they are translated into executable programs. One can also insert pieces of code to control the simulation. The challenge is to define the *fidelity* property between the model and its translation [165]. In contrast, the task of SimB is not to make models executable, but to simulate executable models to apply statistical validation techniques. Therefore, SimB is built on top of the ProB animator.

Similar to our approach, Dieumegard et al. [61] present a simulator for an anti-collision function of a small robotic rover based on Event-B to understand how the specification behaves. Note that the simulator presented by Dieumegard et al. is not a generic one. So, it is limited to the robotic rover case study.

Co-Simulation. In VDM, simulation is already more common than animation or model checking. It has now been extended by a co-simulation toolset named INTO-CPS [223]. INTO-CPS tools also contain design space exploration implemented with search algorithms where simulation parameters may vary and scenarios outcomes evaluated. Thus, INTO-CPS can search for optimal simulation parameters. In SimB, it is still a challenge for the modeler to choose simulation parameters such that realistic scenarios are generated. The modeler could, e.g., vary the simulation parameters for the same

model to see how it behaves afterwards. But this process has to be done manually. Compared to this co-simulation tool-set, our approach is somewhat limited but much more lightweight. For example, there is neither a continuous simulation tool running nor an FMI interface in SimB. In the future, SimB annotations could actually be used on top of a co-simulation using (Event-) B.

Other Formalisms. It would have been possible to use CSP control annotations for (Event-) B models [114], as available in ProB [43]. But, CSP does not cater to probabilities or time. The Timed CSP interpreter from [67] is not available in the current release of ProB, and also lacks probabilistic features. There are also formalisms combining probabilistic and timing behavior such as Probabilistic Time Petri Nets [70]. Since this formalism is not supported in ProB, it would be necessary to implement an interpreter to control the model. SimB is designed to be as simple as possible, but strong enough for simulation of models with probabilistic and timing behavior. Moreover, on a technical side, our annotations also work for other formalisms (such as TLA+).

2.7. Conclusion and Future Work

In this paper, we presented SimB – a simulator for formal models, which adapts the concept of activations annotating events with timing and probabilistic elements. Here, it was particularly important to separate probabilistic and timing behavior from each other to keep the syntax and semantics of SimB understandable. By building SimB on top of ProB, it was possible to support formalisms that are supported by ProB such as B, Event-B, Z, TLA+, and CSP. SimB is capable of simulating environment inputs, e.g., by users, and models' behaviors.

In this work, the usability of SimB was demonstrated in several examples. SimB can either be used to extend existing models by timing and probabilistic behavior, or adapt to models where time is modeled as a variable. SimB is capable to validate formal models using Monte Carlo simulation, hypothesis testing, estimation, and timed trace replay. Using Monte Carlo simulation, the modeler can generate scenarios to gain insights into how the model might behave in real-world. It is then possible to replay them with timing behavior, or to validate timing and probabilistic properties with hypothesis testing and estimation.

More information on SimB with screenshots and a tutorial is available at:

`https://prob.hhu.de/w/index.php?title=SimB`

As future work, it would be possible to add more statistical validation methods. Furthermore, the performance of SimB could be improved. On the one hand, SimB should compute a single transition given the defined probabilistic annotations, rather than computing all and choosing afterwards. On the other hand, one could apply code generation for SimB. To cover a wide range of models, it would be necessary to target generated code from other high-level code generators such as B2Program [233], EventB2Java [49] or Asm2C++ [36]. Additionally, we would also like to describe

the semantics of SimB in formal logic. One could then implement an interpreter and integrate it into ProB's core. This could be a way to reduce the overhead of SimB to ProB. Furthermore, it is then possible to animate and model check (Event-) B models together with SimB annotations.

Regarding the future, one could investigate how SimB can be used for co-simulation. Furthermore, it is still a challenge for the modeler to choose simulation parameters such that realistic scenarios are generated. So, another future work would be to analyze how optimal simulation parameters could be explored.

Eventually, we intend to use SimB in the context of *validation obligations* [166], which is the idea of breaking down the validation of a formal model into smaller tasks and associating them with each refinement step. Validations should then be applicable and re-usable for the whole software development life cycle.

3. Validation of Formal Models by Interactive Simulation

Abstract. Validating requirements for safety-critical systems with user interactions often involves techniques like animation, trace replay, and LTL model checking. However, animation and trace replay can be challenging since user and system events are not distinguished, and formulating LTL properties requires expertise. This work introduces interactive simulation, a new technique that combines domain-specific visualization of formal models with timed probabilistic simulation to create more realistic prototypes. It allows domain experts and users to interact with formal models and simulate the system/environment reactions. State diagrams are also generated for inspecting user interactions and system reactions. Finally, we demonstrate *interactive simulation* on the ABZ automotive case study.

Keywords. Validation, Formal Methods, Visualization, Simulation, Interactive

Funding. The research presented in this paper has been conducted within the IVOIRE project, funded by "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF) grant # I 4744-N.

3.1. Introduction and Motivation

Many safety-critical systems require human interaction to trigger a response from the system or environment. For instance, a lift moves on button clicks, car lighting is controlled by a driver [110], the airplane landing gear is operated by a pilot [39], and air traffic controllers schedule airplanes via computers [185].

Safety-critical systems are often modeled using formal methods which make use of mathematical notation. For example, models in B [4] and Event-B [8] rely on set theory and first-order logic. This makes it hard for users and domain experts to understand and interact with the model. These interactions cannot always be fully formalized or verified; hence validation is important to ensure that a formal model meets desired user requirements [116].

Approaches for domain-specific views for formal models include VisB [243] for interactive visualizations, and SimB [237] for simulating real-world behavior with probabilistic and timing properties. Both visualization and simulation are fundamental constructs for *validation obligations* (VOs) [166], an approach to validate requirements in formal models systematically. VOs also take domain experts' and users' feedback into account. Before

this work, SimB was not responsive to user interaction in VisB, making it impossible to trigger system reactions with timing behavior based on user interaction.

This paper introduces a new *interactive simulation* technique, integrated into SimB in ProB2-UI. *Interactive simulation* allows users to execute events via VisB, triggering automatic system reactions via SimB simulation. State diagrams focusing on graphical components in VisB are also presented to provide a domain-specific view of user interactions with the system. The features improve user experience, specifically in formal models with human-machine interactions, providing better access to the validation process for users and domain experts.

3.2. Interactive Simulation

Interactive simulation combines animation, simulation, and visualization. First, we present the principles of VisB and SimB, and then the implementation of *interactive simulation*.

Principles. VisB is a visualization tool in ProB2-UI [25] which uses the animator, model checker and constraint solver ProB [152]. A VisB visualization consists of an SVG file, and a glue file that links SVG objects with the formal model. The glue file includes observers for SVG objects (VisB items) that change the objects' attributes (like colour) based on the model's current state, and click listeners on SVG objects (VisB events) that execute events in the formal model.

```
{"id":"peds_red", "attr":"fill",
"value":"IF tl_peds = red THEN \"red\" ELSE \"black\" END"},
{"id":"peds_green", "attr":"fill",
"value":"IF tl_peds = green THEN \"green\" ELSE \"black\" END"}
```

Listing 3.1. Example of VisB Items

```
{"id": "PitmanUpward",
 "event": "ENV_Pitman_DirectionBlinking", "predicates": ["newPos=Upward7"]}
```

Listing 3.2. Example of VisB Event

Listing 3.1 shows VisB items for the pedestrians' traffic light's appearance based on the variable `tl_peds` (e.g. `fill` attribute of `peds_red` is `"red"` when `tl_peds` is equal to `red`, otherwise `"black"`). Listing 3.2 shows an example of a VisB event from an automotive case study (see Section 3.4). The VisB event states that a click on the SVG object with `PitmanUpward` as *id* executes the event `ENV_Pitman_DirectionBlinking` with `newPos=Upward7` in the formal model.

Figure 3.1 shows a complete visualization of the automotive case study with the car lighting system, the pitman controller (to turn on the direction indicators), the key ignition (to turn on the engine), and the warning lights button.

However, VisB has some limitations, e.g., VisB does not enable the activation of a sequence of events or control the time elapsed between events, nor allow probabilistic event selection. These features are provided by another component.

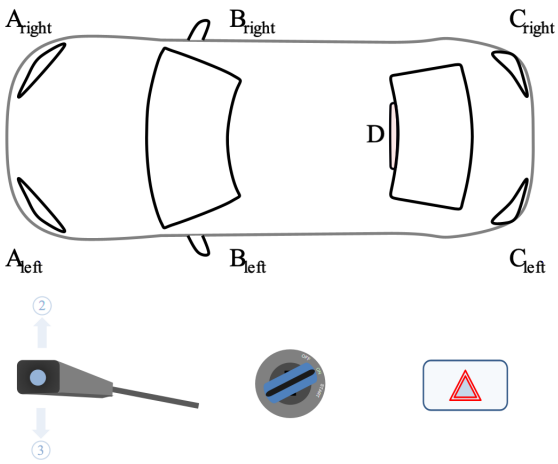


Figure 3.1.: VisB Visualization for Automotive Case Study [154]

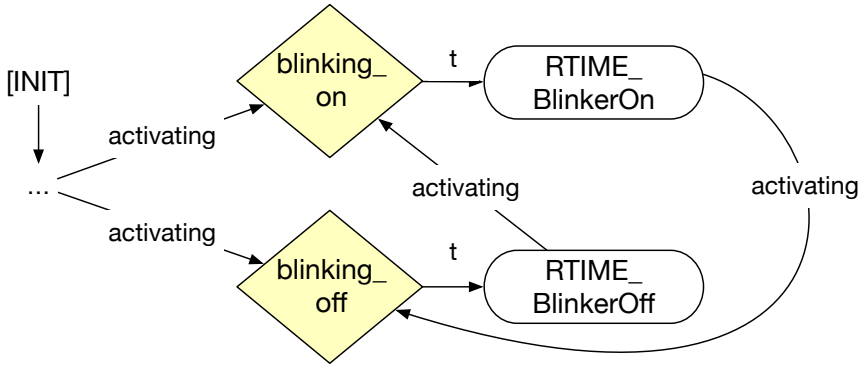


Figure 3.2.: Example of SimB Diagram

SimB is a tool in ProB2-UI which uses ProB's animator to simulate realistic scenarios. A modeler can use SimB to encode simulations with activation diagrams (see Figure 3.2) annotating events in formal models with time and probabilities. Simulations start automatically at the model's initialization, triggering other events. Ideally, simulations run deadlock-free, i.e., events continue triggering each other. The core concept is *activations* of two kinds [237]: (1) *Direct activations* which execute events after a specific time, and optionally trigger other activations, and (2) *probabilistic choices* which choose between activations probabilistically (eventually a *direct activation* must be reached). SimB manages a scheduling table to represent the simulation's current state as a multiset of scheduled *direct activations*, along with the scheduled time, i.e., the time until the corresponding event is executed. For illustration, we only show *direct activations* (yellow diamonds in Figure 3.2) in this paper.

While a simulation is running, the user can still intervene and execute events in ProB2-UI. However, SimB was not responsive to user interaction as there was no link between user interaction and SimB's activation diagram. Thus, it was not possible to apply a user interaction to trigger a chain of system events. To address this issue, we

developed an *interactive simulation* technique.

Figure 3.2 shows parts of a SimB activation diagram for [154] where both activations (yellow diamonds `blinking_on` and `blinking_off`; JSON representation in Figure 3.3) trigger each other in a cycle. Each activation executes events from the model (`RTIME_Blinker_On` and `RTIME_Blinker_Off` after a delay of `t`). The complete activation diagram controls both user behavior and the vehicle's reaction automatically, with no distinction between user and system events or activations.

```
{"id": "blinking_on",
 "execute": "RTIME_BlinkerOn",
 "after":
    "curDeadlines(blink_deadline)",
 "activating" : "blinking_off", ...}
```

```
{"id": "blinking_off",
 "execute": "RTIME_BlinkerOff",
 "after":
    "curDeadlines(blink_deadline)",
 "activating" : "blinking_on", ...}
```

Figure 3.3.: Example of SimB Activations in Figure 3.2

Architecture. Figure 3.4 shows the architecture of ProB2-UI and ProB together with VisB and SimB. When loading a VisB visualization or a SimB simulation, they are first checked syntactically and semantically wrt. the model. A user can then interact with the formal model via ProB's animator, or the VisB visualization. With *interaction simulation*, users can execute an event that automatically triggers a sequence of other events with time elapsing in between. This is realized by (newly introduced) SimB listeners that recognize user interactions and trigger SimB activations accordingly. A user can then observe the system's reaction.

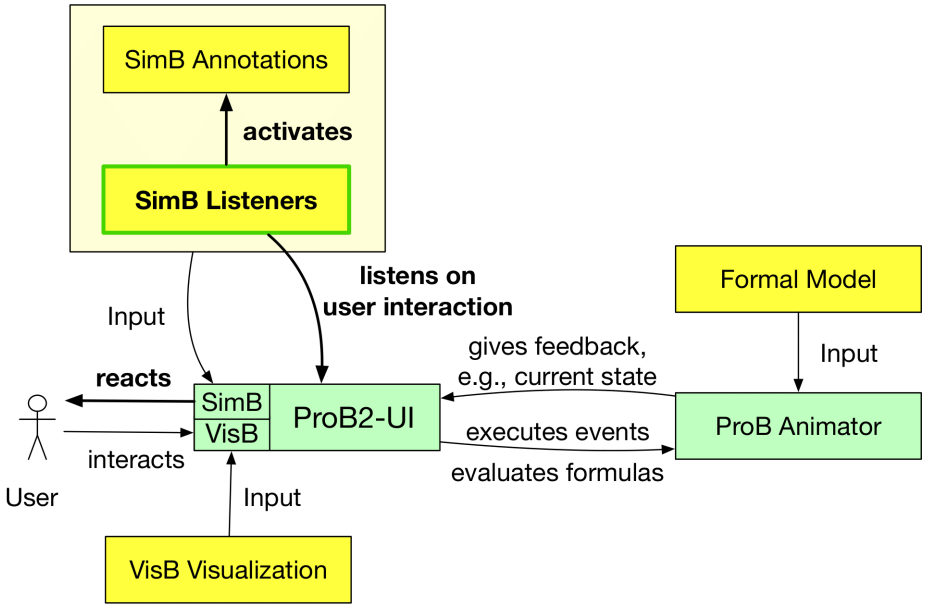


Figure 3.4.: Architecture with ProB2-UI, ProB, VisB, SimB, and User Interaction (new features marked in **bold**)

Implementation. In the implementation, we distinguish events of two types: those triggered by SimB, and those triggered via user interaction. Events triggered by SimB are already part of the activation diagram.

SimB listeners are defined on events that are manually triggered, fulfilling a predicate (realized with `event` and `predicate` in JSON). Based on the user interaction, a SimB listener triggers simulations associated with the `activating` field which stores activations. Thus, SimB listeners define additional entry points into the activation diagram which are triggered by user interaction. Listing 3.3 shows a SimB listener which detects user interactions on `ENV_Pitman_DirectionBlinking`, and triggers the activations `blinking_on` and `blinking_off` (see Figure 3.3).

This results in the activation diagram in Figure 3.5. Unlike Figure 3.2, user interaction is integrated into SimB as an entry point for the simulation. The blinking lights are triggered by user interaction, and not as part of a fully automatic simulation activated at the model's initialization.

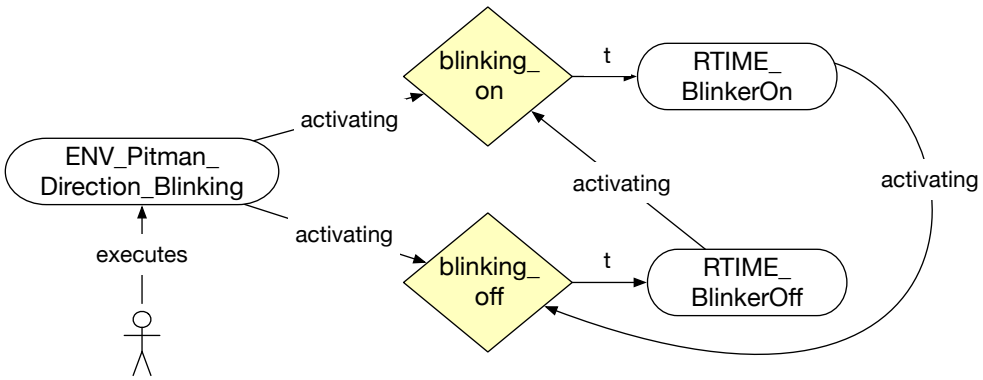


Figure 3.5.: Activation Diagram with SimB Listener

```
{"id": "start_blinking",
 "event": "ENV_Pitman_DirectionBlinking", "predicate": "1=1",
 "activating" : ["blinking_on", "blinking_off"]}
```

Listing 3.3. Example for SimB Listener

3.3. VisB Diagrams

ProB has a feature that projects the state space onto an expression [136]. Such an expression could be a tuple of variables of interest. These diagrams are useful to study the model's behavior for a particular aspect or feature. This work extends that feature by combining it with VisB. This results in VisB diagrams (e.g., Figure 3.6) that can be read by domain experts, without having to understand the textual representation of B values.

VisB diagrams combined with *interactive simulation* help to see how user events and system/environment interact with each other from the user's perspective in VisB. A

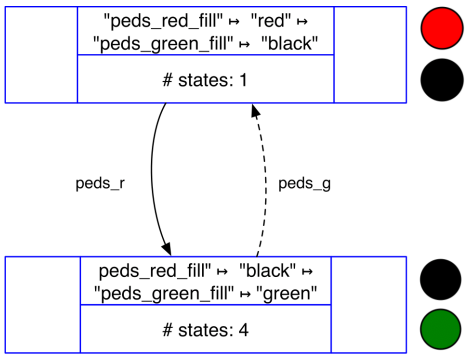


Figure 3.6.: VisB Diagram from Listing 3.1

detailed case study is presented in Section 3.4 (notably Figure 3.8). VisB diagrams focus on a subset of graphical objects and attributes. We use ProB to compute the state space projection for relevant expressions used by VisB to compute the attributes. We also use VisB to render each projected state graphically. Figure 3.6 shows two projected states (out of five in the complete state space), along with their graphical renderings[1].

Let us describe this feature more formally. Let V_{items} be the set of VisB items and let V_{prj} with $V_{prj} \subseteq V_{items}$ be the subset of VisB of interest. A VisB item $v \in V_{items}$ contains attributes for the SVG object's id, attribute and value, i.e., $v = (v.id, v.attr, v.value)$. A VisB diagram is created with a projection [136] on:

$$v_1.id \mapsto v_1.attr \mapsto v_1.value \mapsto \ldots \mapsto v_n.id \mapsto v_n.attr \mapsto v_n.value$$

where $V_{prj} = \{v_1, \ldots, v_n\}$ and $\forall i, j \in 1..n \wedge i \neq j \implies v_i \neq v_j$.
An example is given for Listing 3.1, resulting in the left-hand side of Figure 3.6:

```
"peds_red"↦"fill" ↦ "IF tl_peds = red THEN \"red\" ELSE \"black\" END" ↦
"peds_green"↦"fill" ↦ "IF tl_peds = green THEN \"green\" ELSE \"black\" END"
```

3.4. Case Study

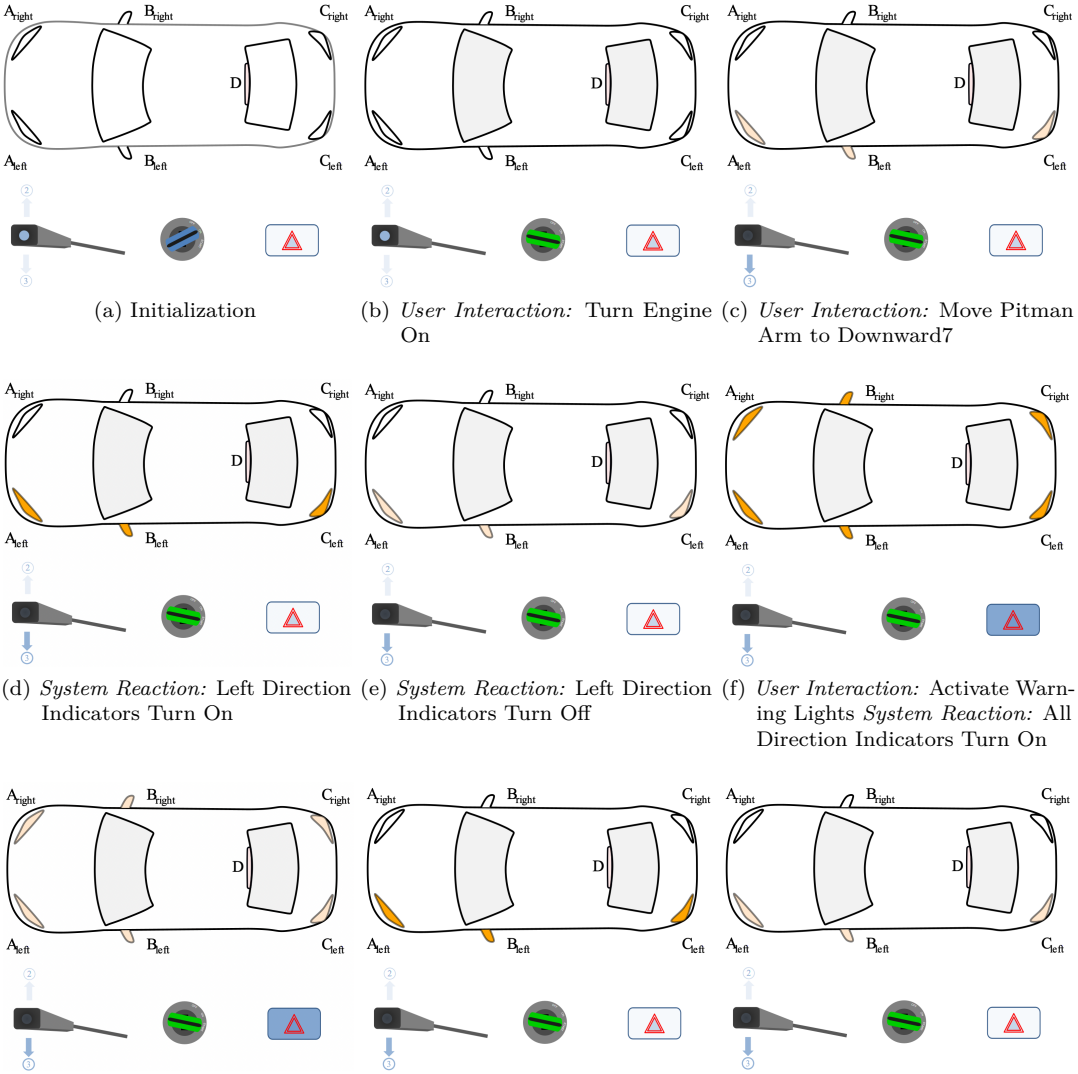
This section demonstrates the features introduced in Section 3.2 and Section 3.3 on an automotive case study [154]. A VisB visualization is shown in Figure 3.1. Now, we focus on specific requirements that have been modeled and validated by Leuschel et al. [154] and Vu et al. [237], with a special interest in the interactive/human (*italic*) and automatic/autonomous (underlined) parts, and their connection:

- **ELS-1** *Direction blinking left*: *Assuming that the ignition key is inserted: When moving the pitman arm in position "turn left"*, the vehicle flashes all left direction indicators (...) synchronously [...] and a frequency of 1.0 Hz +- 0.1 Hz (i.e. 60 flashes per minute +- 6 flashes).

[1]The technique is not yet fully automated: VisB visualisations were added manually to the right-hand side of Figure 3.6. Note that our feature was inspired by *transition diagrams* in BMotionWeb [137, 133].

- **ELS-8**: As long as *the hazard warning light switch is pressed (active)*, all direction indicators flash synchronously. [...]

- **ELS-12**: When *hazard warning is deactivated again*, the *pitman arm is in position "direction blinking left" or "direction blinking right" ignition is* `On`, the direction blinking cycle should be started (see Req. **ELS-1**).



(a) Initialization (b) *User Interaction:* Turn Engine On (c) *User Interaction:* Move Pitman Arm to Downward7

(d) *System Reaction:* Left Direction Indicators Turn On (e) *System Reaction:* Left Direction Indicators Turn Off (f) *User Interaction:* Activate Warning Lights *System Reaction:* All Direction Indicators Turn On

(g) *System Reaction:* All Direction Indicators Turn Off (h) *User Interaction:* Deactivate Warning Lights *System Reaction:* Left Direction Indicators Turn On (i) *System Reaction:* Left Direction Indicators Turn Off

Figure 3.7.: Validation of **ELS-1**, **ELS-8**, **ELS-12** from User's Perspective in ProB2-UI (Visualization and User Interaction in VisB, System Reaction via SimB)

Validation by Interactive Simulation. Based on requirements and model [154], we encode SimB listeners and activations. We use VisB to perform user interactions described in **ELS-1**, **ELS-8**, and **ELS-12** and check if the car reacts as desired. Initially, the engine is off, warning lights are not active, and the pitman arm is in `Neutral` position (see Figure 3.7a). First, the driver turns on the engine (see Figure 3.7b) and moves the pitman arm to `Downward7` (see Figure 3.7c) corresponding to the user interaction of **ELS-1**. The car's left direction indicators are expected to blink every 500ms, which is confirmed in Figure 3.7d and Figure 3.7e. Secondly, the driver activates the warning lights, and checks if all direction indicators blink every 500ms (described in **ELS-8**). This user interaction is shown in Figure 3.7f, and the car's reaction is confirmed in Figure 3.7f and Figure 3.7g. Finally, the driver deactivates the warning lights (see Figure 3.7h), and checks if all left direction indicators blink every 500ms (as pitman arm is still in `Downward7`; requirement **ELS-12**). The desired reaction is confirmed by the user in Figure 3.7h and Figure 3.7i.

Validation by VisB State Diagram. After running user scenarios for **ELS-1**, **ELS-8**, and **ELS-12** via interactive simulation (described in Figure 3.7), we inspect the VisB state diagram (see Figure 3.8). For clarity, we replaced the state diagram nodes (textual representation) with the corresponding graphical objects. This is currently done manually, but we attempt to automate it in the future.

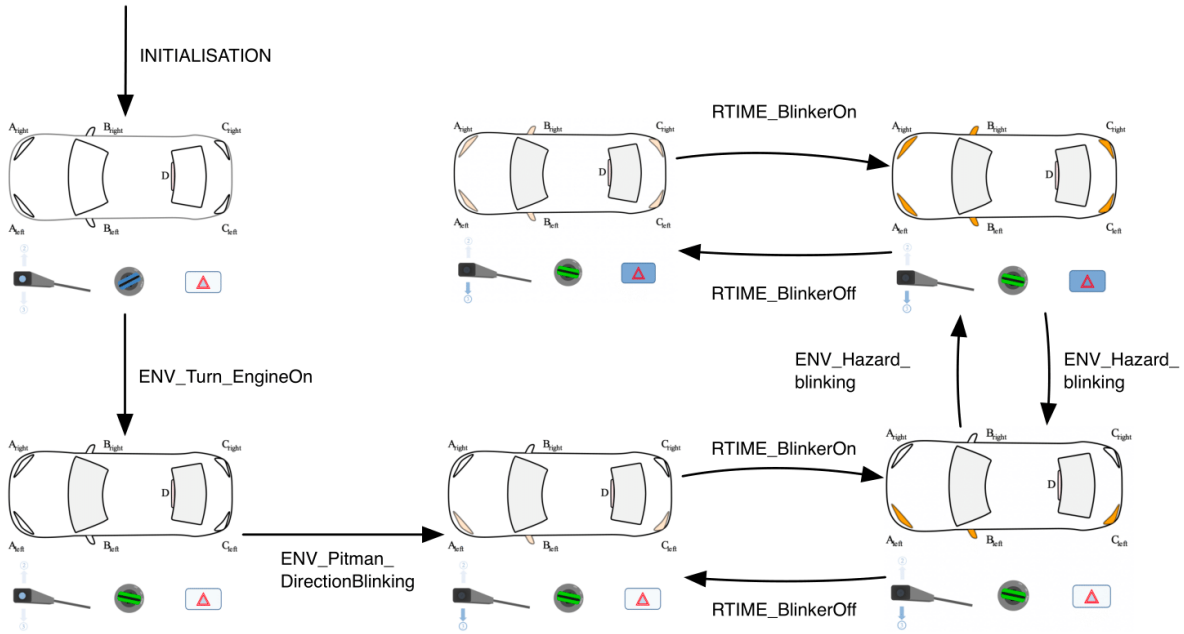


Figure 3.8.: State Diagram from Figure 3.7

This results in Figure 3.8 with six states. The edges represent events executed in Figure 3.7. Thus, Figure 3.8 does not show events that are not part of the scenario in

Figure 3.7. The diagram shows that turning on the engine does not result in any reaction from the car, while user events on the pitman arm and warning lights button trigger the flashing cycles. Deactivating the hazard lights switches to the left blinking lights cycle as the pitman arm is still in `Downward7`.

3.5. Related Work

Animation, Testing. In animation, the user has to execute all events manually. *Interactive simulation* only requires users to execute user events manually after which system events are executed automatically. This improves usability for users but requires additional effort in encoding the simulation. Existing animators are, e.g., the ProB animator [152], and AsmetaA for ASMs [34]. Domain-specific scenarios are supported for Event-B with Gherkin using ProB [212, 74], and for ASMs with AsmetaV [47] and the AValLa language, and Asmeta2C++ [36]. As we ask: "*when the user executes an event, then how does the system react?*", there is some overlap between such scenarios and SimB activation diagrams.

The scenario checker uses ProB for animation and BMotionStudio [134] for visualization of formal models [213]. It distinguishes between external (executed manually) and internal events (fired automatically), similar to our work. SimB simulates events more precisely as it encodes probabilistic and timing behavior.

Simulators. There are various simulators like SimB: JeB [167], AsmetaS [83], Uppaal [29], or the co-simulation tool INTO-CPS [223]. In particular, Uppaal and INTO-CPS can handle continuous time, while our approach works with discrete time only. A more detailed comparison is given by Vu et al. [237].

Visualizations. VisB has been compared with BMotionWeb [137, 133], BMotionStudio [134], and ProB's animation function [155] in [243]. Those tools all make it possible to interact with a formal model via a visualization. Unlike this work, they do not support easy simulation of autonomous events as a reaction to a user event. BMotionWeb also includes a feature to generate a projection diagram on graphical objects which is an inspiration for VisB state diagrams.

Brama [206] allows animation of formal B models through Flash visualizations, and contains listeners to simulate system events. Brama was also used in an architecture by Méry and Singh where real-time data were collected, trained, and used to animate formal models [171]. SimB also uses listeners to trigger simulations with timing and probabilistic behavior. While Brama was a standalone Flash application, SimB is fully integrated into ProB2-UI, allowing for use with other features in ProB2-UI. Using real-time data in SimB is still future work.

PVSio-Web [241] is a tool to create prototypes for PVS models. Like SimB, it also extends simulation features to support human-machine interfaces.

Looking a bit further, there is also a considerable amount of research on formal methods and human-computer interaction (e.g., [63]); some may benefit from our new

tooling. Other work on combining verification with simulation (e.g., [205]) can inspire further linking our simulation techniques with B verification techniques. We may also investigate using CSP (already supported by ProB) and its associated refinement notions with support for external and internal choice, as a means of formally verifying our user interactions.

3.6. Conclusion and Future Work

This work presented SimB's *interactive simulation* which is coordinated with domain-specific interactive VisB visualizations. The feature is realized by SimB listeners which recognize user interactions (e.g. in VisB) and trigger SimB simulations, i.e., autonomous events with probabilistic and timing behavior. *Interactive simulation* helps (1) to improve the user experience of formal models, and (2) to validate requirements related to user interactions and expected system reactions. For domain-specific users, *interactive simulation* is more accessible than LTL as writing LTL requires expertise. Compared to classic animation, *interactive simulation* reduces the user's effort to interact with formal models as the user only has to execute user events while automatic events are simulated. In exchange, *interactive simulation* requires additional effort to be invested in modeling the simulations including human/machine interaction. We also presented state diagrams for domain-specific visualizations in VisB, supporting domain-specific inspection. In an automotive case study, we demonstrated the effectiveness of *interactive simulation* and those state diagrams. Here, we successfully validate requirements by executing user events and observing desired system reactions.

- Case studies are available at: `https://github.com/favu100/SimB-examples/tree/main/Interactive_Examples`
- ProB2-UI (with presented features) is available at: `https://prob.hhu.de/w/index.php/ProB2-UI`
- More information on SimB including *interactive simulation* are available at: `https://prob.hhu.de/w/index.php?title=SimB`

In the future, we plan to formalize SimB's semantics. This could help verify SimB's *interactive simulator*. Another future work is the refinement of SimB simulation (as mentioned in [237]) which also affects SimB listeners.

Acknowledgements. We would like to thank Sebastian Stock and anonymous reviewers for proofreading and giving feedback.

4. Development and Validation of a Formal Model and Prototype for an Air Traffic Control System

Abstract. This article presents an Event-B model and an interactive GUI prototype for an air traffic control system called the arrival manager (AMAN). AMAN is a safety-critical interactive system designed for air traffic controllers to manage landings at an airport. The presented formal model consists of a human-machine interface comprising interactive and autonomous parts. Safety properties of the system were proven using the Rodin platform, while validation was carried out using the ProB tool. We turned the formal model into an executable AMAN prototype by combining interactive domain-specific visualizations and automatic simulation using the VisB and SimB components of ProB. We used validation obligations (VOs) to systematically validate the model's and the prototype's compliance with the requirements and uncovered some contradictions and ambiguities in the case study.

CCS Concepts. Software and its engineering → Software verification and validation, Software and its engineering → Formal methods, Software and its engineering → Requirements analysis, Human-centered computing → Interface design prototyping

Keywords. Event-B, Refinement, Prototype, Simulation, Visualization, Validation Obligations

Funding. The research presented in this article has been conducted within the IVOIRE project, which is funded by "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF) grant # I 4744-N. The work of Sebastian Stock and Atif Mashkoor has been partly funded by the LIT Secure and Correct Systems Lab, which is sponsored by the province of Upper Austria.

4.1. Introduction

In this work, we model an air traffic control system, namely the arrival manager (AMAN), described in a case study presented by Palanque and Campos [186]. AMAN is a semi-interactive tool with a graphical user interface (GUI) for air traffic controllers (ATCos) that consists of both interactive and autonomous parts. While AMAN automatically computes

a landing sequence for arriving airplanes, a human, i.e., an ATCo, can manually intervene and change this sequence or manage time slots. From a low-level GUI perspective, the behavior of mouse events, such as mouse position, click, and zoom, becomes particularly important.

We create a formal model to check its correctness with validation and verification techniques. With formal methods, we can ensure specific safety properties for AMAN in all possible situations and uncover ambiguities in the requirements. The state-of-the-art techniques and tools also enable the refinement of high-level functionalities in AMAN to low-level events, including mouse behavior at the pixel level. Building on this, we developed a prototype for domain experts to support the validation process.

We have developed our model using the Event-B [5] modeling language, which has been effective to model interactive safety-critical systems, including human-machine interfaces, e.g., by Aït-Ameur et al. [16] and Singh [210]. A unique feature of our formal model is that it also models GUI events at the pixel level in later refinements.

Based on our Event-B model, we create an interactive prototype of the AMAN GUI by combining visualization via VisB [243] and simulation via SimB [237]. Stakeholders and domain experts can use this prototype for validation and experimentation.

The model itself was developed with the Rodin platform [5]. We verify the consistency of our model through model checking with ProB [153] and discharging of proof obligations. However, our primary focus is to systematically validate the requirements for the AMAN system that Palanque and Campos [186] provide. Our goal was to present validation results that are intelligible to domain experts, enabling them to provide feedback. To this end, we employ the systematic approach of validation obligations (VOs) [166] and use a VO management system implemented in ProB2-UI [25]. Validation is done with a variety of techniques, such as (temporal) model checking, proving, animation, trace replay, model coverage statistics, and visualizations from different domain perspectives.

This article is an extended version of our ABZ 2023 case study track paper [88]. We have extended our contribution in multiple ways:

- A more detailed description of the Event-B model
- In-depth description of the AMAN prototype based on the Event-B model, including details about the VisB visualization and the SimB simulation
- Enhanced validation process by employing state-space projection diagrams and checking certain behaviors in the AMAN prototype
- A retrospective analysis of the lessons learned

After describing the background, we present the AMAN system and its functionality in Section 4.3. We then present our Event-B model for the AMAN system in Section 4.4. We then focus on verification via model checking and POs (see Section 4.5). Based on the formal model, we describe how we implemented our AMAN prototype using the visualization tool VisB and the simulator SimB (see Section 4.6). Afterward, we validate the formal model with VOs (see Section 4.7). Section 4.8 highlights the lessons learned

during this modeling and analysis exercise, showing parts of the specification where VOs helped to formulate questions for the stakeholders, make assumptions, uncover ambiguities, and show the importance of a prototype. Section 4.9 discusses some related work, and finally, we conclude in Section 4.10.

4.2. Background

Event-B [5] is a state-based formal method based on first-order logic and set theory.

Event-B uses a refinement-based approach, where systems are gradually developed at several levels, from an abstract representation to more concrete ones closer to implementation.

```
1: context Zoom_Ctx
2: constants ZOOM_LEVELS
3: axioms
4:   @axm1 ZOOM_LEVELS =
       {15,20,25,30,35,40,45}
```

Listing 4.1. Context for zoom levels in Event-B.

Each level is developed by refining the previous level, typically adding details, while conforming to the behaviors of the abstract level. This gradual enhancement is performed over multiple steps and is called a refinement chain.

Within Event-B, a component is either a *context* or a *machine*. A context defines static parts of a model, namely *constants* and carrier *sets* (i.e., new abstract types), along with *axioms*, which constrain the possible values of the constants and sets.

A machine describes the dynamic parts of a model. As such, it contains *variables*, which represent the state, and *invariants*, which must always be true and also define the types of the variables. A machine contains an *initialization* and *events*, which can modify the values of the variables and, thus, the current state. Events consist of *parameters*, a *guard*, and *actions*. When the guard is true for some parameter values, the event is *enabled*, i.e., it can be executed by performing the actions (various

```
1: machine Zoom sees Zoom_Ctx
2: variables zoomLevel
3: invariants
4:   @inv1_1 zoomLevel ∈ ZOOM_LEVELS
5: events
6:   event INITIALISATION
7:   then
8:     @act1_1 zoomLevel := 45
9:   end
10:
11:  event changeZoom
12:  any newZoom
13:  where
14:    @grd1_1 newZoom ∈ ZOOM_LEVELS
15:    @grd1_2 newZoom /= zoomLevel
16:  then
17:    @act1_1 zoomLevel := newZoom
18:  end
19: end
```

Listing 4.2. Simple machine in Event-B with an event to change zoom.

forms of assignments). Listing 4.1 and Listing 4.2 show, respectively, an Event-B context and machine to change the zoom level. A machine can also have other clauses, such as *variants*, which are relevant for proving the absence of infinite loops, but we do not use them in this case study.

The **Rodin** platform [7] is a toolset for modeling and verifying systems in Event-B. Based on an Event-B model, Rodin generates proof obligations (POs), which must be discharged to verify the system. Solvers in Rodin perform the proofs themselves, either fully automatically or guided interactively when the solvers are not strong enough. A PO is a predicate to be proven that ensures a specific property in the model, such as invariant preservation, well-definedness, the absence of an infinite loop, or even consistency between refinement steps.

ProB [152] is an animator, constraint solver, and model checker for formal methods including B, Event-B, Z, CSP, and TLA$^+$. ProB2-UI [25] is a graphical user interface, built on top of ProB. Using ProB2-UI, one can manage projects with multiple machines and their validation tasks. ProB2-UI supports various verification and validation techniques, such as:

- domain-specific visualization with VisB [243]
- timed probabilistic simulation with SimB [237]
- various model checking techniques (LTL [194], CTL, symbolic [128])
- animation and trace replay
- domain-specific state space visualizations and projections [136]
- evaluation of coverage criteria

More recently, ProB2-UI has been extended by a *validation obligation* (VO) manager. **VOs** have been introduced to structure and guide validation [166, 217], analogous to the POs used for verification in Event-B. We will briefly introduce VOs in Section 4.7.

VisB is a visualization tool in ProB2-UI to create domain-specific visualizations [243]. With VisB, a user can view the formal model's state graphically and execute operations by clicking on graphical elements. The visualization comprises SVG graphics linked to the formal model via a VisB glue file. The idea of VisB is similar to the model-view-controller (MVC) pattern [239].

SimB is a simulation tool in ProB2-UI which allows timed probabilistic simulation [237] with user interaction [236] for formal models. To use SimB, a modeler has to annotate events with timing and probabilistic behavior for execution. In particular, those annotations encode an *activation diagram* describing how events trigger each other with probabilities and delays.

Combining VisB with SimB, one can create prototypes that simulate realistic human-machine interaction, capturing the realistic behaviors of system reactions and autonomous events. SimB supports a feature called *interactive simulation* [236], which enables defining listeners on user interactions that trigger a process as a simulation.

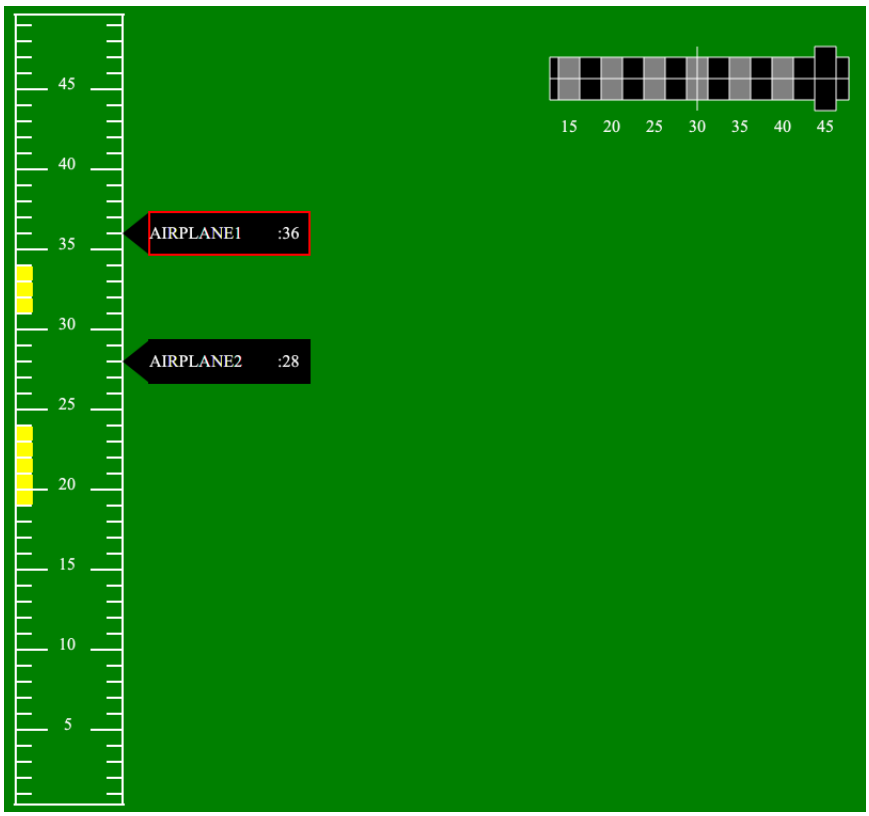


Figure 4.1.: AMAN Prototype; Visualization is created with VisB; simulation is created with SimB.

4.3. AMAN System

This section introduces the core functionalities of an air traffic control system called Arrival Manager (AMAN) [186]. The purpose of AMAN is to manage airplane arrivals at an airport, aiming for minimum separation between them. Figure 4.1 shows a prototype of the AMAN GUI, which is developed in this work using VisB and SimB.

There are two (human) air traffic controllers (ATCos) in the context of AMAN. First, the planning air traffic controller (PLAN ATCo) organizes airplane traffic through the AMAN system. Second, the executive air traffic controller (EXEC ATCo) communicates with the pilots but does not interact with the AMAN system. Because our work focuses on the AMAN system, we will only consider the PLAN ATCo, which interacts with AMAN. In the rest of this article, the term „ATCo“ will refer only to the PLAN ATCo.

The AMAN system contains both an interactive and an automatic component. The automatic component monitors airplanes near the airport and schedules them in a timeline for landing at the airport (called landing sequence) as shown on the left-hand side of Figure 4.1). The automatic component computes the trajectory and the landing time based on the airplanes' technical data. We abstracted this behavior in the formal model presented in this article. If the automatic AMAN component stops working, i.e., no longer responds within a given time frame, the GUI must inform the ATCo. In this

mode, the ATCo schedules airplanes manually rather than through the AMAN GUI.

The ATCo operates the interactive components of AMAN. From the ATCo's perspective, airplanes appear in the landing sequence. The ATCo can perform certain operations on the landing sequence through the AMAN system, such as moving an airplane to a different time slot or putting an airplane on hold. Airplanes on hold are highlighted with a red frame, e.g., in Figure 4.1, `AIRPLANE1` is on hold. Further interactions with the landing sequence include blocking time slots (yellow in Figure 4.1), e.g., when the runway is occupied for other reasons during that time. It is, hence, required that no airplanes are scheduled in a blocked time slot. While AMAN should never schedule an airplane in a blocked time slot, the ATCo can block a time slot in which an airplane was already scheduled; AMAN processes such situations, which results in moving the airplane to a different available slot. An air traffic control can also change the zoom level of the GUI (see top-right of Figure 4.1).

Users perform all user interactions with the AMAN GUI using a mouse. The screen displays a mouse cursor, and the GUI must respond to mouse events, such as moving the mouse, pressing and releasing the mouse button, and dragging while keeping the mouse button pressed.

The main challenges of this work are: (1) creating a formal model that captures the interactive and automatic components of AMAN, (2) employing a prototype with a GUI for domain experts and stakeholders to experiment with, and (3) validating that the formal model fulfills the specification and verifying the consistency of the specification and the formal model. With the AMAN model and the prototype we created in this work, we validate different properties such as:

- Requirements concerning AMAN's automatic components, e.g., that airplanes can only be added/removed by AMAN's automatic events
- Requirements concerning AMAN's interactive components, e.g., that there must be a separation distance between airplanes in the landing sequence.
- Requirements concerning the GUI, e.g., that airplane labels do not overlap when displayed.

We rely on the specification document provided by Palanque and Campos [186] for all of these steps. Table 4.1 shows an overview of the requirements covered in this article. All requirements except **BEH1**, **BEH2**, and **Req5.1** are given explicitly in the specification document. **BEH1**, **BEH2**, and **Req5.1** are requirements that we derived from the specification text and during the validation process. **BEH1** was derived while validating and reasoning about the behavior of airplanes in the landing sequences, particularly, **Req1** and **Req2**. **BEH2** was derived from the requirements document while reasoning about airplanes on hold (for more details, see Section 3.1 in the specification document). **Req5.1** was extracted from Section 2.2 of the specification document. While **Req5.1** is related to **Req5**, it is not directly derived from **Req5**. More details on their validation are provided in Section 4.7.

External Events	
Req1	Planes can [be] added to the flight sequence e.g. planes arriving in a close range of the airport
Req2	Planes can be removed from the flight sequence e.g. planes changing their landing airport for some reason
Req3	Planes moved earlier or later on the timeline by the PLAN ATCo thus requiring from AMAN the processing of a new prediction;
Req4	Planes put on hold by the PLAN ATCo. Planes removed from HOLD will appear as normal aircrafts handled by AMAN.
Safety Requirements	
Req5	Aircraft labels should not overlap;
Req6	An aircraft label cannot be moved into a blocked time period;
Req7	Moving an aircraft label might not be accepted by AMAN if it would require a speed up of the aircraft beyond the capacity of the aircraft;
Req8	If AMAN is not functioning (e.g. no update after 10 seconds) the ATCo must be informed about the failure and landing sequence preparation will be done manually (without AMAN).
Interaction Requirements	
Req15	the HOLD button must be available only when one aircraft label is selected;
Req16	the zoom value cannot be bigger than 45 and smaller than 15;
Req17	aircraft labels must always be positioned in front of a small bar of the timeline;
Req18	Lift of the zoom slider should always be located on the slider bar
Req19	the value displayed next to the zoom slider must belong to the list of seven acceptable values for the zoom
Req20	each movement of the mouse on the ATCo table must be reflected by a movement of the cursor on the screen
Req21	there must be one and only one mouse cursor on the screen
Req22	Hold(aircraft) function can only be triggered after a mouse press and a mouse released have been performed on the HOLD button.
Req23	Hold(aircraft) function must not be triggered if there is not a mouse press and a mouse released performed on the HOLD button.
Derived Requirements	
Req5.1	[...] a landing separation of 3 minutes between aircraft is requested.
BEH1	An AMAN update adds scheduled airplanes, which can only be removed by an AMAN update.
BEH2	The ATCo can always put any airplane on hold, and only an AMAN update can remove an airplane on hold from the landing sequence.

Table 4.1.: Requirements covered in this article.

4.4. AMAN Model

This section describes the AMAN formal model that we developed in Event-B based on the specification [186] by Palanque and Campos.

Figure 4.2 provides a high-level view of the AMAN system.

AMAN's automatic components compute a trajectory for each airplane approaching the airport and schedule it in the landing sequence. The ATCo can also modify the

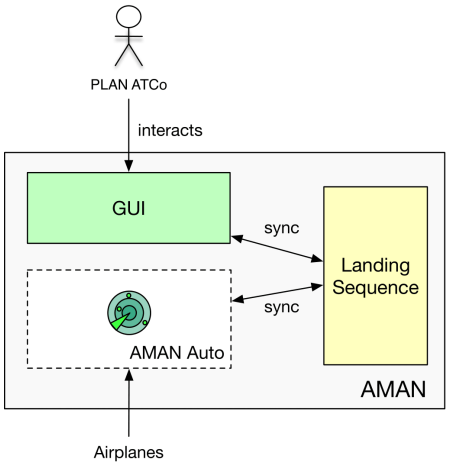


Figure 4.2.: High-Level View of the AMAN System showing the interaction between AMAN's Automatic Components, the Landing Sequence, and Airplanes

landing sequence via the GUI, e.g., by moving an airplane to a different time slot. Here, we must ensure that changes made by AMAN and ATCo are in sync.

In the formal model, we specify the GUI and the behavior of the landing sequence in detail, abstracting away AMAN's automatic components. As a result, we model AMAN's automatic events as airplanes (dis)appearing in the landing sequence; we do not precisely model the computation process.

In the following, we provide more details on the Event-B model. We create a formal model consisting of an abstract model `M0` and 10 refinement steps (`M1`, `M2`, ..., `M10`)[1]

In the modeling steps from `M0` to `M5`, we focus on high-level aspects of AMAN. These are autonomous AMAN events and interactive events such as moving airplanes, blocking time slots, and putting airplanes on hold. In the next refinement steps (`M6` to `M9`), we refine high-level interactive events into low-level mouse events. For instance, blocking a time slot is refined by moving the mouse to the corresponding position, clicking, and releasing it. The final refinement step `M10` refines `M9` one step further to a concrete pixel representation of all graphical UI elements. We create an AMAN prototype based on the formal model (presented in Section 4.6). We perform various verification and validation activities with the formal model and the AMAN prototype.

4.4.1. Refinement Hierarchy

HAMSTERS [14, 162, 77] is a task modeling notation to describe human activities and their relation to system events. The specification document provided by Palanque and Campos [186] includes HAMSTERS diagrams to describe activities in AMAN at a high level. Figure 4.3 shows a version of the HAMSTERS diagram with the main

[1]The model and all other mentioned files are available in a public Git repository: `https://github.com/hhu-stups/AMAN-case-study/tree/c09836985b5141dbe50aa7cd46ac9d7a6bccc18d`

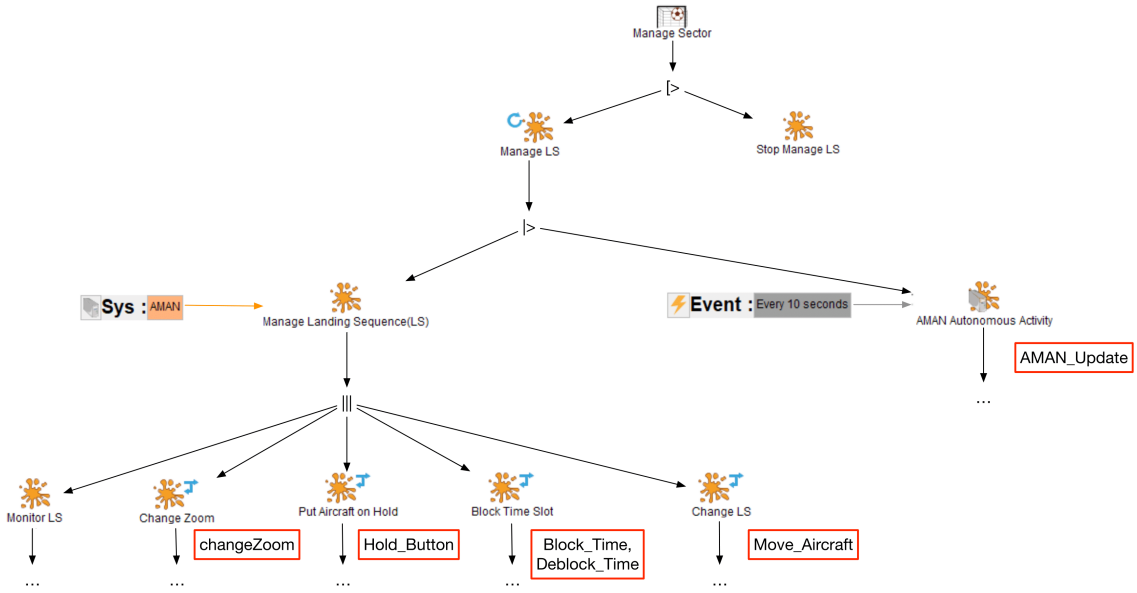


Figure 4.3.: AMAN HAMSTERS Diagram (based on Figure 10 from the specification [186]); red boxes and their content are not part of the HAMSTERS diagram. We add them to emphasize the correspondence to Figure 4.4. Red boxes contain the corresponding event names in the Event-B model.

activities of the AMAN system. We modified the HAMSTERS diagram by Palanque and Campos [186] to show only the parts of the AMAN that we focused on.

At the top level, the HAMSTERS diagram begins with the system either managing the landing sequence or stopping to do so. The next level in the HAMSTERS diagram consists of a user interaction (Manage Landing Sequence) or an AMAN Autonomous Activity.

Referring to the refinement steps of the formal model, we follow the structure of the HAMSTERS diagram, particularly implementing it in `M0-M5`. We only modeled the left branch of Figure 4.3, so we have not implemented stopping/shutting down AMAN. Following the HAMSTERS diagram, we model the corresponding requirements (see Table 4.1) in the formal model. As a result, one can trace the refinement hierarchy to the HAMSTERS diagram and the corresponding requirements.

For the refinements following `M5`, we no longer closely follow the HAMSTERS diagram because the remaining parts of the diagram mainly focus on the human aspect, which our model does not represent in detail. The introduced variables correspond to the introduced events in Figure 4.4.

4.4.2. AMAN Update and Landing Sequence (M0, M1)

In `M0`, we introduce the `AMAN_Update` event, which modifies the set of airplanes scheduled for landing (see Listing 4.3). This event encapsulates the autonomous part of AMAN,

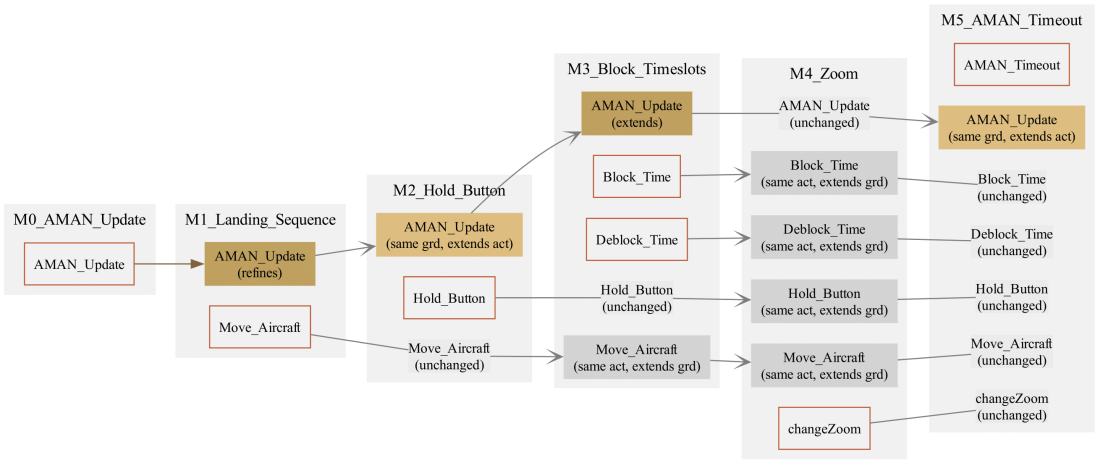


Figure 4.4.: Event refinement hierarchy until M5 (generated by ProB); this figure shows which events are introduced throughout the refinement steps and how they are refined; nodes with borders are newly introduced events; meanings of the other nodes' appearances/colors are given in parentheses.

which is refined in later development steps.

```
1: event AMAN_Update
2: any newScheduledAirplanes
3: where
4:   @grd0_1 newScheduledAirplanes ⊆ AIRPLANES
5: then
6:   @act0_1 scheduledAirplanes := newScheduledAirplanes
7: end
```

Listing 4.3. Event for AMAN update (`AMAN_Update`) introduced at M0; the set of scheduled airplanes is assigned to any subset of all airplanes (`AIRPLANES`).

In `M1`, the scheduled airplane set is refined to a *landing sequence* with associated landing times expressed as minutes relative to the current time. We could have modeled absolute times with an increasing variable representing the current time. However, this would have led to an infinite state space. Therefore, we model the timing aspects as follows: the current time is always 0, and all times are represented relative to the current time. This renders the state space finite concerning timing aspects (cf. [139, 198]; for more details on model checking, see Section 4.5). Listing 4.4 shows the refinement of the `AMAN_Update` event from Listing 4.3 in `M1`. In Listing 4.4, one can see that `AMAN_Update` uses a *partial function* for the new landing sequence (`grd1_1`) where its airplanes (i.e., domain) must match `newScheduledAirplanes` from `M0` (specified in the witness `newScheduledAirplanes` after the `with` keyword). We do not use a total function

because not all airplanes are present in the landing sequence.

```
1:  event AMAN_Update refines AMAN_Update
2:  any new_landing_sequence
3:  where
4:    @grd1_1 new_landing_sequence ∈ AIRPLANES ⇸
        PLANNING_INTERVAL
5:    @grd1_2 ∀a1,a2. a1 ∈ dom(new_landing_sequence)
6:      ∧ a2 ∈ dom(new_landing_sequence) ∧ a1 ≠ a2
7:      ⇒ (DIST(new_landing_sequence(a1) ↦ new_landing_sequence
          (a2)) ≥ AIRCRAFT_SEPARATION_MIN)
8:  with
9:    @newScheduledAirplanes newScheduledAirplanes =
10:     dom(new_landing_sequence)
11: then
12:   @act1_1 landing_sequence := new_landing_sequence
13: end
```

Listing 4.4. Refined AMAN update event (`AMAN_Update`) at M1; set of scheduled airplanes is refined by a partial function (⇸ symbol) representing the landing sequence where each scheduled airplane is mapped to the scheduled landing time; two distinct airplanes are separated by at least `AIRCRAFT_SEPARATION_MIN` minutes.

Note that in B, a function is represented as a set of pairs; in this case, a set of pairs of airplanes with associated landing times. The range of the partial function contains the time slots in the planning interval, which must satisfy a spacing requirement laid out in Section 2.2 of the requirements document [186] (**Req5.1** in Table 4.1):

> [...] a landing separation of 3 minutes between aircraft is requested.

We implement this requirement in `grd1_2` of Listing 4.4, stating that two (distinct) airplanes in the `landing_sequence` must be separated by `AIRCRAFT_SEPARATION_MIN` minutes. We define `AIRCRAFT_SEPARATION_MIN` as a constant; to match the requirement, we instantiate this constant as `AIRCRAFT_SEPARATION_MIN = 3` (minutes). However, one can define other values for this constant for experimentation as well. `DIST` is a function that computes the distance between the time slots as follows:

$$\texttt{DIST} = (\lambda(x \mapsto y).x \in \mathbb{Z} \land y \in \mathbb{Z} \mid max(\{y - x, x - y\}))$$

`M1` also enables the ATCo to move an airplane in the landing sequence from one time slot to a different one via the `Move_Aircraft` event with respective parameters `aircraft` and `time` (see Listing 4.5). Furthermore, we did not model the landing of airplanes because the specification did not provide details about this. Instead, we assume that AMAN removes landed airplanes from the landing sequence, just like airplanes that disappear from the landing sequence for any other reason.

In Listing 4.5, the guard grd1_1 ensures that the airplane is from the landing sequence, while grd1_2 and grd1_3 ensure that the assigned time slot is within the ATCo's operating horizon and different from the currently assigned time slot. grd1_4 ensures that the desired minimum distance still separates all airplanes after executing Move_Aircraft.

```
1:  event Move_Aircraft
2:  any aircraft time
3:  where
4:    @grd1_1 aircraft ∈ dom(landing_sequence)
5:    @grd1_2 time ∈ PLANNING_INTERVAL
6:    @grd1_3 time ≠ landing_sequence(aircraft)
7:    @grd1_4 ∀a. a ∈ dom(landing_sequence) \ {aircraft}
8:      ⇒ DIST(landing_sequence(a) ↦ time) ≥
          AIRCRAFT_SEPARATION_MIN
9:  then
10:   @act1_1 landing_sequence(aircraft) := time
11: end
```

Listing 4.5. Event for moving an airplane (Move_Aircraft) introduced at M1; this event assigns an airplane to a different time slot; the time slot must be in the ATCo's planning horizon while fulfilling the spacing requirement of 3 minutes.

As mentioned earlier, Section 2.2 of the requirements document [186] states that airplanes must be scheduled at least 3 minutes apart. The guards explicitly considered this property when formalizing AMAN_Update and Move_Aircraft. To ensure that all events preserve this property, M1 features this important invariant:

$$\forall a1, a2.\ a1 \in \texttt{dom}(\texttt{landing\_sequence}) \land a2 \in \texttt{dom}(\texttt{landing\_sequence}) \land a1 \neq a2 \Rightarrow \texttt{DIST}(\texttt{landing\_sequence}(a1) \mapsto \texttt{landing\_sequence}(a2)) \geq \texttt{AIRCRAFT\_SEPARATION\_MIN} \tag{4.1}$$

4.4.3. Putting Airplanes on Hold (M2)

M2 introduces the event for clicking the *hold button*. First, we model the set of airplanes on hold (in the new variable held_airplanes) as a subset of airplanes in the landing sequence.

As described in the specification document [186], the ATCo can put airplanes on *hold*. We assume that the ATCo cannot reverse this event. Airplanes on hold will be removed from the landing sequence and reappear at a later stage.

The new Hold_Button event (see Listing 4.6) takes an airplane as a parameter and adds this to the set of airplanes on hold. As specified in Section 3.1 of the requirements document [186], this airplane must be in the landing sequence and not yet on hold. This is encoded in grd2_1 in Listing 4.6.

```
event Hold_Button
any airplane
where
  @grd2_1 airplane ∈ dom(landing_sequence) \ held_airplanes
then
  @act2_1 held_airplanes := held_airplanes ∪ {airplane}
end
```

Listing 4.6. Event for clicking the HOLD button (`Hold_Button`) introduced at M2; this event puts an airplane in the landing sequence on hold.

Following Section 3.1 of the requirements document [186], a future AMAN update shall *eventually* remove airplanes on hold from the landing sequence. However, an airplane on hold could be rescheduled to a different time slot. Considering both aspects, we refine `AMAN_Update` to handle airplanes on hold as shown in Listing 4.7. With the new landing sequence provided as a parameter in the `AMAN_Update` event (see Listing 4.4), an airplane on hold (as well as any other airplane) may either remain in or be removed from the landing sequence. As long as an airplane on hold remains in the landing sequence, AMAN has not yet removed it. If an airplane disappears, it has either landed or finally been removed by AMAN because it was on hold (or for another reason). We abstract this detail away in our Event-B model.

```
event AMAN_Update extends AMAN_Update
then
  @act2_1 held_airplanes := held_airplanes ∩
    dom(new_landing_sequence)
end
```

Listing 4.7. Refined AMAN update event at M2; airplanes on hold might be removed from the landing sequence or re-scheduled.

4.4.4. Blocking Time Slots (M3)

The third refinement, `M3`, introduces events to block/unblock time slots (stored in the `blockedTime` variable; see Listing 4.8 and Listing 4.9). We extract the details for these events from Section 3.2 of the requirements document [186].

Concerning the `AMAN_Update` and `Move_Aircraft` events, we must ensure that neither AMAN nor the ATCo can move an airplane into a blocked time slot. We encode this behavior in the guards of both events. However, to validate this, we cannot posit the following invariant:

$$\mathtt{ran}(\mathtt{landing\_sequence}) \cap \mathtt{blockedTime} = \varnothing$$

because the user can block a time slot that already contains an airplane in the landing sequence, thus violating the property.

To overcome this, we instead introduced the following invariant:

$$\texttt{blockedTimesProcessed} = \texttt{TRUE} \Rightarrow \texttt{ran(landing\_sequence)} \cap \texttt{blockedTime} = \varnothing \tag{4.2}$$

which will become important when validating the requirement:

Req6: An aircraft label cannot be moved into a blocked time period;

`blockedTimesProcessed` is a helper variable only used in this invariant. It will be set to FALSE by `Block_Time` if the newly blocked time slot `time` already contains an airplane (see `act3_2` in Listing 4.8). It is set to TRUE by `AMAN_Update` (see `act3_2` in Listing 4.10), which ensures that all blocked time slots are free.

```
1: event Block_Time
2: any time
3: where
4:   @grd3_1 time ∈ PLANNING_INTERVAL \ blockedTime
5: then
6:   @act3_1 blockedTime := blockedTime ∪ {time}
7:   @act3_2 blockedTimesProcessed :=
8:     bool(time ∉ ran(landing_sequence) ∧
          blockedTimesProcessed = TRUE)
9: end
```

Listing 4.8. Event for blocking a time slot introduced at M3; a blocked time slot is added to the `blockedTime` set.

```
1: event Deblock_Time
2: any time
3: where
4:   @grd3_1 time ∈ blockedTime
5: then
6:   @act3_1 blockedTime := blockedTime \ {time}
7: end
```

Listing 4.9. Event for deblocking a time slot (`Deblock_Time`) introduced at M3; a blocked time slot is removed from the `blockedTime` set.

As we modeled time slots relative to the current time, an AMAN update must consider how many minutes have passed since the last AMAN update (modeled by the `passed_minutes` parameter in Listing 4.10). `AMAN_Update` then shifts all blocked time slots by `passed_minutes` (encoded with `new_blockedTime` in `grd3_1` and `act3_1` in Listing 4.10).

```
event AMAN_Update extends AMAN_Update
any passed_minutes new_blockedTime
where
  @grd3_0 passed_minutes ∈ ℕ
  @grd3_1 new_blockedTime =
    {t | t ∈ PLANNING_INTERVAL ∧ t + passed_minutes ∈
       blockedTime}
  @grd3_2 ran(new_landing_sequence) ∩ new_blockedTime = ∅
then
  @act3_1 blockedTime := new_blockedTime
  @act3_2 blockedTimesProcessed := TRUE
end
```

Listing 4.10. Refined AMAN update event at M3; airplanes within blocked time slots are removed from landing sequence; blocked time slots are processed, i.e., shifted `passed_minutes` further as time is modeled relative to the current time.

On the one hand, the next minute might have started since the last AMAN update. On the other hand, user interactions have a higher priority according to the requirements document (Response to **Q7** in Section 5 of the requirements document [186]); thus, a user interaction could—in theory—take several minutes. With the introduction of blocked time slots in `M3`, `AMAN_Update` must also ensure that an airplane is never scheduled in a blocked time slot; this behavior is encoded in `grd3_2` in Listing 4.10.

```
event changeZoom
any targetZoom
where
  @grd4_1 targetZoom ∈ ZOOM_LEVELS
  @targetZoom_changed targetZoom ≠ zoomLevel
then
  @act4_1 zoomLevel := targetZoom
end
```

Listing 4.11. Event for changing the zoom level (`changeZoom`) introduced at M4.

4.4.5. Zooming (M4)

`M4` introduces the zoom for the landing sequence. As described in the specification document [186], the zoom level defines how many minutes ahead of the current time the landing sequence is displayed. For instance, a zoom level of 15 means that the landing sequence for the following 15 minutes is visible. Following Section 3.2 of the requirement document [186] and the requirement

> **Req16**: the zoom value cannot be bigger than 45 and smaller than 15;

we define a constant `ZOOM_LEVELS = {15, 20, 25, 30, 35, 40, 45}` for all possible zoom values.

`M4` introduces the `changeZoom` event which updates the zoom represented by the `zoomLevel` variable (see Listing 4.11).

Interactions with time slots and airplanes are limited to the current zoom level. We implement this behavior by adding guards to the interaction events. For `Move_Aircraft` and `Hold_Button`, we introduce the guard shown in Equation (4.3), which ensures that the ATCo can only operate on airplanes within the zoom level. Furthermore, we encode the guard shown in Equation (4.4) for `Move_Aircraft`, `Block_Time`, and `Deblock_Time`, which ensures that the ATCo can only operate on parts of the timeline that are within the zoom level.

$$\texttt{landing\_sequence}(\texttt{aircraft}) \leq \texttt{zoomLevel} \tag{4.3}$$

$$\texttt{time} \leq \texttt{zoomLevel} \tag{4.4}$$

Note that the zoom does not affect AMAN's autonomous activities — AMAN can still schedule airplanes for a time slot that is not visible to the ATCo.

4.4.6. Timeout (M5)

`M5` introduces timeouts for AMAN updates via an `AMAN_Timeout` event. The event shall occur when AMAN does not respond within 10 seconds. Consequently, `AMAN_Timeout` sets `timeout` to `TRUE`, while `AMAN_Update` sets `timeout` to `FALSE`. In our AMAN prototype, the user interface provides feedback that the AMAN is no longer working, according to the requirement:

> **Req8**: If AMAN is not functioning (e.g. no update after 10 seconds) the ATCo must be informed about the failure and landing sequence preparation will be done manually (without AMAN).

4.4.7. Selecting/Deselecting Airplanes (M6)

`M6` adds two events for the ATCo to select/deselect an airplane: `selectAirplane` and `deselectAirplane`. In the formal model, we store the selected airplane in the `selectedAirplane` variable. These events are necessary to interact with airplanes on the landing sequence and with the hold button (see Section 3.2 of the requirements document [186]). We refine moving an airplane and putting an airplane on hold to perform respective events on `selectedAirplane`, matching the following requirement:

> **Req15**: the HOLD button must be available only when one aircraft label is selected;

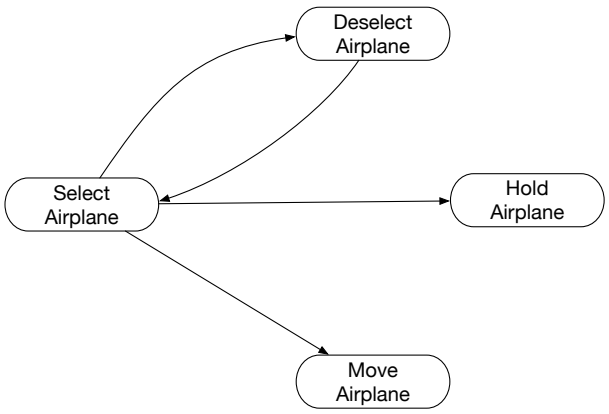


Figure 4.5.: High-level view of user interactions in `M6` with selecting/deselecting, moving, and putting airplanes on hold.

Figure 4.5 shows an overview of the user interactions that we modeled until `M6`. Initially, an AMAN update must schedule airplanes in the landing sequence to enable interactions with them. In our formal model, selecting and deselecting an airplane enable each other, while both events disable themselves. Once an airplane is selected, the ATCo can put it on hold or move it to a different time slot. Putting an airplane on hold also deselects the airplane. Both `selectAirplane` and `deselectAirplane` are used to set up a VisB visualization (see Section 4.6.1).

4.4.8. Detailed User Interaction (M7, M8, M9)

From `M7` to `M9`, we refined user interactions into mouse events, such as *mouse movement*, *mouse clicks*, *mouse drags*, and *mouse releases*. These refinements are challenging because they introduce many variables, and some events are split into sub-events. In particular, these refinements implement tracking the mouse position and all allowed combinations of user interactions.

Figure 4.6 shows a high-level view of how we realized this. We introduced events for the mouse movement, which is always possible. For all relevant graphical elements, we introduce events for clicking, which an ATCo can only perform when the mouse is above the element and the user is currently not clicking or dragging. When an ATCo clicks a graphical element, our formal model enables the events for dragging the mouse and releasing the mouse button while disabling other click events. Releasing the mouse button only applies if the mouse is located on the corresponding graphical element. For instance, there are the following requirements for the hold button:

> **Req22**: Hold(aircraft) function can only be triggered after a mouse press and a mouse released have been performed on the HOLD button.

> **Req23**: Hold(aircraft) function must not be triggered if there is not a mouse press and a mouse released performed on the HOLD button.

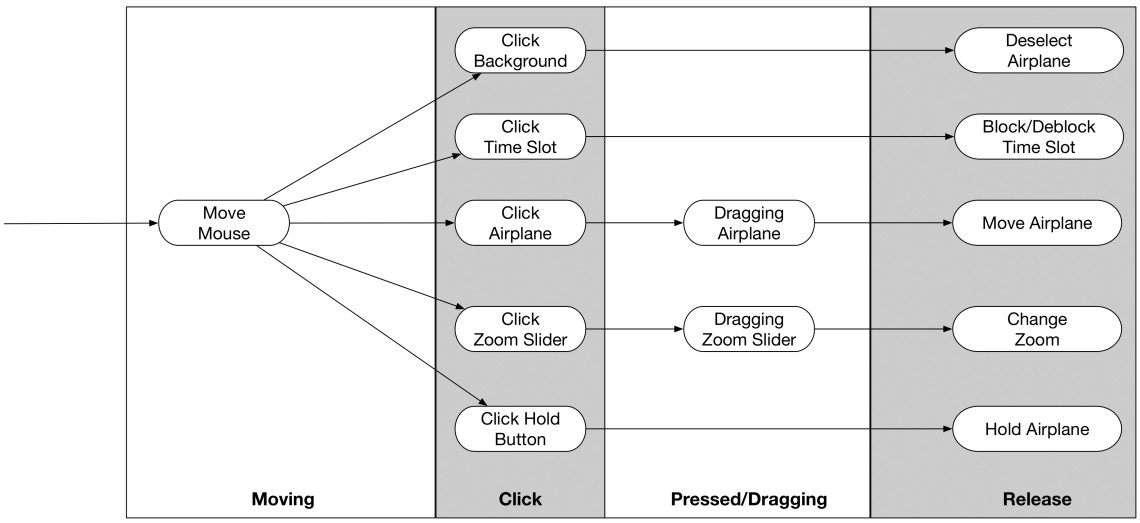


Figure 4.6.: High-level view of user interactions in M9, showing events for moving, clicking, dragging, and releasing the mouse

Thus, M9 models two kinds of events for releasing the mouse button, which either perform or abort the action, depending on whether the mouse is still over the same graphical element. The events in M7 through M9 that perform a mouse action refine the events from M6 and earlier for the corresponding action. The events that abort a mouse action are newly introduced in M7 through M9 and have no effect on the rest of the model.

Furthermore, we assume that when the ATCo is moving the zoom slider, the new zoom level is only applied once the mouse button is released.

Finally, the specification [186] requires prioritizing user interactions over system computations. Therefore, we ensure that AMAN updates do not occur while the ATCo is dragging an airplane. We implement this behavior in the formal model by adding the guard

$$\texttt{dragging\_airplane} = \texttt{FALSE} \tag{4.5}$$

for the AMAN_Update event at M7. dragging_airplane is a boolean variable introduced at M7. The value is assigned to TRUE when the user is dragging an airplane and FALSE when the user finishes an airplane movement or stops dragging the airplane.

4.4.9. Concrete Graphical Interface (M10)

M10 models a raster-based UI rendered on a screen. All UI elements are assigned concrete pixel coordinates, establishing a total mapping from pixels on the screen to the corresponding UI elements. We implement this behavior with the variable ui_element_at_point, which is a total function of type POINTS $\rightarrow$ mouse_positions. POINTS is the set of pixel coordinates (a subset of $\mathbb{N}_0 \times \mathbb{N}_0$) and mouse_positions is an enumerated set: {hold_button_pos, airplane_pos, block_time_pos, nothing_pos, zoom_slider_pos}.

Based on these coordinates and mappings, we introduced several theorems and invariants to ensure that all UI elements are always placed appropriately, e.g., that they do not go off-screen and that they fulfill the requirement:

Req5: Aircraft labels should not overlap;

For example, we added the following invariant (related to invariant 4.1) to ensure that the airplane labels for the landing sequence never overlap with each other in the UI:

$$\begin{aligned} \forall t_1, t_2 \cdot\ & t_1 \in 1..\texttt{zoomLevel} \land t_2 \in 1..\texttt{zoomLevel} \land t_1 < t_2 \\ & \land\ t_1 \in \text{ran}(\texttt{landing\_sequence}) \\ & \land\ t_2 \in \text{ran}(\texttt{landing\_sequence}) \\ & \Rightarrow \texttt{airplane\_points}(t_1) \cap \texttt{airplane\_points}(t_2) = \varnothing \end{aligned} \tag{4.6}$$

Given the current zoom level and a time slot, the set of pixels occupied by an airplane label is defined as:

$$\begin{aligned} & \texttt{AIRPLANE\_X}..(\texttt{AIRPLANE\_X} + \texttt{AIRPLANE\_WIDTH} - 1) \\ \times\ & \texttt{AIRPLANE\_YS\_BY\_ZOOM}(\texttt{zoom})(\texttt{time})..(\texttt{AIRPLANE\_YS\_BY\_ZOOM}(\texttt{zoom})(\texttt{time}) + \\ & \texttt{AIRPLANE\_HEIGHT} - 1) \end{aligned} \tag{4.7}$$

where the constants in capital letters are the concrete X and Y values that we have defined to match the visualization.

To illustrate the level of detail of the pixel-wise representation in `M10`: an airplane label at a specific time slot consists of 7000 pixels (175×40). With a zoom level of 45, there are 122 500 different pixels in total that could be mapped to airplanes while AMAN is running. Moreover, a `mouse_pos` variable tracks the pixel position of the mouse cursor as a pair of integers representing the x and y coordinates. We also add events for mouse movements and performing user interactions on UI elements. Listing 4.12 shows an example of the HOLD button.

```
1: event Move_Mouse_Hold extends Move_Mouse_Hold
2: any pos
3: where
4:   @pos_type pos ∈ POINTS
5:   @at_hold_button ui_element_at_point(pos) = hold_button_pos
6: then
7:   @mouse_pos mouse_pos := pos
8: end
```

Listing 4.12. Refined event for moving the mouse to the HOLD button (`Move_Mouse_Hold`) at M10; `Move_Mouse_Hold` is only executable when the mouse is moved to coordinates where the pixels are within the HOLD button.

The modeled pixel coordinates for the UI elements are identical to those in the VisB visualization (see Section 4.6.1). Consequently, the visualization aligns with the formal model. Thus, the proofs are meaningful because they accurately correspond to the visualization.

Despite the size of the pixel sets, the provers in Rodin handled them well in most cases. Where the provers could not deal with the set expressions directly, we used Rodin's rewrite rules to transform the set expressions to integer relational expressions on the x/y coordinates. This approach was possible because all pixel sets are defined using Cartesian products of intervals (and unions thereof). An example was shown in Equation (4.7).

Thus, our proof process based on set expressions was analogous to modeling bounding boxes for all UI elements. The advantage of the set representation is that we can use the standard set union/intersection operators, which are known to behave correctly, rather than having to implement our logic for checking whether bounding boxes overlap — a potential source of errors. Furthermore, the set representation can precisely represent non-rectangular UI elements.

Overall, the effort to develop and verify `M10` was particularly high, roughly equivalent to the combined effort of all preceding refinements. We have not yet finished modeling a few aspects of it — specifically, dragging airplanes is not yet implemented in the final refinement step.

A particular challenge was that during the verification of `M10`, we repeatedly discovered incorrect assumptions in the initial design of our model. For example, we initially assumed that the mouse's abstract location (i.e., the UI element under the mouse cursor) can only change if the ATCo explicitly moves the mouse. However, it can also change in various other situations — for example, an AMAN update may add a new airplane whose label appears under the mouse cursor, thus changing whether the mouse is over an airplane label, even though the cursor has not moved.

Fixing these incorrect assumptions required non-trivial changes to `M10` and even earlier refinement steps back to `M4`, again increasing the proving effort.

4.5. Verification

This section describes how we verify the AMAN model with proving and model checking. Furthermore, we also discuss where both verification techniques reach their limits.

Proving. Using Rodin, proof obligations (POs) are automatically generated from the model and can be discharged using various provers. The POs ensure the preservation of invariants, the absence of well-definedness errors, and the consistency between the refinement steps. Later in Section 4.7, we also use POs for validation.

Table 4.2 shows the number of POs in all refinement steps (including automatic, manual, and unproven POs). 606 out of 755 POs were proven automatically, while 149 POs had to be proven manually. All POs are discharged, giving us strong guarantees throughout the refinement chain regarding the invariants, well-definedness, and consistency between refinement steps.

Machine	Total	Automatic	Manual	Not Discharged
M0_ctx	0	0	0	0
M0	0	0	0	0
M1_ctx	3	3	0	0
M1	13	12	1	0
M2	4	4	0	0
M3	9	9	0	0
M4_ctx	0	0	0	0
M4	4	4	0	0
M5	0	0	0	0
M6	25	24	1	0
M7	10	10	0	0
M8	74	63	11	0
M9_ctx	0	0	0	0
M9	306	295	11	0
M10_ctx	54	17	37	0
M10	253	165	88	0
Total	755	606	149	0

Table 4.2.: PO statistics in Rodin from `M0` to `M10` with contexts; statistics include the total number of POs, the number of POs proven automatically/manually, and the number of POs not discharged.

Proving provides limited feedback when a PO is not discharged. In such cases, it is necessary to determine whether the underlying proposition is false or whether the prover needs manual support. We used ProB [153] with its animation, disproving, and model checking capabilities to discover errors and inspect counterexamples. We can then inspect concrete traces where, e.g., an invariant is violated.

After discharging the POs, we proceeded to the validation part (see Section 4.7).

Model Checking. We applied model checking to find errors that caused POs to fail. As explained in Section 4.4.2 for M1, we modeled time relative to the current time to keep the state space finite. Still, other aspects render exhaustive model checking intractable. We instantiated the constants for M0 to M9 with more restricted values (e.g., for the number of airplanes or the amount of zooming possible) to reduce the state space size and make exhaustive model checking feasible.[2] Note that `M10` only uses the full configuration for AMAN; therefore, we do not list `M10` in Table 4.3.

Table 4.3 shows the model checking results. The first configuration (*_inst_1) restricts the model to a single zoom level value of 15 (rather than allowing seven values from 15 to 45) and to only three different airplanes. In the second configuration (*_inst_2), we

[2]Note that even on infinite state spaces, model checking can be beneficial in detecting errors. We also applied ProB with the complete AMAN configuration during the verification and validation process to find potential errors when a PO is not discharged.

Machine	States	Transitions	Time [s]	Memory [MB]
M0_inst_1	9	66	0.35	160.315
M1_inst_1	1505	2 287 908	329.38	1425.511
M2_inst_1	9884	15 045 795	2032.62	8354.335
M3_inst_1 - M9_inst_1	-	-	> 3600.00	-
M0_inst_2	5	18	0.34	160.292
M1_inst_2	18	339	0.35	160.684
M2_inst_2	46	913	0.39	161.065
M3_inst_2	1953	49 154	2.66	188.211
M4_inst_2	1953	49 154	2.76	188.372
M5_inst_2	3905	102 210	4.4	212.439
M6_inst_2	9665	256 962	9.64	286.455
M7_inst_2	15 425	297 282	11.02	301.048
M8_inst_2	48 129	611 970	25.45	462.456
M9_inst_2	687 169	10 224 194	390.8	3996.342

Table 4.3.: Model checking statistics with ProB for two different configurations along the refinement chain with number of states, transitions, runtime (in seconds), and memory (in MB)

reduce the single zoom level to 5 and only two airplanes.

We use ProB to check all machines for invariant violations and deadlock freedom.[3] Furthermore, we activated ProB's operation reuse feature [150] together with state compression to increase the performance (`-p OPERATION_REUSE full -p COMPRESSION TRUE`). We run all experiments with ProB version 1.12.2, built with SICStus 4.7.1 (arm64-darwin-20.1.0) on a MacBook Pro (14", 2021) with an 8-core Apple M1 Pro processor and 16 GB of RAM. For the experiments, we set a timeout of one hour. The values in the table are the median from five runs.

As shown in Table 4.3, the state space rapidly grows for the first configuration. With ProB, we can determine variables that are assigned many distinct values, thereby significantly increasing the size of the state space. These variables are related to blocked time slots and airplanes in the landing sequence: `blockedTime` and `landing_sequence`.

In contrast, the second configuration allows efficient application of model checking. Here, we can model check all AMAN behaviors with the given configuration. However, as soon as the GUI events (clicking, dragging, and releasing) are split into multiple ones in M9, the state space also grows rapidly. Thus, model checking is also feasible to verify the AMAN model, but only for configurations that limit the state space. Consequently, model checking does not achieve full coverage, unlike proving. However, a significant advantage of Event-B is the availability of both proving and model checking: model checking for quickly finding errors early in the development, and proving to ensure many properties in general.

[3]Rodin does not generate POs for deadlock freedom.

4.6. Prototype for AMAN

This section presents the AMAN prototype that we created from the formal model augmented with domain-specific visualization in VisB and simulation in SimB.

VisB and SimB are used for two different tasks. With VisB, we implement the interactive components of the AMAN prototype, specifically the events on graphical elements performed by an ATCo. With SimB, we implement autonomous parts of the AMAN prototype, i.e., events that perform AMAN updates at a specific rate.

The prototype enables domain experts, stakeholders, and modelers to experiment with AMAN for validation purposes. Figure 4.1 shows the AMAN prototype created for `M6`.

4.6.1. Visualization

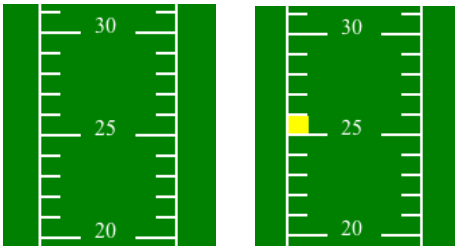
We created three VisB visualizations: a high-level version for `M6` where UI interaction is implicit, a lower-level version for `M9` with explicit UI interaction, i.e., with a mouse cursor and events for mouse-clicking and dragging, and for `M10`, a colored overlay on top of `M9` to visualize the meaning of mouse clicks at every possible pixel. The VisB visualization created for `M6` has also been adapted and used by Mammar and Leuschel for their model [159].

As the modeled system is an interactive GUI, these VisB visualizations are also virtual AMAN prototypes.

The visualizations present relative time, whereas the specification document displays absolute time. Assuming the current time is 9:03, our visualizations display 9:05 as 2, while Figure 6 in the specification [186] denotes 9:05 as 5.

Visualization at M6. Up to `M6`, we have modeled AMAN events and manual ATCo events. Yet, the ATCo events are not refined to mouse events.

On the left-hand side of Figure 4.1, one can see the airplanes in the landing sequence (as black arrows with a label) and the blocked time slots (in yellow). The user can block/unblock time slots by clicking on them on the left-hand side of the timeline.



(a) No Time Slots Blocked

(b) Time Slot 26 (`bl_26`) blocked after clicking on time slot

Figure 4.7.: Clicking on Time Slot 26 blocks the time slot (in yellow).

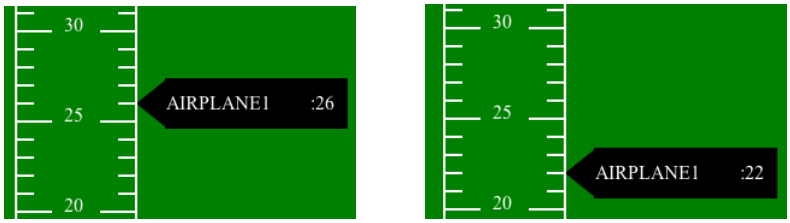
Figure 4.7 illustrates an example where the ATCo clicks on the (unblocked) time slot 26 in Figure 4.7a. As a result, this time slot is then blocked (shown in yellow in Figure 4.7b; corresponding to the description in Section 3.2 of the requirement document [186]).

Within the AMAN prototype, aircraft labels are visualized corresponding to:

Req17: aircraft labels must always be positioned in front of a small bar of the timeline;

Clicking on an airplane label in the VisB visualization executes `selectAirplane` with the selected airplane. One can then change the landing time or put the airplane on hold.

Figure 4.8 shows an example of moving an airplane. Initially, `AIRPLANE1` is scheduled to land in 26 minutes. After selecting the airplane, the user clicks on time slot 22. Following this, the airplane is scheduled to land in 22 minutes. In the prototype for `M9`, we refine the movement of an airplane by introducing ghost airplanes that appear as an intermediate step during dragging.



(a) Airplane scheduled at 26

(b) Airplane scheduled at 22 after selecting airplane and clicking on time slot 22

Figure 4.8.: Clicking on the airplane at 26 selects the airplane; clicking on time slot 22 then moves the airplane to this time slot.

In the AMAN prototype, the HOLD button is only visible when an airplane is selected. Figure 4.9 shows an example of putting an airplane on hold. Initially, an airplane is scheduled to land in 26 minutes (see Figure 4.9a). After the ATCo clicks on the airplane label, it is selected, and the HOLD button appears (see Figure 4.9b). When clicking the HOLD button afterward, the airplane is marked as HOLD (with a red frame, corresponding to Section 3.1 in the requirements document [186]).

The part of the landing sequence shown to the user depends on the *zoom level*, which can be changed by clicking on the zoom slider in the top right corner (see Figure 4.1). We model the zoom slider and the possible values corresponding to the following requirements:

- **Req18**: Lift of the zoom slider should always be located on the slider bar
- **Req19**: the value displayed next to the zoom slider must belong to the list of seven acceptable values for the zoom

Visualization at M9. Until `M9`, we have refined user events into multiple mouse events, i.e., events the ATCo can perform. We also introduce a *mouse* cursor in the `M9` visualization to fulfill:

Req21: there must be one and only one mouse cursor on the screen

(a) Airplane scheduled at 26

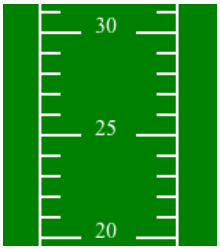
(b) HOLD button appears after selecting airplane at 26

(c) Airplane at 26 on HOLD after clicking hold button

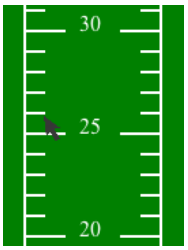
Figure 4.9.: Clicking on the airplane at 26 selects the airplane; clicking on the HOLD button then puts this airplane on hold.

amongst others. In `M9`, we refine each click event in `M6` into multiple events: (1) moving the mouse to the graphical object, clicking on the object (for some objects also dragging), and releasing the mouse click. As VisB only supports click events, the user must now click multiple times to execute an event in `M6`. While performing a user interaction, one can continue or abort the current event.

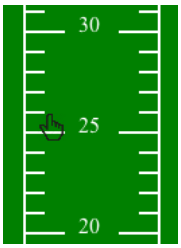
For example, within our `M9` prototype, the user has to click three times to block a time slot. Figure 4.10 shows an illustration where the ATCo performs each step by clicking on the time slot 26. Referring to the AMAN prototype for `M6`, Figure 4.10 can be seen as a refinement of Figure 4.7.



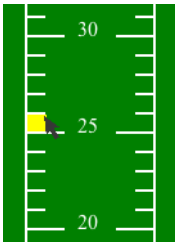
(a) No Time Slots Blocked



(b) Move Mouse to Time Slot 26



(c) Click Time Slot 26



(d) Release Clicking on Time Slot 26 blocks the time slot

Figure 4.10.: Blocking Time Slot 26 refined at M9; each mouse event (moving the mouse to a time slot, clicking on a time slot, releasing a click on a time slot) is performed by one click on the time slot with intermediate states in between.

We also implement this behavior for other graphical elements, such as the zoom, the HOLD button, and airplanes, fulfilling the following requirements, among others:

Req22: Hold(aircraft) function can only be triggered after a mouse press and a mouse released have been performed on the HOLD button.

Req23: Hold(aircraft) function must not be triggered if there is not a mouse press and a mouse released performed on the HOLD button.

To refine the dragging behavior, we introduced ghost airplanes that appear while the ATCo is dragging an airplane. This behavior corresponds to Figure 7 of the requirement document [186]. Figure 4.11 shows an example of a dragged airplane. This step is an additional one added between Figure 4.8a and Figure 4.8b.

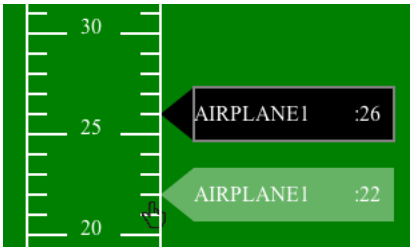


Figure 4.11.: Dragging an airplane from time slot 26 to 22; the dragged airplane is shown as a ghost.

Visualization at M10. As presented in Section 4.4, we modeled all UI elements with concrete pixel coordinates in `M10`. The `M10` VisB visualization adds a colored overlay that displays the meaning of each pixel position, i.e., whether there is an airplane, time slot, zoom slider, or HOLD button at that position. Figure 4.12 shows an example of the pixel overlay visualization.

The pixel overlay helps the modeler validate whether the formal pixel representation aligns with the visualization. In our first attempt, we tried to represent each pixel exactly. This implementation caused performance problems in VisB because it required creating 800 000 graphical objects (one for each pixel of a 1000 × 800 screen), each with its dynamic attributes that must be evaluated and updated. The current implementation of VisB struggles with such a large number of dynamic attributes.

To work around this issue, we reduced the resolution of the colored overlay to one-tenth along each axis. This implementation improves performance, although the overlay no longer precisely matches the coordinates. However, this approximate overlay was functional when developing `M10` to validate that the UI elements appear at the correct positions.

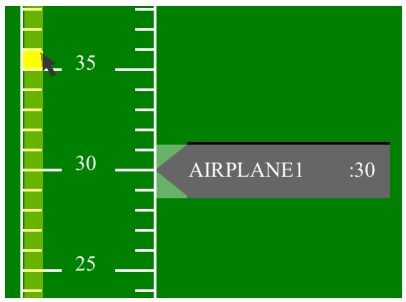


Figure 4.12.: Visualization for Pixel Overlay at M10.

Note that the pixel objects are used as an overlay only. There, we have not yet defined any listeners on the pixels to implement:

Req20: each movement of the mouse on the ATCo table must be reflected by a movement of the cursor on the screen

The development of `M10` also contributed to the visualizations for the previous refinement steps. After modeling the pixel coordinates of all UI elements in `M10`, we revised the `M9` visualization and used the coordinate and size constants in the `M10` model. We do this to ensure that the layout of the previous visualization aligns with `M10`.

4.6.2. Simulation

We created a real-time prototype for experimental purposes by combining simulation with domain-specific visualization. After demonstrating how VisB covers visualization, we now explain how SimB covers simulation.

As mentioned earlier, SimB is a simulator built on top of ProB. Interactive events can be triggered by clicking in VisB, while SimB automatically executes AMAN events. First, we describe how we simulate AMAN updates in the prototype from a high-level perspective. We then explain how SimB's concept for simulation works. Based on this, we present a detailed implementation of AMAN updates simulation with SimB's activation diagram.

High-Level Description of Simulation in Prototype. In particular, SimB simulates the execution of AMAN updates every 10 seconds. Because user interactions have higher priority, AMAN updates scheduled during ATCo interactions are blocked by the formal model. AMAN then schedules the next update to take 10 seconds.

For each triggered AMAN update, there is a probability of spawning a new airplane (and a complementary probability of not spawning an airplane). In the simulation of our prototype, the newly spawned airplane's time slot is uniformly chosen from all free time slots.

Another relevant aspect of the simulation is how we modeled time in the formal model. As we decided to model relative time instead of absolute time, the simulation must recognize when a minute has passed. The simulation then shifts all blocked time slots and all airplanes one minute further.

Furthermore, the simulation removes all *airplanes on hold* from the landing sequence in the next AMAN update. Note that the specification document only requires that airplanes in hold *eventually* disappear from the landing sequence and not necessarily in the next AMAN update. Alternatively, it would also be possible to simulate a probability or time for whether/when an airplane disappears from the landing sequence. However, there are no differences when experimenting with the prototype.

Description of SimB's Concept for Simulation. To implement this, we need a concept/technique that allows us to (1) encode timing behavior, (2) encode probabilistic behavior, and (3) combine timing and probabilistic behavior for real-time simulation. SimB supports those aspects with the underlying concept of *activation diagrams* [237].

Activation diagrams represent simulations synchronized with a formal model. Within SimB, there are two types of *activations* [237]:

- *Direct activations* to trigger an event in the formal model after a delay. Afterward, other activations are triggered.
- *Probabilistic choices* to choose between activations probabilistically.

One can also choose parameter values probabilistically.

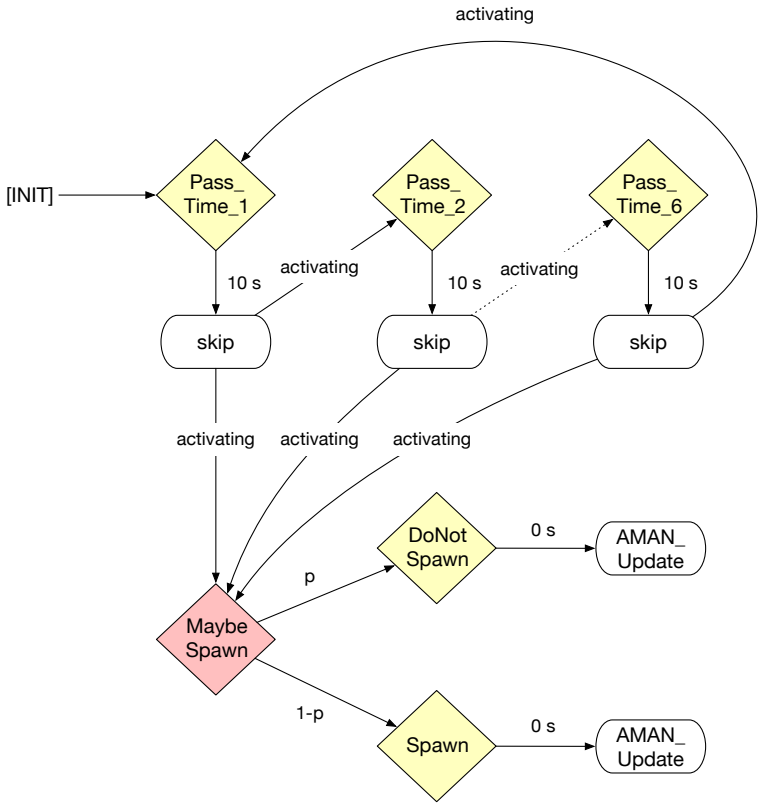


Figure 4.13.: SimB activation diagram for AMAN updates; `Pass_Time_*` is activated every 10 sec, triggering an AMAN Update; each AMAN update might spawn an airplane with a specific probability; the scheduled time slot is selected with a uniform distribution over all free time slots probabilistically.

Implementation of AMAN Simulation in SimB Figure 4.13 shows the activation diagram for the simulation of AMAN updates in the prototype. The yellow diamonds describe the direct activations, while the red diamond describes the probabilistic choice. The edges illustrate how activations trigger one another, along with timing and probabilistic behavior.

In the following, we describe the activation diagram in more detail. The simulation initializes the formal model with the entry point `[INIT]` and triggers `Pass_Time_*` every

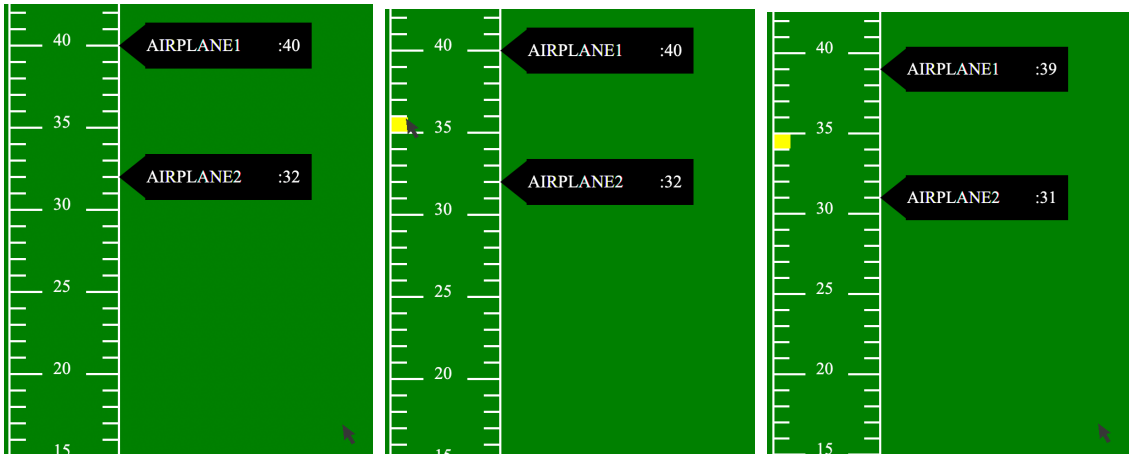
10 seconds. After `Pass_Time_6`, i.e., 60 seconds, the next AMAN update is triggered with `passed_minutes` increased by 1. Each time 10 seconds pass, the `AMAN_Update` event is triggered via the `MaybeSpawn` *choice activation*. There, we assign a specific probability `p` for not spawning another airplane (implemented by the `DoNotSpawn` *direct activation*), and a complementary probability `1-p` for spawning a new airplane at a random time slot, which is chosen uniformly from all free time slots (realized by the `Spawn` *direct activation*).

Note that we encoded fixed artificial values for the probabilities, i.e., we do not extract them from real air traffic control data. In the future, one could analyze whether/which other distributions (Poisson distribution, normal distribution, etc.) are more realistic for modeling airplane arrivals. Still, the prototype is usable for experimental purposes.

Referring to the formal model (see Section 4.4), an AMAN update also considers the number of minutes passed since the last AMAN update. This is why we encoded 6 different activations of `Pass_Time_*`. Here, we implement `Pass_Time_6` to increase the number of passed minutes (realized in SimB) by 1 regardless of whether SimB performs an AMAN update. The number of passed minutes is used as a parameter for the AMAN update and becomes relevant when SimB performs an AMAN update after a minute. Once SimB executes an AMAN update, SimB also resets that variable to 0.

As explained, we encode AMAN updates in SimB to remove all *airplanes on hold* from the landing sequence in the next AMAN update. We implement this behavior in the `Spawn` and `DoNotSpawn` activations.

Furthermore, AMAN updates are blocked while users interact with an airplane; we ensure this behavior by the guard 4.5. Once the user completes an interaction, AMAN updates are activated again.



(a) *System Event:* AMAN Schedules Two Airplanes (b) *User Interaction:* Block Time Slot (c) *System Event:* Minute Passes

Figure 4.14.: Example of AMAN prototype with user interaction via VisB and automatic simulation of AMAN autonomous events via SimB

In summary, the activation diagram adds timing and probabilistic behavior for simulation. SimB uses the activation diagram to execute events in the formal model directly. In

particular, we encode the activation diagram at the same level as the formal model. For example, the activation diagram encodes how airplanes appear/disappear in real-time by controlling the parameters for the AMAN update event. However, the activation diagram does not cover airplane capabilities or trajectory prediction, as these aspects are also not modeled in our Event-B model.

Figure 4.14 illustrates an example of user interaction and simulation in the AMAN prototype. Initially, SimB performs AMAN updates to schedule two airplanes in the landing sequence to arrive in 32 and 40 minutes. Afterward, the ATCo blocks the time slot 35. For the next AMAN update, where a minute has passed, both airplanes and the blocked time slots are shifted forward by one minute, considering that we have modeled relative time (and not absolute time).

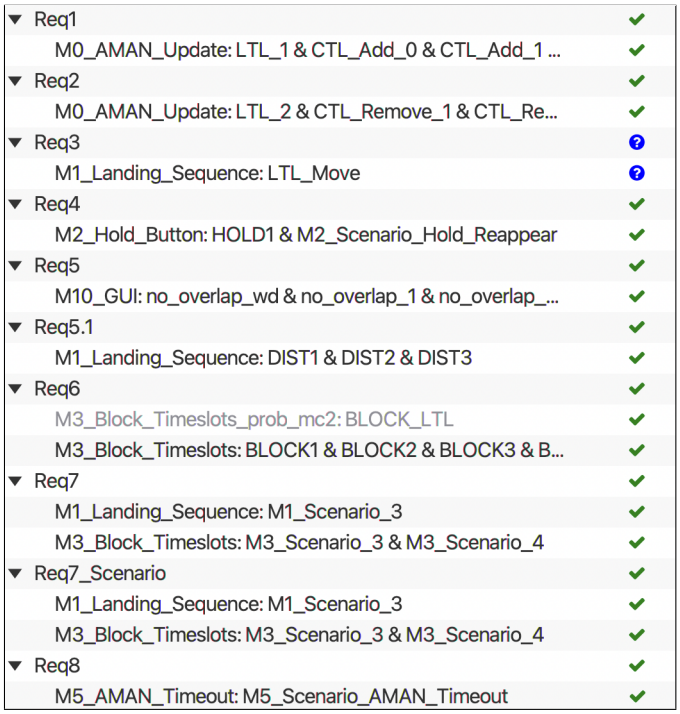


Figure 4.15.: Overview in ProB2-UI's VO manager showing the validated requirements and the underlying validation tasks that are combined to VOs.

4.7. Validation

In the following, we validate the AMAN model using validation obligations (VOs) [166, 217]. The creation and management of VOs are supported in the VO manager, which is part of ProB2-UI (partially shown in Figure 4.15). We also describe how the AMAN prototype helps validate specific GUI requirements.

As presented by Mashkoor et al. [166], VOs provide a systematic approach to structure the validation process. As defined in [218], a VO consists of: the requirement name, the

model (in our case, M0 to M10) to validate the requirement on, and the validation task to be performed to validate the requirement on the model (e.g., animation or LTL model checking). One can connect validation tasks in a VO using logical operators like $\wedge$ and $\vee$. Furthermore, one can pass the result of a validation task as an argument to another task. An example of a VO is:

$$\texttt{Req1/M1} : \texttt{TR}(\texttt{MC}(\texttt{GOAL}, \textit{some predicate}), [\texttt{op1}, \texttt{op2}])$$

This VO expresses that `Req1` shall be validated on model `M1` by running model checking to find a state satisfying the given predicate and then executing the trace $[\texttt{op1}, \texttt{op2}]$ starting from the found state.

Within the VO manager (see Figure 4.15), colored symbols indicate whether the VO is successful (green check mark), not evaluated (blue question mark), or failed (red x mark, not shown here). Furthermore, the VO manager enables systematic tracking of requirements throughout the modeling process and helps detect contradictions between requirements [166, 218].

We formulated validation tasks based on invariants, temporal properties, scenarios, projections, and coverage criteria for the AMAN requirements. The following outlines some detailed examples of VOs we developed to validate our AMAN model.

4.7.1. Invariant Properties

Invariant: Req5.1. Section 2.2 of the requirements document requires a minimum separation of 3 minutes between airplanes. We extract a requirement **Req5.1** from the document:

> **Req5.1**: [...] a landing separation of 3 minutes between aircraft is requested.

In Section 4.4, we model **Req5.1** as guards in the `AMAN_Update` and `Move_Aircraft` events. Furthermore, we formulated the invariant (4.1) in Section 4.4.2 to validate **Req5.1**. Rodin's PO generator generates three POs from this invariant (4.1). We define the validation tasks `DIST1`, `DIST2` and `DIST3` based on the corresponding proofs for these POs:

$$\texttt{DIST1} \triangleq \texttt{PO}(\text{Move\_Aircraft/inv13}, 2/\text{INV})$$
$$\texttt{DIST2} \triangleq \texttt{PO}(\text{INITIALISATION/inv13}, 2/\text{INV})$$
$$\texttt{DIST3} \triangleq \texttt{PO}(\text{AMAN\_Update/inv13}, 2/\text{INV})$$

We combine those POs into a VO:

$$\texttt{Req5.1/M1} : \texttt{DIST1} \wedge \texttt{DIST2} \wedge \texttt{DIST3}$$

Req5.1 is successfully validated on `M1` because the validation tasks `DIST1`, `DIST2`, and `DIST3` are completed with a successful result.

Invariant: Req5. At the GUI level, there is a requirement for airplane labels:

Req5: Aircraft labels should not overlap;

Req5 is handled in the final refinement `M10`, which models concrete pixel placements for all UI elements. `M10` introduces new invariants to ensure that the UI elements' pixels indeed do not overlap (see invariant (4.6)). Rodin generates POs for these new invariants, and based on the proofs for these POs, we define validation tasks and construct another VO from them to validate **Req5**:

$$\begin{aligned}\texttt{Req5/M10} : \; & \texttt{no\_overlap\_wd} \land \texttt{no\_overlap\_1} \land \ldots \land \texttt{no\_overlap\_6} \land \\ & \texttt{no\_overlap\_airplanes\_wd} \land \ldots \land \texttt{no\_overlap\_airplanes\_6} \land \\ & \texttt{no\_overlap\_block\_slots\_wd} \land \ldots\end{aligned}$$

In the AMAN prototype, we visually confirm that the airplane labels do not overlap. Note that this is not as strong as a formal proof. However, proof alone can also be insufficient, particularly if the visual representation and the underlying formal model do not correspond. We counteract this by building the visual representation of our AMAN prototype based on the formal model so that both the proofs and the visualization use the same constant definitions for pixel positions, sizes, etc. — see Section 4.6.1.

Invariant: Req6. The next requirement we validated is:

Req6: An aircraft label cannot be moved into a blocked time period;

To implement **Req6** in the formal model, we formulated the invariant (4.2) shown in Section 4.4.4. This invariant ensures that no airplanes are scheduled in a blocked time slot ($\texttt{ran}(\texttt{landing\_sequence}) \cap \texttt{blockedTime} = \varnothing$), unless the ATCo has blocked new time slots and AMAN has not yet updated the landing sequence accordingly ($\texttt{blockedTimesProcessed} = \texttt{TRUE}$). Based on this invariant, Rodin's PO generator generates five POs, and once again, the corresponding proofs are composed as validation tasks ($\texttt{BLOCK1}, \ldots, \texttt{BLOCK5}$) into a VO and assigned to the requirement:

$$\texttt{Req6/M3} : \texttt{BLOCK1} \land \texttt{BLOCK2} \land \texttt{BLOCK3} \land \texttt{BLOCK4} \land \texttt{BLOCK5}$$

However, the invariant (4.2) alone is too weak to ensure **Req6**. Especially when `blockedTimesProcessed` is equal to `FALSE`, the invariant does not ensure that the ATCo cannot move an airplane into a blocked time slot.

In the AMAN prototype, one can manually validate **Req6** by attempting to move an airplane label into a blocked time slot – which is rejected.

To illustrate the validation of **Req6** more in detail, we generated a state space projection diagram [136] based on the reduced instantiation `M6_inst_2` (see Section 4.5) with a predicate which checks that there is no airplane in a blocked time slot (see Figure 4.16). The diagram partitions the state space into those states satisfying

`landing_sequence` ∩ `blockedTime` = ∅ and those which do not, together with the variable `blockedTimesProcessed`. The corresponding VO is as follows:

$$\texttt{BEH1/M6\_inst\_2} : \texttt{VIS}(\texttt{MC}(\texttt{COV}), \texttt{PRJ}(\text{bool}(\text{ran}(\texttt{landing\_sequence}) \cap \texttt{blockedTime} = \varnothing) \mapsto \texttt{blockedTimesProcessed}))$$

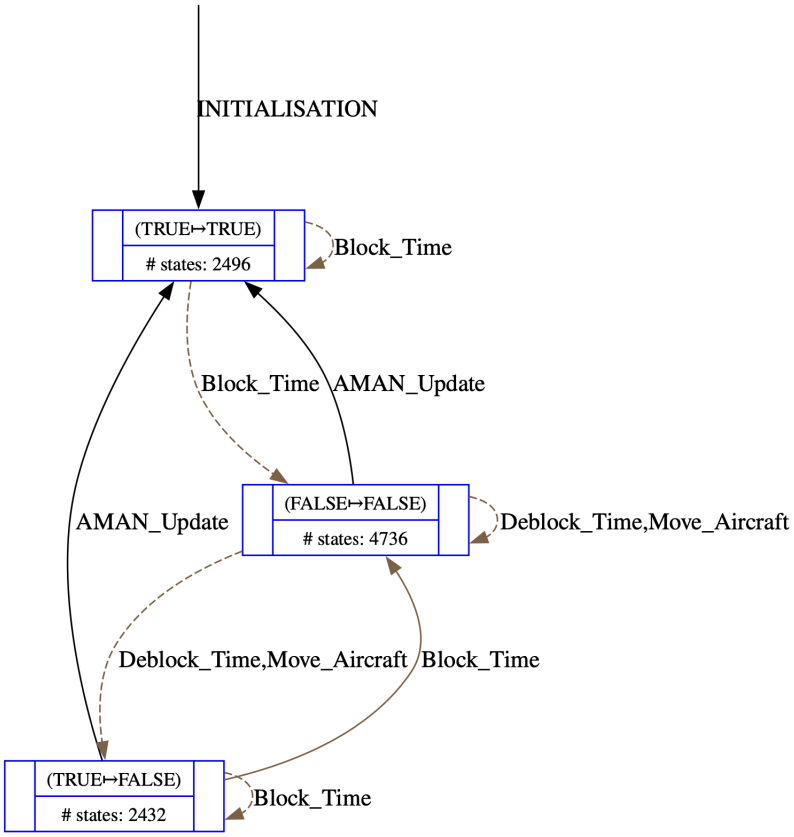


Figure 4.16.: Projection on (bool(ran(`landing_sequence`) ∩ `blockedTime` = ∅)) ↦ `blockedTimesProcessed`; solid arrow - there is a transition for every associated state in the original state space; dashed arrow - a transition exists for at least one state in the original state space.

`MC(COV)` is a validation task for applying model checking to cover the entire state space. `PRJ(expr)` is a validation task to create a state space projection on the expression `expr`. `VIS` is a validation task for inspecting a visualization, which the modeler or user must manually approve.

In combination, this means that one has to (1) run model checking to cover the entire state space (of the reduced instantiation), (2) create a projection diagram on the explored state space that focuses on blocked time slots with scheduled airplanes, and (3) inspect the resulting projection diagram.

The diagram shows that a user can block a time slot with a scheduled airplane (the dashed arrow `Block_Time` going from TRUE to FALSE). In contrast, moving an airplane into a blocked time slot is impossible (no arrow labeled `Move_Aircraft` from TRUE to FALSE). We also observe that `AMAN_Update` resolves all conflicts between blocked slots and airplanes (solid arrow from FALSE to TRUE).

Other Invariants. We also validated other requirements by expressing them as invariants in the formal model and following the same pattern by combining the resulting POs as a conjunction in one VO.

As another example, the formal model contains an invariant $\texttt{zoomLevel} \in \texttt{ZOOM\_LEVELS}$ with $\texttt{ZOOM\_LEVELS} = \{15, 20, 25, 30, 35, 40, 45\}$ to check:

> **Req16**: the zoom value cannot be bigger than 45 and smaller than 15;

The resulting VO is

$$\texttt{Req16/M4} : \texttt{ZOOM1} \wedge \texttt{ZOOM2}$$

whereas `ZOOM1` and `ZOOM2` are POs generated from the invariant.

4.7.2. Temporal Properties

Temporal Property: Req1. In the following, we discuss the requirements we validated using temporal model checking. One such requirement is:

> **Req1**: Planes can [be] added to the flight sequence e.g. planes arriving in a close range of the airport

First, we tried to validate this requirement with an LTL model checking task on `M0`:

$$\texttt{LTL}_1 := \texttt{LTL}(\texttt{GF}(\texttt{BA}(\{\texttt{scheduledAirplanes} \neq \texttt{scheduledAirplanes\$0}\})) \Rightarrow \texttt{GF}(\texttt{BA}(\{\exists x.(x \in \texttt{scheduledAirplanes} \wedge x \notin \texttt{scheduledAirplanes\$0})\})))$$

The `BA` operator is an extension to LTL supported by ProB, which allows a before-after predicate.

In this example, `scheduledAirplanes$0` and `scheduledAirplanes` denote the airplanes before and after executing an event. The LTL formula expresses that new airplanes are scheduled to the landing sequence infinitely often, under the fairness condition that the scheduled airplanes change infinitely often.

However, this does not fully cover the requirement. For example, the fairness condition excludes traces where the scheduled airplanes never change. It should be possible to add airplanes to the landing sequence, assuming the landing sequence is not fully occupied. Therefore, we apply the CTL model checking. $\texttt{CTL\_Add}_i$ checks that for all paths, there is always a next state where an airplane can be added to the landing sequence if it is not fully occupied.

$$\texttt{CTL\_Add}_i := \texttt{CTL}(\texttt{AG}(\{\texttt{card}(\texttt{scheduledAirplanes}) = i\} \Rightarrow \texttt{EX}\{\texttt{card}(\texttt{scheduledAirplanes}) > i\}))$$

$\forall i \in \{0, \ldots, n-1\}$ where n is the maximum number of airplanes in the landing sequence. The resulting VO on M0 is as follows:

$$\texttt{Req1/M0} : \texttt{LTL}_1 \wedge \texttt{CTL\_Add}_0 \wedge \ldots \wedge \texttt{CTL\_Add}_{n-1}$$

Analogously, we validated

> **Req2**: Planes can be removed from the flight sequence e.g. planes changing their landing airport for some reason

with the checking of a CTL formula. Here, we encountered the same problem with LTL model checking.

Both requirements **Req1** and **Req2** describe the *possibility* for AMAN updates to add or remove airplanes. Within the AMAN prototype, a user can experiment with different user scenarios while SimB simulates AMAN updates, adding or deleting airplanes. This allows a user to test/validate the interplay between user interactions and autonomous AMAN updates in the prototype, i.e., performing user actions while SimB performs AMAN updates.

Temporal Property: Scheduled Airplanes. Combining **Req1** and **Req2**, we formulated the following requirement for scheduled airplanes:

> **BEH1**: An AMAN update adds scheduled airplanes, which can only be removed by an AMAN update.

No other event and ATCo can schedule or remove an airplane. This behavior is satisfied for M0 because AMAN_Update is the only event that modifies the set of scheduled airplanes. More precisely, the Move_Aircraft event introduced at M1 refines skip and therefore does not modify the scheduled airplanes.

We validate that M6 and M9, where we created the prototypes, still fulfill this behavior. We create a projection on M6_inst_2 where we focus on airplanes in the landing sequence. There, we check that dom(landing_sequence) is only modified by the AMAN_Update event, resulting in the following VO:

$$\texttt{BEH1/M6\_inst\_2} : \texttt{VIS}(\texttt{MC}(\texttt{COV}), \texttt{PRJ}(\text{dom}(\texttt{landing\_sequence})))$$

Afterward, we confirm the desired behavior by inspecting Figure 4.17. That projection is also similar to the state space at M0, which gives us confidence that the desired behavior is still maintained. The validation of this behavior at M9 is done analogously.

The AMAN prototypes at M6 and M9 should also have no graphical element to add or remove an airplane. Theoretically, it could be possible to link a graphical element to the AMAN_Update event, which would be an incorrect implementation of the prototype in VisB, although the formal model is correct. A user can validate this while experimenting with the prototype.

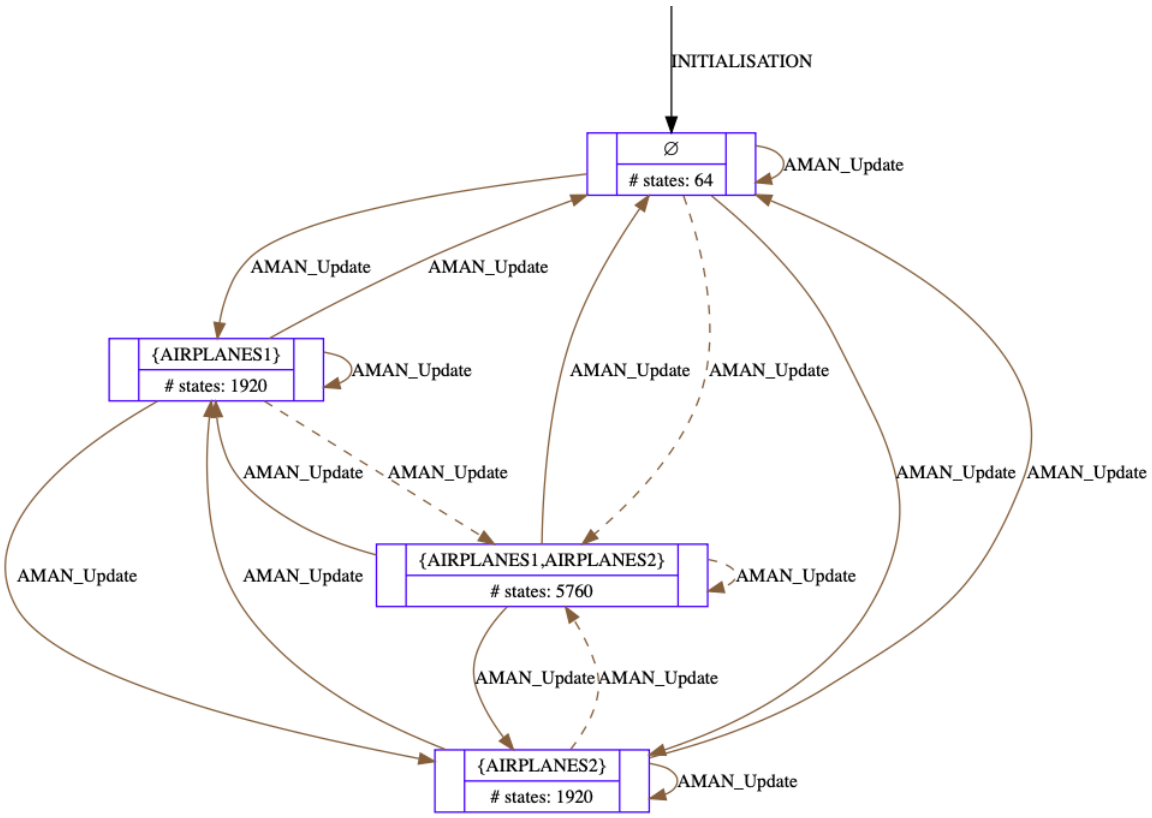


Figure 4.17.: Projection on scheduled airplanes (`dom(landing_sequence)`); solid arrow - there is a transition for every associated state in the original state space; dashed arrow - a transition exists for at least one state in the original state space.

Scenario: Req7. A scenario describes a sequence of events to validate specific behaviors. We create one or multiple traces to validate a scenario. The VO approach allows combining multiple trace replay tasks into a VO. Thus, VOs enable us to represent a complex scenario with different variations.

In the following, we formulate VOs to validate the requirement:

> **Req7**: Moving an aircraft label might not be accepted by AMAN if it would require a speed up of the aircraft beyond the capacity of the aircraft;

Our model does not implement airplane capabilities. However, we can still validate scenarios where AMAN overrides the time slots set by the ATCo while airplane capabilities are abstracted. Therefore, we formulate a scenario for **Req7**, i.e., a sequence of events in natural language:

1. AMAN schedules an airplane to land in a specific time slot.
2. The ATCo moves the airplane for landing to an earlier time slot.

3. AMAN detects that the airplane cannot land at the earlier time slot, thus processes the airplane again.

We validate the scenario with a VO which contains validation tasks to replay the two traces $\mathtt{T}_{\mathtt{m1.1}}$ and $\mathtt{T}_{\mathtt{m1.2}}$ on M1:

$$\mathtt{Req7/M1} : \mathtt{TR}(\mathtt{T}_{\mathtt{m1.1}}) \wedge \mathtt{TR}(\mathtt{T}_{\mathtt{m1.2}})$$

In the first trace $\mathtt{T}_{\mathtt{m1.1}}$, AMAN schedules the airplane to a later time slot, while in $\mathtt{T}_{\mathtt{m1.2}}$, AMAN removes the airplane from the landing sequence. In M3, we added blocked time slots as a feature. $\mathtt{T}_{\mathtt{m1.1}}$ is refined to $\mathtt{T}_{\mathtt{m3.1}}$ and $\mathtt{T}_{\mathtt{m3.2}}$, while $\mathtt{T}_{\mathtt{m1.2}}$ is refined to $\mathtt{T}_{\mathtt{m3.3}}$. This results in the following VOs with corresponding validation tasks:

$$\mathtt{Req7/M3} : \mathtt{TR}(\mathtt{T}_{\mathtt{m3.1}}) \wedge \mathtt{TR}(\mathtt{T}_{\mathtt{m3.2}}) \wedge \mathtt{TR}(\mathtt{T}_{\mathtt{m3.3}})$$

The refined traces cover different variations of the third step in the sequence:

1. AMAN schedules the airplane to a later time slot in the landing sequence. The time slot is the earliest that can be maintained based on the airplane's capabilities.

2. AMAN schedules the airplane to a later time slot in the landing sequence. The time slot is the earliest available after multiple blocked time slots that can still be maintained, given the airplane's capabilities.

3. AMAN removes the airplane from the landing sequence as all possible time slots (based on the airplane's capabilities) are blocked or do not fulfill the separation between airplanes.

Temporal Property: Airplanes on Hold. Following Section 3.1 of the requirements document [186], we extracted the following requirement for airplanes on hold:

> **BEH2**: The ATCo can always put any airplane on hold, and only an AMAN update can remove an airplane on hold from the landing sequence.

As a result, the airplane is no longer on hold and shall reappear later. Note that the set of airplanes on hold is a subset of airplanes in the landing sequence, formulated as an invariant and proved with POs.

To validate **BEH2**, we created a state space projection in M6_inst_2, focusing on airplanes on hold. With Figure 4.18, one can validate **BEH2** by running the following VO:

$$\mathtt{BEH2/M6\_inst\_2} : \mathtt{VIS}(\mathtt{MC}(\mathtt{COV}), \mathtt{PRJ}(\mathtt{held\_airplanes}))$$

Users can also validate **BEH2** while interacting with the AMAN prototype. For instance, one can put an airplane on hold and validate that an AMAN update eventually removes this airplane.

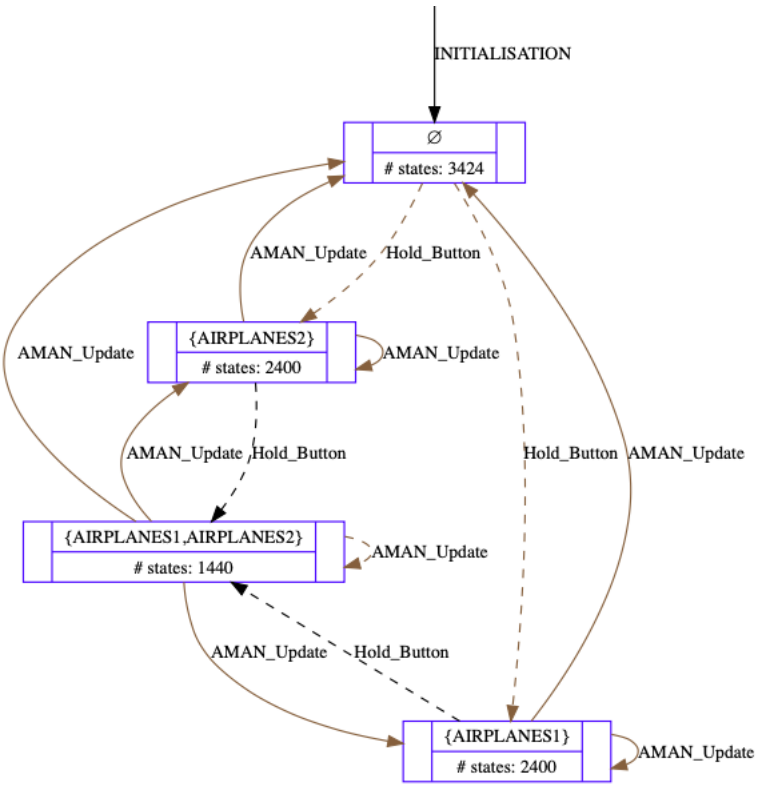


Figure 4.18.: Projection on airplanes on hold (`held_airplanes`); solid arrow - there is a transition for every associated state in the original state space; dashed arrow - a transition exists for at least one state in the original state space.

4.7.3. Other Properties

Coverage Criterion. In the following, we demonstrate how we evaluate the event coverage for a set of traces. An event is *covered* if the event is executed within any of the traces. This VO ensures that the validation activities are complete for the events executed. While validating the formal model, we uncovered parts of AMAN that we forgot to validate.

Event	Covered
AMAN_Update	yes
Move_Aircraft	yes
Hold_Button	no
Block_Time	yes
Deblock_Time	yes

Table 4.4.: Coverage Results from Scenarios for `M3` during the validation process

For example, multiple scenarios in `M3` are validated by replaying the traces $\mathtt{T}_{\mathtt{m3.1}}, \ldots, \mathtt{T}_{\mathtt{m3.4}}$. A VO to evaluate the coverage is:

$$\mathtt{Coverage/M3} : \mathtt{COV}(\mathtt{TR}(\mathtt{T}_{\mathtt{m3.1}}) \wedge \mathtt{TR}(\mathtt{T}_{\mathtt{m3.2}}) \wedge \mathtt{TR}(\mathtt{T}_{\mathtt{m3.3}}) \wedge \mathtt{TR}(\mathtt{T}_{\mathtt{m3.4}}))$$

Table 4.4 shows the results, which outline that `Hold_Button` is not covered, i.e., we never tested this feature. Consequently, we introduce a new VO that contains a new

trace, covering `Hold_Button`.

The coverage VOs were also used at further refinement levels, such as `M6`, to ensure that the user validation activities in the AMAN prototype are complete.

Interactive Requirements. The specification document includes some interactive requirements that refer to the appearance of graphical elements in the GUI. We validated them by animating traces through the prototype and inspecting the appearance of the graphical elements afterward. To systematically validate these requirements, we created VOs that contain trace replay tasks.

After performing a trace, the user has to confirm the appearance of the graphical elements in the prototype. Table 4.1 provides an overview of these requirements and the validation results. We could not validate **Req20** with the prototype because VisB currently supports only click events. In our prototype, we implement mouse movements as mouse clicks on specific positions.

4.7.4. Abstractions

The complexity of the formal model at the final refinements leads to the state space explosion. Therefore, we created an abstraction of the model to decrease the mental and computational load. Stock et al. [216] give a detailed presentation of this technique for the AMAN model.

An abstraction is created by removing elements hindering the deep investigation of a chosen phenomenon. The abstraction for this case study focuses on the user elements of `M0` through `M9`, i.e., the interactions the user can perform while being able to ignore a sizable amount of actions that the automatic part of AMAN can perform. The goal was to remove a sizable number of introduced states by considering the passing of time, i.e., the elements introduced in `M1`. One of our key efforts was to understand all of the implications that the variable of `time` has on user behavior.

Practical challenges for creating an abstraction were Rodin's currently missing tool support and technical limitations. Rodin struggles with quick, successive changes in the long refinement chains, as it has to rerun all of its evaluations.

4.7.5. Summary

In summary, we successfully validated most of the requirements in the specification document [186] with 39 VOs. We applied many validation techniques such as animation, trace replay, (temporal) model checking, state space projection, and evaluation of coverage criteria. The validation process for AMAN also involves informal validation tasks, such as validating behaviors through visualizations. An example is inspecting a state space projection used in a VO. However, some GUI requirements require inspecting the graphical elements in the AMAN prototype. We validated those requirements with traces where a domain expert or stakeholder must inspect the appearance of the graphical elements in the prototype afterward. Note that the total number of VOs depends on the stakeholders and domain experts. Unlike POs, VOs must be created manually based on the stakeholder's

requirements. Thus, the list of applied VOs is extensive and could be extended in future work if more requirements and relevant aspects for validation are taken into account.

4.8. Lessons Learned

This section presents the lessons learned from this case study.

Lesson 1: Validation obligations help us to systematically structure the requirements and validation tasks for formal models.

Using the VO manager integrated into ProB2-UI, we had a clear overview of which requirements still had to be modeled, which requirements had problems, and which validations were successful (see Figure 4.15). In particular, the VO manager also provided an excellent way to link the natural language requirements (the „what“ and possibly „why“) to validate tasks that a machine can execute (the „how“).

Lesson 2: With the current tooling, it is possible to develop pixel-precise formal models of graphical user interfaces, building a prototype based on the formal model that domain experts can validate.

With VisB and SimB, we could develop a prototype for a pixel-precise formal model. This prototype enables validation from domain experts' and users' perspectives. While we validated many requirements through formal validation techniques, prototyping allows us to experience and validate the interplay between user interaction and system actions. Furthermore, many requirements, especially interactive requirements and GUI requirements, require prototype inspection. In this case study, we further modeled the individual pixels of UI components and proved properties, such as the absence of overlaps.

Lesson 3: Validation obligations and prototyping help us – as non-experts – to reason about the requirements and ask questions to stakeholders and domain experts, uncovering ambiguities.

First, we present some issues that we uncovered while encoding requirements into VOs. One could have uncovered these issues without VOs. However, with VOs, we forced ourselves to explain the requirements in more detail. These are the issues that we have identified.

1. It was unclear to us modelers which part of the system the term AMAN refers to. In particular, it was unclear whether AMAN refers to the automatic scheduling part only or also to the GUI. This information is relevant for **Req8** (see Table 4.1); if there are no AMAN updates for 10 seconds, does the GUI stop working entirely, or does it continue operating in a „manual only“ mode without the autonomous part of AMAN? **Solution:** We assumed that when a timeout occurs, the UI still functions, but the ATCo shall not work with the AMAN system in this situation.

2. While formulating VOs to validate the interaction between the user and AMAN, we were unsure whether AMAN activities or user interactions have a higher priority, i.e., whether AMAN updates can occur while the user is performing an action or whether a user interaction blocks AMAN updates. First, we assumed that the AMAN overrules the user. However, after discussing it with the case study providers,

the updated requirements specification of AMAN states that user interactions have higher priority. Thus, user interactions can block AMAN updates, but not vice versa.

3. The requirements document used two terms: the *landing sequence* and the *arrival sequence*. When creating VOs, we discovered that we had not considered the arrival sequence. However, when creating VOs for the arrival sequence, we suspected both terminologies could refer to the same sequence. Following discussions with the case study providers, they agreed that both sequences are identical. This inconsistency was then removed in the updated requirements specification of AMAN.

Formalization often uncovers many issues. Here, the VO approach helps us to be more precise and formal, not just for verification but also for validation, hence uncovering more issues. We uncover the following issues not during the formulation of VOs but by running them:

1. The document must clarify what happens to airplanes on HOLD. Are they moved into a separate „HOLD sequence“ and still shown to the ATCo? Alternatively, do they disappear entirely from the AMAN GUI? And does the 3-minute separation between landing times also apply to airplanes on HOLD?

   Initially, we formulated a VO that checks that airplanes on hold are not in the landing sequence and that the 3-minute separation does not apply to them. As this VO failed, we thought we needed to adjust this behavior in the formal model so that it would pass. Furthermore, we thought that a requirement was missing to capture this behavior. Later, we discovered in Figure 6 of the specification that airplanes on HOLD still have an expected landing time.

   Solution: Thus, we assumed that airplanes on hold stay in the landing sequence — and that the 3-minute separation continues to apply to them — until AMAN explicitly removes them. Consequently, we removed the VO.

2. At an earlier development stage, we formulated an invariant that no airplanes are scheduled in a blocked time slot. We first concluded this invariant from the requirement that a user cannot move an airplane into a blocked time slot (see **Req6** in Table 4.1). We encoded a VO containing a conjunction of the generated POs from the invariant. After we could not prove the POs, we determined through model checking that this invariant, and thus the VO, is violated. The counter-example presents a scenario where the ATCo blocks a time slot for a currently scheduled airplane. First, we thought it should be impossible to block time slots where an airplane is scheduled. However, we discovered that this is possible when revising the requirements, particularly Section 3.2 of the specification [186]. The specification also states that AMAN must process those airplanes in the next step. This information also led us to improve the AMAN Update event in `M3`. We presented a detailed discussion in the description of `M3` in Section 4.4 and in the validation of **Req6** in Section 4.7.1.

Furthermore, we uncovered the following issues while interacting with the prototype guided by the VOs:

1. Figure 6 in the specification [186] shows an airplane on HOLD at 31 minutes, but the zoom level is 30, so the GUI should only show airplanes up to 30 minutes away. Is this an error in the example figure, or does this mean airplanes on HOLD are excluded from the zoom constraints? **Solution:** We assumed that airplanes outside the zoom are only relevant for the landing sequence, but nothing else. We later discovered that Figure 6 in the specification displays absolute time, not relative time.

2. Initially, we assumed that Figure 6 in the case study specification shows the minutes relative to the current time. With the AMAN prototype, we observed that our visualization displays relative time, while Figure 6 in the specification [186] displays absolute time. The trigger here was that the landing sequence in our prototype starts at 1, while in the specification document, it starts at 2. Furthermore, Figure 6 shows an airplane scheduled for landing at minute 31, while the zoom is 30.

4.9. Related Work

Verifying and validating human-machine interfaces has been an ongoing research topic for many years.

Dix [63] notes that while formal notations help tackle the development of interfaces, deploying formal methods is often time-consuming as they require a high degree of expertise. Dix further notes that a graphical representation is helpful to verify/validate a desired outcome.

In this work, we apply formal modeling techniques and tools to validate and verify human-machine interfaces (HMI), although these techniques and tools were not specifically designed for HMI. Referring to Dix [63], we experienced that developing a prototype with visualization was particularly useful and enriched the validation process. However, formal verification and validation are still crucial as they lead to stronger guarantees regarding the verified/validated properties compared to inspecting the visualization alone.

PVSio-web [241] is a toolkit for creating human-machine interfaces for PVS [184] models. PVS was initially developed for conducting formal proofs. There is also a co-simulation framework for PVS [187], which combines visualization and simulation into the human-machine interface. In our work, we create an AMAN model with Rodin and verify it using the Rodin provers. Our work combines visualization via VisB and simulation via SimB to create an AMAN prototype with interactive and autonomous aspects.

Masci and Muñoz created a prototype for a Detect and Avoid (DAA) system [163], another type of software from the aviation domain. Masci and Muñoz's DAA toolkit inherits the architecture linking interactive prototypes to executable formal specifications from PVSio-web. As presented there, the DAA toolkit also supports 3D simulation. In

this work, we created AMAN prototypes as VisB visualizations; yet VisB is not capable of 3D visualization.

CIRCUS[4] is a development environment specifically designed for interactive systems. An essential feature of CIRCUS is the support of HAMSTERS diagrams, as shown by Campos et al. [44]. The AMAN requirements document [186] presents many tasks with HAMSTERS diagrams. ProB and Rodin do not support HAMSTERS diagrams; instead, our work uses the HAMSTERS diagrams from the AMAN specification document to guide the development of our Event-B model, particularly structuring the Event-B machine hierarchy.

Another work exploits Norman's action theory in the context of CIRCUS to evaluate the impact of formal modeling tools on engineering activities [72]. In our work, we did not evaluate the impact of formal methods on the activities of engineers or users. Instead, we created a prototype based on the formal model for users to experiment with. Additionally, we focused on the development process driven by VOs, where formal validation and requirements engineering take center stage and impact each other.

Using VisB as a Model-View-Controller (MVC) pattern is related to other works that combine human-machine interfaces and MVC to create a prototype [210, 89]. Another work is **formal MVC** [31], which is a pattern to develop GUIs for ASMs using the Asmeta toolset [84]. Bombarda et al. [31] also used formal MVC for the AMAN case study.

Some tools and frameworks already address safety-critical interactive systems with formal methods [211, 23, 170, 178]. Those tools mainly deal with formalisms explicitly designed for interactive systems. Our work presents a formal model for AMAN in Event-B, a state-based formal method. Thus, one challenge was the creation of a formal model based on HAMSTERS diagrams, which are designed specifically for interactive systems. Here, we used HAMSTERS diagrams as requirements to create the formal Event-B model for the AMAN system rather than analyzing them directly with a formal method tool. A similar approach is also followed by Bombarda et al. for ASMs [31]. Cunha et al. [56] follow a more high-level approach for Alloy: in that work, task models are generally formalized in Alloy to encode task models for AMAN.

Another work by Navarre et al. [178] links task models and system models via scenarios. We modeled the AMAN system and high-level user events in the initial refinements. Later, we refine user events to mouse events and a pixel-by-pixel GUI.

Another HMI case study is the Control Panel Interface, which was modeled and validated by Campos and Harrison [45]. Campos and Harrison [45] modeled the system with MAL (Modal Action Logic) and validated it with CTL. That case study also considers a state- and action-based representation for validation purposes.

As an alternative to state-based formal methods, some approaches use the dataflow language Lustre. For example, d'Ausbourg uses Lustre to create user interfaces [57]. Another approach uses Lustre as an intermediate language to translate from LIDL (a language designed for interactive systems) to HLL for analysis [85].

[4] `https://www.irit.fr/recherches/ICS/softwares/circus/`

Overture [55] is a formal method toolbox for VDM[5] [30] models and offers a variety of techniques to verify and validate VDM models. A notable feature for modelers aiming to validate user interfaces and interactive environments was shown by Oda et al. [181]. Oda et al. model a GUI directly linked to an underlying functional model. Another feature is the ability of Overture to animate 3D objects, as shown by Thule et al. [223]. Furthermore, VDM tools include a code generator to generate a GUI from VDM++ specifications [180]. VDM models can be proven with PVS [10]. We use Rodin and its provers to verify our AMAN model. Using VisB, one can either create a formal B model as a GUI directly or define a GUI to link it with a functional model. However, the sophisticated possibilities of 3D models are still lacking, but they are intriguing as such models allow for a more immersive and, therefore, thorough validation experience.

In conclusion, formal methods are well-considered when creating safety-critical human-machine interfaces. While many tools are designed specifically for human-machine interfaces, our work applies existing techniques such as SimB and VisB to refit Event-B for validating human-machine interfaces. Thus, our formal method tools and techniques are flexible to enable high-quality reasoning with reasonable effort for human-machine interfaces. In particular, we combine techniques such as verification by proof, validation by animation, domain-specific visualization, and simulation to create and analyze a human-machine interface.

4.10. Conclusion and Future Work

This article presented a formal Event-B model of an air traffic control system developed using Rodin. The challenge of the case study was to model a GUI with user interactions, to model the autonomous part with timing behavior, and to coordinate the interactive part with the autonomous part.

We developed the first refinement steps guided by task models in HAMSTERS notation, which consist of autonomous events and interactive events formulated at a high level. We later refined the formal model, particularly the GUI parts, to a low-level GUI with mouse events and pixel-level detail. In this work, we successfully derived a real-time prototype for AMAN from the formal model, proved the formal model even at the GUI level, and validated the formal model and the prototype with VOs.

We create the AMAN prototype as an interactive GUI in ProB via VisB and real-time simulation via SimB. VisB and SimB cover two distinct parts in the prototype: VisB implements interactive components performed by the ATCo. This means that users can apply events by clicking on graphical elements within the GUI of the prototype. In the future, one could extend VisB with more mouse events, such as *mouse button press* and *mouse button release* to handle GUI behavior more accurately (currently VisB only supports simple mouse clicks and hovers). SimB controls the autonomous parts of the AMAN system.

With the introduction of complex GUI behavior in the formal model, discharging the POs became increasingly challenging. Although model checking struggles with the state

[5]VDM has several dialects that we do not list explicitly every time.

space explosion problem, we could still apply model checking to the formal model on instantiated configurations.

Our experience during the validation of AMAN was that validation obligations are particularly useful in structuring the validation process and linking validations to requirements. In the process of formulating VOs, the validation efforts to ensure that each requirement is fulfilled are systematically written down. If a VO fails, this feedback can help uncover inconsistencies and ambiguities in the requirements.

5. Validation of Reinforcement Learning Agents and Safety Shields with ProB

Abstract. Reinforcement learning (RL) is an important machine learning technique to train agents that make decisions autonomously. For safety-critical applications, however, the decision-making of an RL agent may not be intelligible to humans and thus difficult to validate, verify and certify.

This work presents a technique to link a concrete RL agent with a high-level formal B model of the safety shield and the environment. This allows us to run the RL agent in the formal method tool ProB, and particularly use the formal model to surround the agent with a safety shield at runtime. This paper also presents a methodology to validate the behavior of RL agents and respective safety shields with formal methods techniques, including trace replay, simulation, and statistical validation. The validation process is supported by domain-specific visualizations to ease human validation. Finally, we demonstrate the approach for a highway simulation.

Keywords. AI, Reinforcement Learning, B Method, Validation, Shielding

Funding. The work of Fabian Vu is part of the KI-LOK project funded by the "Bundesministerium für Wirtschaft und Energie"; grant # 19/21007E, and the IVOIRE project funded by "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF) grant # I 4744-N.

5.1. Introduction and Motivation

Artificial intelligence (AI) and machine learning (ML) [248] are increasingly used to develop software applications. One popular ML technique is reinforcement learning (RL) [219] which also finds use in safety-critical domains such as the automotive domain [203], the railway domain [190], and the aviation domain [197]. Hereby, an *agent* learns to make autonomous decisions within an *environment* to maximize an accumulated *reward*. In a trial-and-error approach, the agent receives a reward as feedback for actions taken based on their observed outcome and uses this feedback to optimize its *decision policy*.

In the context of safety-critical applications, it is important to *verify* and *validate* an RL agent's learned behavior. As RL agents typically are black boxes, their decision-making

may be unintelligible and hard to reason about. Validation and verification of RL agents is thus an ongoing research topic [224, 79, 140]. *Safety shields* [12] is a runtime monitoring and verification technique to ensure the safety of RL agents. A safety shield intervenes when a dangerous situation might occur, i.e., its task is to avoid or prevent dangerous situations. Safety shields are related to Sha's concept of "using simplicity to control complexity" [208] where a simpler system monitors and intervenes in a complex system when rules are violated.

This work presents a technique to link a concrete RL agent with a high-level formal model of the B method [4] for the RL agent and its environment with a safety shield. This allows us to run the RL agent in the ProB [152, 153] animator and model checker, and use the formal model as a safety shield at runtime. While ProB also supports verification of the formal model via model checking, we focus on the validation of RL agents with other formal methods techniques such as trace replay, simulation, and statistical validation. With trace replay, it is possible to re-play a single execution run to reason about the RL agent's decisions. Trace replay also checks whether an execution run is feasible; thus, one can validate whether a safety shield has out-ruled a dangerous situation. Using SimB [237], one can run the RL agent in ProB in real-time, or as Monte Carlo simulation. Based on multiple simulated runs, one can apply statistical validation such as computing the likelihood of violating certain properties, and estimating probabilities, averages, and sums. Finally, we demonstrate the applicability and efficacy of this methodology in a highway environment [147]. In this context, we evaluate how safety shields in this work improve the safety and the achieved reward for the RL agent. We also use the insights gained from this technique, to improve the safety shield and the reward function.

5.2. Background

The B method.

The B method [4] is a formal method for specifying and verifying software systems. The B language is based on set theory and first-order logic, and makes use of *general substitution* for state modifications as well as *refinement calculus* to model *state machines* at various levels of abstraction.

Within a B model, the modeler has to specify an `INVARIANT` clause which contains a predicate to provide typing for variables and define (safety) properties which must be fulfilled in each state of the model. The `INITIALISATION` contains substitutions (also called statements) to describe the model's initial states, assigning values to each machine variable. Within the

```
1: MACHINE CoinToss
2: SETS Side = {Heads, Tails, None}
3: VARIABLES lastToss
4: INVARIANT lastToss ∈ Side
5: INITIALISATION lastToss := None
6: OPERATIONS
7:   toss = lastToss :∈ {Heads, Tails}
8: END
```

Listing 5.1. B Model for Coin Toss

`OPERATIONS` clause, a modeler can specify operations with respective *guards* and substitutions. When the guard is true, the operation's substitution can be executed by modifying the model's current state. Listing 5.1 shows a simple B model for a coin toss with an operation `toss` that chooses between `Heads` and `Tails` non-deterministically.

In this paper, we use established tools from the B landscape, namely ProB and SimB. ProB [152, 153] is an animator, constraint solver, and model checker for formalisms such as B, Event-B, TLA$^+$ or CSP. It provides capabilities such as animation, trace replay [25], simulation [237], and different model checking techniques [128, 194] to *verify* and *validate* formal models. SimB [237, 236] is a simulator with support for timing, probabilities, and live user interaction. SimB also provides statistical validation techniques such as hypothesis testing and value estimation for probabilities, averages, and sums.

Reinforcement Learning.

Reinforcement Learning [219] is a machine learning paradigm in which an agent learns to maximize a cumulative reward function via a feedback loop with its *environment* in a trial-and-error manner. The agent interacts with its environment through a set of available actions which can alter the environment's state. The respective actions are chosen via a gradually learned *policy* which dictates the agent's decision-making process. The benefits of actions are quantified by a *reward* function evaluated in the successor states. By estimating the *value* (i.e. predicted long-term reward) of actions, instead of only the immediate reward of the next state, a policy can make short-term trade-offs which lead to higher long-term rewards.

In this work, we use the Deep Q-Network (DQN) algorithm [175] which mixes deep learning with Q learning [240]. Given a state-action pair (s, a), the idea behind Q learning revolves around learning an action-value function $Q(s, a)$ that estimates the long-term value of executing action a in state s [219]. In the DQN algorithm, the learning of the Q function is done by a deep neural network [175].

Safety shields.

Safety shields [12] is a formal technique to ensure the safety of an agent at runtime. More precisely, the agent is surrounded by a *safety shield* which intervenes to prevent/avoid dangerous actions. Safety shields align with Sha's concept of "using simplicity to control complexity" [208] where a simpler system monitors and enforces properties/rules in a complex system. Two techniques are pre-shielding and post-shielding [125]. In the pre-shielding approach, actions are shielded before execution and then provided to an RL agent to choose the next action from. In post-shielding, actions are corrected to safe ones when the agent's decisions are considered unsafe. In shield synthesis [125] the safety shield is synthesized via training from the underlying environment and RL agent.

The Highway Environment.

The highway environment [147] is an available environment for training RL agents to navigate a particular vehicle on a highway. We refer to that vehicle as the *ego vehicle*.

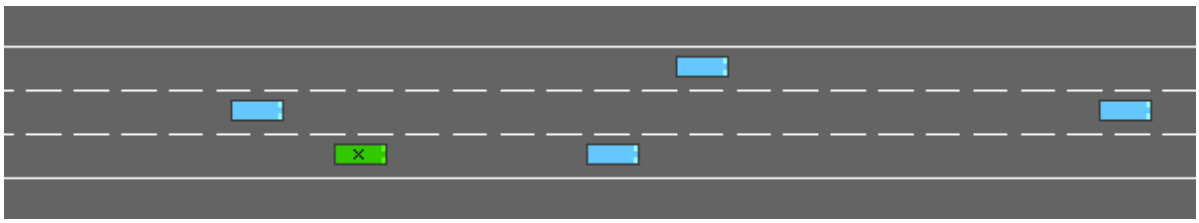


Figure 5.1.: Screenshot from the highway environment. The green box (manually marked with an X, on the bottom line to the left) represents the ego vehicle controlled by the RL agent, while the blue boxes are surrounding vehicles.

The observed environment contains positional information (x/y-coordinate) and velocities (in x/y direction) of all vehicles. There is information about whether the ego vehicle has crashed, and the reward resulting from the current state. Hereby, the reward function favors driving fast and on the right-most lane. The environment is simulated in a frequency of one frame per second, following the default configuration. Hence, each second the RL agent observes the current state and reacts accordingly. As the goal is to learn a policy which lets the ego vehicle drive fast and collision-free, the agent has to learn when to accelerate or decelerate, and when to switch lanes to keep momentum. The agent's action space consists of 5 actions: `IDLE`, `LANE_LEFT`, `LANE_RIGHT`, `FASTER`, and `SLOWER`. Figure 5.1 displays a visualization of an exemplary environment state.

5.3. Formal Models for Reinforcement Learning Agents

This section presents a technique to use formal models for the validation of RL agents. Based on a trained RL agent, a modeler creates a formal model which captures the RL agent's actions/decisions, and the environment's state. In this work, we do not formally model the internal decision-making process of the RL agent. This means that the decisions are still made by the RL agent, while its decision and the resulting environment's state are synchronized with the formal model. Adding the agent's actions as machine operations in the formal model, we can also define rules in the formal model to use it as a safety shield (discussed in detail in Section 5.3.2).

Figure 5.2 illustrates the interaction between the formal model and the RL agent with Shielding. During the RL agent's runtime, there is a sensor capturing the RL agent's environment. The environment is updated in the formal model and the RL agent accordingly. At runtime, the formal model is used to compute the set of actions that are considered to be safe, which is then passed to the RL agent. From these safe actions, the RL agent then chooses the one with the highest estimated long-term reward. This action is then executed in the environment.

Using a formal model at runtime gives us the ability to apply formal method tools and techniques to the RL agent. This enables us to evaluate and uncover weaknesses in the reward function and (possible) safety shields. Furthermore, this work is not limited to RL, but caters to other AI and real-time systems.

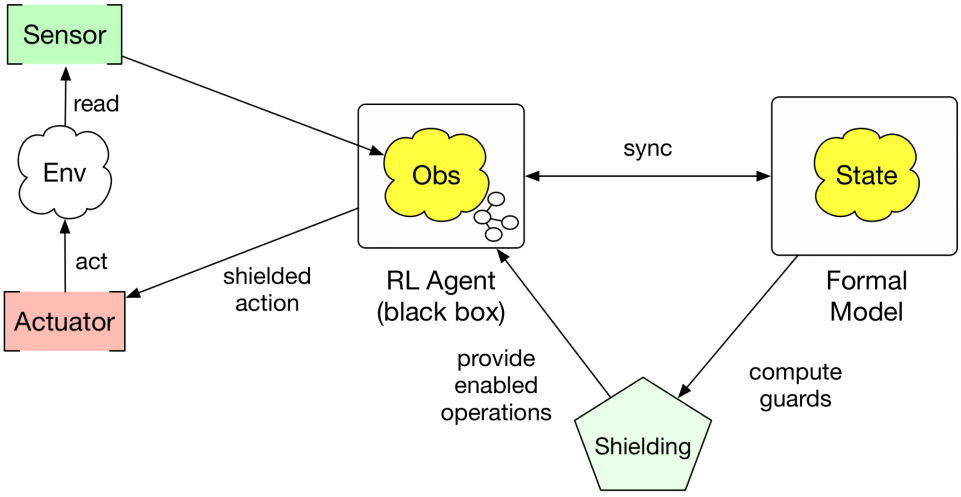


Figure 5.2.: Interaction between Formal Model, RL Agent with Shielding, and the Environment; shielding process works similarly to pre-shielding [125].

5.3.1. Creation of the Formal Model

The formal B model contains (1) the current state of the environment, and (2) the agent's actions. The environment's current state is represented using *sets*, *constants*, and *variables* in the formal model.

Let us assume that the RL agent can execute the actions $a_1, \ldots, a_m$. For each action a_j with $j \in \{1, \ldots, m\}$, we introduce a respective operation o_j which consists of a guard g_{o_j} and a state-altering substitution s_{o_j}:

$$o_j = \texttt{PRE}\ g_{o_j}\ \texttt{THEN}\ s_{o_j}\ \texttt{END}$$

Each operation's guard g_{o_j} defines whether the operation is considered safe for execution; we use this to encode a safety shield. The guards must hence be encoded in such a way that at least one operation is always enabled. Otherwise, the agent runs at risk of being unable to act at all in certain cases, as it will only be able to execute actions with their corresponding guards enabled. This property can be checked by techniques that are made available in this work.[1] Within the substitution s_{o_j}, the variables are assigned to a possible value wrt. their expected domain. The `INITIALISATION` substitution is encoded similarly. With this encoding, it is also possible to validate the implementation of the RL agent. Let us assume that v_i is a variable whose value changes after executing an action o_j. Within s_{o_j}, one could then encode:

- an assignment by value val ($v_i := val$),
- a non-deterministic assignment via a domain set S ($v_i :\in S$), or
- a non-deterministic assignment via a domain predicate P ($v_i :| (P)$).

[1] Cf. relative deadlock freedom [5, Chapter 14] for a proof-based approach.

Let us assume that we create a formal B model for the highway environment in Section 5.2 using a variable `velocity`. Assume we would like to encode a `FASTER` operation with the following conditions: (1) `FASTER` shall only be executable if the `velocity` is less than or equal to 30 m/s, and (2) the `velocity` is expected to increase when executing `FASTER`. We could then encode this by an operation:

$$\texttt{FASTER} = \texttt{PRE velocity} \leq 30 \\ \texttt{THEN velocity} :| (\texttt{velocity} > \texttt{velocity}') \texttt{ END}$$

Remark: $\texttt{velocity}'$ refers to the previous state; thus, $\texttt{velocity} > \texttt{velocity}'$ means that the speed increases after the action has been executed.

5.3.2. Implementing a Safety Shield around the RL Agent

Referring to Figure 5.2, we implemented the synchronization and communication (including shielding) between the formal model and the RL agent in ProB and SimB. The simulation is done by the RL agent and synchronized with the simulation in ProB and SimB. As mentioned before, the formal B model encodes safety shields in the operations' guards to apply pre-shielding [125]. The decision process with shielding is illustrated in Figure 5.3. For each executed action, the following steps are performed:

1. The current state of the environment and the last executed action is captured by the RL agent, and provided to ProB.
2. ProB synchronizes the internal state of the animated formal model to match the current observation provided by the environment. Based on the encoding of the operations (discussed in Section 5.3.1), ProB also checks that the target state matches the desired effect of the provided action.
3. Based on the current state, ProB computes enabled operations by evaluating their guards. Actions where the guard is violated in the current state are deemed unsafe.
4. ProB provides a list of enabled operations to the RL agent.[2]
5. Based on the current observation, the RL agent predicts the enabled operation/action with the highest reward.
6. The chosen action is subsequently executed by an actuator.
7. The environment changes according to the action and the respective reward is computed.

Referring to the highway environment in Section 5.2, an example of the shielding process in Figure 5.3 could be as follows: First, the RL agent observes the environment containing other vehicles and provides the information to ProB. Second, ProB computes

[2]Note that at least one operation must always be enabled as discussed before.

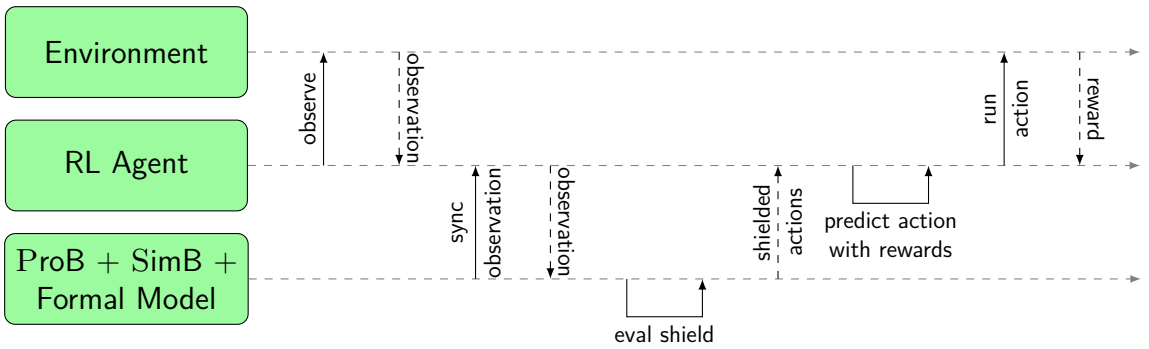


Figure 5.3.: Shielding the RL Agent with a Formal Model. The general control loop captures the current environmental state over which a set of enabled actions are computed by ProB, matching the safety shield's specification. The RL agent chooses the enabled action which has the highest reward for execution.

`SLOWER` and `LANE_LEFT` as operations that are considered to be safe and provides the shielded actions to the RL agent. Finally, the RL agent executes the enabled action with the highest reward which can be `SLOWER`, for example. The environment updates accordingly, and the reward is returned to the RL agent. Without a safety shield, the RL agent could predict `FASTER` with the highest reward and execute the action although it could be evaluated as unsafe.

Section 5.4 shows that manually encoded safety shields can improve the RL performance over unshielded agents. However, we do not promote manually encoded shields but rather demonstrate the possibility of using a formal specification as a shield. In cases where this is not suitable, we recommend the synthesis of safety shields based on safety guarantees instead [125].

5.3.3. Validatability and Verifiability

This work facilitates simulating and reasoning about the RL agents' execution runs, despite their black-box nature. Based on a single execution run, one can evaluate the behavior with trace replay. If the agent behaves correctly in a critical situation, we can understand which decisions were particularly important. We can also assess errors leading to a safety-critical situation. With trace replay, one can evaluate which dangerous scenarios are avoided by safety shields. If the execution of an operation in a trace is blocked by a safety shield, the safety shield was effective in avoiding this particular dangerous scenario.

Given multiple execution runs, one can apply statistical validation techniques, e.g., estimation of certain values (probabilities, averages, and sums) and the likelihood of certain properties. This allows us to validate the choice of the reward function as well as the behavior and impact of safety shields.

By encoding expected domains for the variables' values after executing an operation, this work allows us to validate the implementation of the RL agent. We can also validate that the RL agent and its environment match the encoded domains. Consequently, the

formal model together with the encoded safety shield can be seen as an over-approximation of the RL agent and its environment. In the future, we intend to use those validated domains as assumptions to (1) prove the formal model under these assumptions, and (2) also restrict the state space to make model checking easier to apply. With these techniques, one could then check safety properties (including invariants) on the formal model. When the formal model, and thus also the safety shield fulfill the safety property, one can conclude that the safety property is enforced for the RL agent. As the formal model works as an over-approximation, this does not apply to liveness properties.

5.4. Case Study

We applied this work's methodology to various case studies which are available online[3]. In this section, we focus on using this technique to validate a highway environment RL agent [147]. First, we present the formal B model. We then describe how we train the agent, and how we apply SimB's simulation and statistical validation. We then apply trace replay, and domain-specific visualization to reason about the agent's decisions.

Table 5.1.: Encoding of Shield for Highway Agent

Action	Disabling Condition (Guard)
`LANE_LEFT`	Action is not executable if there is a vehicle on a lane further left which (1) is between 10 m and 30 m in front and drives slower (2) is between 10 m behind and 10 m in front (3) is between 10 m and 20 m behind and drives faster
`LANE_RIGHT`	Action is not executable if there is a vehicle on a lane further right which (1) is between 10 m and 30 m in front and drives slower (2) is between 10 m behind and 10 m in front (3) is between 10 m and 20 m behind and drives faster
`FASTER`	Action is not executable if distance to front vehicle is less than 40 m
`IDLE`	Action is not executable if distance to front vehicle is less than 30 m
`SLOWER`	Action is not executable if distance to front vehicle is less than 10 m and (1) `LANE_LEFT` is enabled or (2) `LANE_RIGHT` is enabled

5.4.1. Formal B Model for Highway Environment

In the formal B model, we define variables storing the set of present vehicles (`PresentVehicles`), and total functions mapping each vehicle to its respective x and y-coordinates (`VehiclesX`, `VehiclesY`), and its velocities (`VehiclesVx`, `VehiclesVy`)

[3] `https://github.com/hhu-stups/reinforcement-learning-b-models`

```
1:  FASTER =
2:  PRE
3:      EgoVehicle ∈ dom(VehiclesVx) ∧
4:      ¬(∃v. (v ∈ PresentVehicles \ {EgoVehicle} ∧
5:      VehiclesX(v) > 0.0 ∧ VehiclesX(v) < 45.0  ∧
6:      VehiclesY(v) < 3.5 ∧ VehiclesY(v) > -3.5))
7:  THEN
8:      Crash :∈ BOOL ||
9:      PresentVehicles :| (PresentVehicles ∈ ℙ(Vehicles) ∧
10:     EgoVehicle : PresentVehicles) ||
11:     VehiclesX :∈ Vehicles → ℝ ||
12:     VehiclesY :∈ Vehicles → ℝ ||
13:     VehiclesVx :| (VehiclesVx ∈ Vehicles → ℝ ∧
14:         (Crash = FALSE ⟹
15:         VehiclesVx(EgoVehicle) ≥
16:         VehiclesVx'(EgoVehicle) - 0.05)) ||
17:     VehiclesVy :∈ Vehicles → ℝ ||
18:     VehiclesAx :| (VehiclesAx ∈ Vehicles → ℝ ∧
19:         (Crash = FALSE ⟹ VehiclesAx(EgoVehicle) ≥ -0.05)) ||
20:     VehiclesAy :∈ Vehicles → ℝ ||
21:     Reward :∈ ℝ
22: END
```

Listing 5.2. `FASTER` Operation in B Model for Highway Environment; each vehicle's position corresponds to its center, each vehicle's length is 5 m, each vehicle's width is 2 m; therefore we encode $[0.0, 45.0]$ in x-direction and $[-3.5, 3.5]$ in y-direction to formulate that the distance to the vehicle in front is less than 40 m.

and accelerations (`VehiclesAx`, `VehiclesAy`) in x and y-directions. The accelerations are computed from the current and previous observations wrt. the elapsed time between these two observations (one second). To make the formal model easier to understand, we define a set of `Vehicles` which includes the `EgoVehicle`. We further introduce a `Crash` and a `Reward` variable for validation purposes. Note that the encoding of the formal model in this section differs from the more abstract illustration described in Section 5.3.1.

Corresponding to the agent's action space, we encoded 5 actions into the formal B model: `IDLE`, `LANE_LEFT`, `LANE_RIGHT`, `FASTER`, and `SLOWER`. Table 5.1 shows the description of the guards for all operations that we use as safety shield in our experiments. The `SLOWER` action is guaranteed to be enabled if no other guard would hold. Listing 5.2 shows the `FASTER` operation in our formal model with a safety shield. The guard (see lines 3–6) for shielding the `FASTER` action states that `FASTER` is not enabled if the distance to the vehicle in front is less than 40 m. As each vehicle's position corresponds to its center, and its length is 5 m, we encode 45 m in the formula. In lines 13–16, we encode that the expected speed remains the same or increases with a tolerance of -0.05 m/s. Likewise, the acceleration should be positive with the same tolerance (see lines 18–19).

5.4.2. Training the Agents

We compare two trained DQN agents for the environment, both are trained over the `highway-fast-v0` environment with three lanes. The first agent uses default configurations for the environment and the reward function. We will refer to this agent as Base agent. The reward function rewards the agent based on the resulting environment state caused by its last action. The environment's default rewards are -1 for a collision, 0.1 when the agent is on the right-most lane, and 0.0–0.5 for a speed between 20–30 m/s (linearly scaled over the speed interval).

As it turned out, the agent's driving behavior proved to be rather risky, preferring speed over collision avoidance in certain cases and thus ending up with a high collision rate of almost 60 % (see Table 5.3). In response, we changed the penalty for collisions from -1 to -2. We also adjusted the reward for driving on the right-most lane from 0.1 to 0.2 to further the desired behavior of prioritizing the right-most lane. The agent trained with this altered reward function will be referred to as Higher Penalty.

For the DQN, we used a neural network with two hidden layers of 256 neurons each and a learning rate of 0.0005. The discount factor was set to 0.9 which affects the value of future rewards [219]: A reward received in k steps will only be 0.9^{k-1} times as valuable as if received immediately. The exploration rate decayed linearly from 1.0 to 0.05 within the first 6000 of a total of 20 000 training steps, indicating the ratio of actions which are taken randomly rather than following the thus far learned policy. This randomness is meant to overcome local maxima in the learned policy by regularly bypassing greedy behavior. The agents were each trained within 15 min.

Table 5.2.: Estimation of Average Values, Application of SimB Validation Techniques, and the Result of Validation; Values represent average metric values with standard deviation.

	Base		Higher Penalty	
Metric	no shield	with shield	no shield	with shield
Episode Length	38.85 ± 22.41	56.71 ± 11.47	53.02 ± 15.32	59.16 ± 5.54
Velocity [m/s]	23.37 ± 2.17	21.49 ± 0.94	21.14 ± 0.79	20.95 ± 0.63
Distance [m]	876.35 ± 477.62	1213.30 ± 244.48	1117.18 ± 321.71	1238.04 ± 122.12
On Right Lane [s]	31.69 ± 22.29	42.26 ± 20.51	47.07 ± 17.85	48.73 ± 17.52
Total Reward	30.41 ± 17.39	42.88 ± 8.86	39.90 ± 11.77	44.20 ± 4.42

5.4.3. Statistical Validation

Now, we apply SimB's statistical validation techniques to validate safety properties for the highway agent. For this, we evaluate 1000 execution runs per agent, once with and without a safety shield. We choose an episode length of 60 seconds for each run with a frequency of one observation per second. An episode might end earlier than 60 seconds if an accident occurs.

To estimate the RL agents' quality, we first gathered statistics over the resulting traces to get a feeling for how well the agents act in the first place. We measured averages of episode length, speed, distance traveled per episode, time on the right lane per episode, and reward. The results are shown in Table 5.2. One can see that Higher Penalty increases the average episode length to over 53 seconds, an increase of 14 seconds (+36.5 %) to Base. This indicates that the higher penalty was indeed a sensible choice. Further, we already see the benefits of shielding.

Table 5.3.: Safety Properties, Application of SimB Validation Techniques, and the Result of Validation. Percentages represent ratio of measured traces fulfilling the safety property.

		Base		Higher Penalty	
Safety Property		no shield	with shield	no shield	with shield
SAF1:	The agent must avoid collisions with other vehicles	45.4 %	91.8 %	78.5 %	97.4 %
SAF2:	The agent must drive faster than 20 m/s	93.4 %	91.4 %	76.9 %	83.0 %
SAF3:	The agent must drive slower than 30 m/s	95.2 %	98.8 %	100.0 %	100.0 %
SAF4:	The agent should decelerate at a maximum of 5 m/s^2	100.0 %	100.0 %	100.0 %	100.0 %
SAF5:	The agent should accelerate at a maximum of 5 m/s^2	100.0 %	100.0 %	100.0 %	100.0 %
SAF6:	To each other vehicle, the agent should keep a lateral safety distance of at least 2 m and a longitudinal safety distance of at least 10 m	6.4 %	49.2 %	41.6 %	70.5 %

Table 5.3 lists the safety properties we validated with SimB and the corresponding results for both agents with and without safety shields. The safety properties **SAF1**–**SAF6** cover the following aspects:

- **SAF1** is the main property and states that the agent must avoid collisions with other vehicles.

- **SAF2** and **SAF3** check that the agent drives with an appropriate speed.
- **SAF4** and **SAF5** check that the agent does not change speed by acceleration or braking abruptly.
- **SAF6** check that the agent should maintain appropriate distances from other cars to have enough room for reactions when accelerating, braking, and switching lanes.

Note that the validation objectives are not necessarily favored by the reward function, i.e., the agent is unaware of these specifications. For instance, **SAF6** is not rewarded during training. We intentionally validate untrained properties to show how the approach might capture such instances.

When evaluating **SAF1**, we found that the RL agent causes significantly fewer accidents if it is penalized more severely for accidents during training. We are also able to reduce the accident rate by encoding a safety shield. Especially for Base, the accident rate with a safety shield could be reduced to be safer than Higher Penalty without a safety shield. Despite the safety shield, collisions still occur in Higher Penalty. From the corresponding simulated traces, our technique discovered that almost all scenarios with collisions consist of the ego vehicle approaching another vehicle in front while performing `SLOWER` and driving at the set minimum speed of 20 m/s: the front vehicle drives even slower leading to the collision. Our technique also discovered that setting a lower minimum speed, e.g. 19 m/s, is also not an appropriate solution. In this case, the ego vehicle sometimes drives slower than all other vehicles which leads to all other vehicles driving away at the front of the highway. This means the ego vehicle drives alone at the back of the highway without any collisions. There is also another rare scenario leading to a collision (in the experiments it was 1 of the 1000 simulated traces): the ego vehicle collides with another vehicle on the other side of the highway, i.e. the ego vehicle and another vehicle drive in the opposite outer lanes and both switch to the center lane simultaneously.

Driving too slow does not seem to be a factor in crashes (see **SAF2**). Sometimes, the actual speed might be slightly below the desired minimum speed, especially for Higher Penalty. While this seems to work against the environment's specification, we did not correct this with the safety shield, as there might be situations where it is sensible to brake and drive slower. Furthermore, Base agent sometimes drives slightly too fast, i.e., exceeding the speed limit of 30 m/s (see **SAF3**). As shown in the values for **SAF6**, it seems that maintaining safe distances is significantly more important to avoid accidents rather than exceeding the speed limit. In all four variations, the RL agent never accelerates or decelerates heavily, i.e., **SAF4** and **SAF5** are never violated. This is to be expected as the encoded acceleration range for the agent is [-5,5] by default. Thus, with the validation of **SAF4** and **SAF5**, we also validated the implementation of the RL agent. Relating the validation results to Table 5.2 again, we see that with safety shields:

- The average speed is slower, but the distance traveled and the average episode length are greater. An interesting result here is that Base with a safety shield

achieves a lower crash rate than HIGHER PENALTY without a safety shield even with a higher average speed.

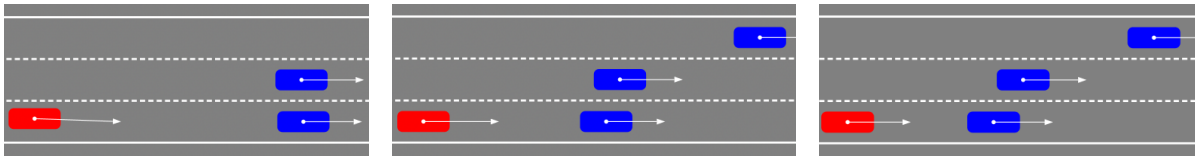
- The safety distances are maintained more often which seems to be the main reason for fewer crashes. Especially, BASE with a safety shield maintains safety distances better than HIGHER PENALTY without a safety shield.
- The cumulative reward is higher with a smaller standard deviation.
- The agent drives on the right lane more often.

Thus, our results highlight the safety capabilities of the employed shield and how pre-shielding can alleviate shortcomings during training. This helped us to calibrate the reward function better. Note that this work does not demonstrate that the manual encoding of the safety shield is perfect; in the future, we will consider shield synthesis [125]. However, we show that one can use a formal specification as a safety shield and that it achieves better RL performance than without.

5.4.4. Validation by Trace Replay

Now, we discuss validating the agent's behavior with trace replay, highlighting the role of safety shields. For easier understanding, we employ a domain-specific visualization [243] for the highway environment. We focus on two different, observed scenarios[4].

Figure 5.4 shows a scenario where the ego vehicle approaches another vehicle and slows down. Here, the agent was able to detect the vehicle in front and brake in time. The safety distance to the vehicle in front is hence kept and an accident could be avoided. Further, the RL agent seems to be aware of another vehicle in the center lane as it decides to slow down rather than switch lane. The scenario shown in Figure 5.4 was simulated without safety shields. When re-playing this trace with safety shields being activated, the trace is still feasible. So, in this scenario, the RL agent behaves correctly without intervention by safety shields.



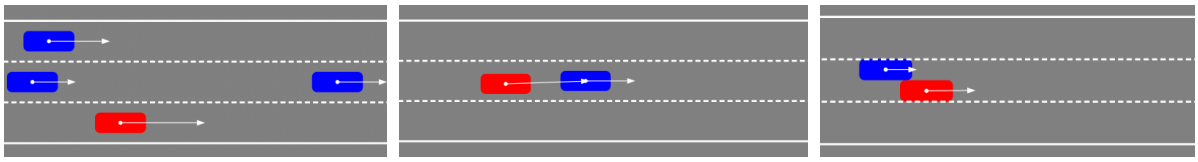
(a) Ego Vehicle Approaches (b) Ego Vehicle Slows Down (c) Ego Vehicle Slows Down

Figure 5.4.: Example for Approaching Scenario; white arrows show the direction of the velocity vector.

A second scenario is shown in Figure 5.5. Here, the agent switches to the center lane while keeping a high velocity. After switching, the ego vehicle has to slow down as it is

[4] A scenario is a sequence of events which alters the system's state. Scenarios as static exports [235] available at: `https://hhu-stups.github.io/highway-env-b-model/`

approaching another vehicle in front. As the agent does not brake in time, it collides with the vehicle in front. This scenario was also simulated without safety shields. When trying to re-play the trace with safety shields, the trace is not feasible anymore, especially when the RL agent tries to execute `LANE_LEFT`. Thus, a collision could have been avoided in this scenario by using safety shields.



(a) Switch to Center Lane (b) Ego Vehicle Slows Down (c) Ego Vehicle Crashes

Figure 5.5.: Example for Crash Scenario; white arrows show the direction of the velocity vector.

Of course, the question of why the agent behaves in the observed manner cannot be answered completely. While the RL agent seems to behave in certain ways that correspond to similar human intuition, there still is no way to properly find reason in the agent's behaviors. This is due to the black box nature of neural networks underlying our DQN approach. While explainable AI methods from research [228] might offer insights, there are no guarantees they may accurately capture black box agents [202] and different explainers might even yield conflicting explanations. These problems with explainable AI emphasize the need for proper validation tools for RL agents, as outlined in this work.

5.5. Related Work

This section compares this work with other works in the field of formal methods for AI, with a stronger focus on RL.

Justified Speculative Control (JSC) [79] is a technique to achieve safe RL with formal methods. In JSC, formal verification results are obtained and integrated into the RL agent's controller. The verification results also provide a set of safe actions from which an RL agent can choose for execution. In our work, the RL agent also chooses from a set of safe operations which are computed from the manually encoded operations' guards in the formal model. While the formal model in our work could be verified for safety properties (depending on the model's state space), the creation of the shield is driven by requirements rather than verification results. Still, we can detect and avoid dangerous situations.

Sha [208] presented an approach to "use simplicity to control complexity" in which a simpler system monitors a complex system at runtime, and intervenes when certain rules are violated. Sha's concept is independent of reinforcement learning; the given example is about a complex Boeing flight system that was checked for laws by a simple, reliable controller. Based on Sha's concept, Phan et al. [193] presented a *neural simplex architecture* (NSA) for reinforcement learning. The NSA consists of a pre-certified *decision*

module which switches between the complex unverified *neural controller* and a verified *baseline controller* if the former tries to execute a potentially dangerous action. Referring to Sha, the RL agent can be viewed as the complex system, and the formal model with the safety shield as the simple controlling system in our work. Furthermore, the safety shield influences every decision for the RL agent as ProB uses the formal model to compute actions that are deemed to be safe.

Shield synthesis [125] is a runtime verification technique which also aims to achieve safe RL. After modeling the environment as a Markov Chain, a shield is synthesized which may take over the agent's decision-making for a (possibly limited) number of steps. The shield acts once the probability of reaching an unsafe state shortly exceeds a given threshold. In our technique, we encoded shields by hand rather than synthesizing them. While the burden of precisely formulating the shielding conditions is now placed on the modeler, we can assume the RL agent's internal decision-making process as black box. However, we do not guarantee that the shield will be returning control to the agent eventually. In shield synthesis, there is also the concept of enforcing temporal properties, especially LTL properties [12]. Assuming that the formal model's state space is finite, one could also use ProB's LTL model checker [194] in our approach, to verify LTL properties on the RL agent (with shielding). When the formal model fulfills a safety property, then we know that this safety property is also enforced for the RL agent. However, this is not the case for liveness properties.

Deep RL is implemented using neural networks for which there are also verification approaches [207, 112, 111], including techniques such as abstract interpretation [86], SMT solving [121], and proving [201]. Our work mainly focuses on validation and does not yet tackle the challenge of verifying the RL agent extensively. In the future, we should investigate how to achieve and guarantee better safety of RL agents in our approach.

Search-based testing [221] is a technique which uses a depth-first search to find safety-critical states. The RL agent is then brought into a situation close to the safety-critical state to test how well it avoids this state. The technique also applies *fuzz testing* to achieve better coverage of the RL agent's behavior. *Differential safety testing* [220] is another technique to test RL agents for safety, which makes use of automata learning [160, 161], probabilistic model checking [11], and statistical methods. With Monte Carlo simulation, our work simulates multiple different scenarios. Based on the resulting execution runs, our work can estimate certain values and compute the likelihood of fulfilling certain safety properties. The results are then used to evaluate and improve the safety shield and the reward function. However, we have not yet navigated the RL agent into critical situations for testing purposes.

Wang et al. [238] presented a safety-falsification method which works as an adversary for the RL agent. The technique uses metric temporal logic formulas to enforce the RL agent to violate safety properties. As these properties are difficult to integrate into the reward function, safety-falsification helps the RL agent to train adversarial behavior. As we do not use a safety shield during training, our RL agents also experience the consequences of bad behaviour in the form of reward penalties.

Shalev-Shwartz et al. [209] presented a formal model for safe behavior of self-driving cars, called responsibility-sensitive safety (RSS). This model was later extended and

translated to Event-B [124]. The rules of RSS in general and the Event-B model, in particular, could be integrated as safety shields into our approach in the future.

5.6. Conclusion and Future Work

This work presented a technique to validate RL agents with formal methods tools and techniques. We create a formal model at a high-level abstraction and link it with the RL agent. This allows to use the formal specification as a safety shield for the RL agent. Furthermore, the formal model encodes the RL agent's expected external behavior, i.e., the RL agent's actions and its environment. It is then possible to apply validation techniques like trace replay, simulation, or statistical validation.

In this work, we successfully demonstrated our technique using the formal B method with the tools ProB and SimB on a highway environment. With trace replay, and real-time simulation, we can replay the agent's situation and reason about its decisions. Here, we also demonstrated that dangerous scenarios are avoided by safety shields in the formal model. Applying statistical validation techniques, we can estimate the likelihood of fulfilling various safety requirements, e.g., the likelihood of crashes or not maintaining safety distances. We also estimate certain values, e.g., the average reward, the average speed, the average distance of one episode, the average time on the right lane of one episode, or the average episode length of the RL agent on the highway. With the gained knowledge, we improved safety shields which again increased safety. Safety shields were effective in reducing the likelihood of crashes at the cost of reducing the average velocity, overall increasing the safety of the model. With the manually encoded safety shields in the formal model, we also achieve higher rewards for the agent. We were even able to validate the reward function, highlighting where we needed to adjust the respective weights. Furthermore, we were able to validate the implementation of the RL agent. All models, including highway environment, are available online at:

`https://github.com/hhu-stups/reinforcement-learning-b-models`

While our approach enables various validation techniques, verification has yet not been tackled actively. We aim to validate and better understand the RL agent's behavior to collect assumptions about the agent and its environment. Based on this, we plan to verify the model with techniques like model checking or proving. Assuming that safety properties are fulfilled for the formal model, we can also conclude these properties for the RL agent, as the formal model is encoded as an over-approximation of the RL agent. As future work, one could further investigate how our approach can be extended by shielding over LTL properties.

Acknowledgements.

We would like to thank Davin Holten for his initial experiments showing the feasibility of ProB's techniques — especially of SimB — for the highway RL agent.

6. Additional Improvements and Evaluations

Chapter 5 enables the simulation of RL agents via SimB at runtime, while using formal models as safety shields. A safety shield defines rules enforced on an RL agent. In Chapter 5, we evaluated a highway AI trained with RL. Here, we formulated the safety distances required to perform an action. We encoded those safety distances as fixed values, e.g., a minimum of 40 meters to the vehicle in front is required to accelerate, i.e., to perform `FASTER`.

The results from Chapter 5 ([232]) showed that a safety shield can improve the safety of a highway AI trained with RL. As an improvement to the formal B model in Chapter 5, Michael Leuschel develops a safety shield based on a formula (see Section 6.1) from the Responsibility-Sensitive Safety (RSS) [209] technique. The formal model is available in:

`https://github.com/hhu-stups/reinforcement-learning-b-models`

The RSS safety shield encodes safety distances based on the positions, speeds, and accelerations of the RL agent and other vehicles in the environment (and not fixed values anymore). The RSS shield computes safety distances, considering the worst case where vehicles in front brake with maximum deceleration. The RSS technique and the results from Chapter 5 ([232]) also inspired the development of the ABZ case study 2025 [157], which presents safety aspects for an autonomous driving system on the highway.

In this chapter, we evaluate the RSS shield with the validation techniques in SimB, comparing it with the shield in Chapter 5. Note that this chapter presents the validation results of the RSS shield only and does not provide a complete proof. Although the results are better than in Chapter 5, the RSS shield is still preliminary, and we may improve it in the future.

I contributed to the training and validation of the RL agents (with safety shields), while Michael Leuschel developed the formal model for the RSS shield. In Section 6.1, we present the RSS technique. In Section 6.2, we present the safety considerations for the RSS model developed by Michael Leuschel. In Section 6.3, we present the evaluation results of the RSS shield. We also train new agents using slightly modified parameters from those in Chapter 5 to improve safety. Furthermore, we train an adversarial agent for stress testing. Section 6.4 discusses the validation-driven development process we followed to improve safety step by step.

6.1. Responsibility-Sensitive Safety

Responsibility-Sensitive Safety (RSS) [209] is a mathematical model to ensure the safety of autonomous driving. It contains five safety rules, covering aspects like safety distance, cutting in, right of way, limited visibility, and avoiding crashes. The formal B model developed by Michael Leuschel implements the first rule, described by the following formula [209]:

$$d_{min} = [v_r * \rho + \frac{1}{2} * a_{max} * \rho^2 + \frac{(v_r + \rho * a_{max})^2}{2 * \beta_{min}} - \frac{v_f^2}{2 * \beta_{max}}]_+ \tag{6.1}$$

The formula for d_{min} computes the *RSS safety distance* between an ego vehicle (rear vehicle) and a front vehicle. We use the notation $[x]_+ := max\{x, 0\}$ from [209]. The formula uses the following variables:

- v_r for the speed of the rear vehicle,
- v_f for the speed of the front vehicle,
- a_{max} for the maximum acceleration of the rear vehicle before braking,
- β_{max} for the maximum deceleration (braking acceleration) of the front vehicle,
- β_{min} for the deceleration of the rear vehicle (reaction to braking of the front vehicle), and
- ρ for the response time, i.e., the reaction time of the rear vehicle

Assuming that the rear vehicle is approaching another vehicle that brakes abruptly to the maximum. The formula takes into account the rear vehicle's response time and current positions, speeds, and accelerations of both vehicles to compute the braking distance.

6.2. Safety Considerations

This chapter presents the safety considerations implemented by Michael Leuschel in the formal RSS model. As mentioned earlier, Michael Leuschel created and developed the formal B model, while my contribution is the training and validation of the RL agents. The main requirement (also part of [157]) to fulfill is:

- **SAF**: The agent must avoid collisions with other vehicles.

Therefore, the formal model takes into account the RSS formula in Equation (6.1) to encode safety distances in the guards of the actions performed by the RL agent: `FASTER`, `IDLE`, `SLOWER`, `LANE_LEFT`, and `LANE_RIGHT`.

The shield blocks the agent from executing `FASTER` if the action does not guarantee the RSS distance to the front vehicle. The agent can execute `IDLE` only if it maintains the

RSS distance to the front vehicle at the current speed. The agent is not allowed to switch lanes, i.e., perform `LANE_LEFT` or `LANE_RIGHT`, if there is another vehicle right next to the agent's vehicle where switching to the next lane would violate the RSS safety distance. The formal model also considers the case where (1) the distance to the front vehicle decreases while violating the safety distance, and (2) the distance to a vehicle on the left-hand or right-hand side in front increases. In this case, `LANE_LEFT` and `LANE_RIGHT` are deemed safer than `SLOWER`. Consequently, this is the only situation where `SLOWER` is disabled. In all other scenarios, the formal model enforces that when `FASTER`, `IDLE`, `LANE_LEFT`, and `LANE_RIGHT` are all deemed unsafe, the RL agent defaults to executing `SLOWER`.

Note that in certain situations, such as when the ego vehicle is surrounded by other vehicles in all directions, no action can guarantee the safety distance.

6.3. Empirical Results for Shields in Highway Environment

This section evaluates the performance of the RSS shield compared to the shield presented in Chapter 5 ([232]). We evaluate the safety shields on three RL agents, with one of them trained to behave adversarially.[1] The reward functions of the Base and the Higher Penalty are slightly modified compared to Chapter 5 ([232]). Therefore, we repeat the training process, resulting in new RL agents. In contrast to this chapter, no adversarial agent was trained in Chapter 5.

This section describes the training of the RL agents, evaluates the RSS shield, and discusses the threat to validity.

6.3.1. Training

Similar to Chapter 5, we use the `highway-fast-v0` [147] environment with three lanes.

We do not employ safety shields during training, arguing that the RL agent should also learn bad behavior. The safety shields are employed at runtime and intervene when a dangerous situation is detected.

Similar to Chapter 5, we train the Base with a penalty of `-1` for a collision and a reward of `0.1` for driving in the right-most lane. For the Higher Penalty, we define a penalty of `-2` for a collision and a reward of `0.2` for driving in the right-most lane (similar to Chapter 5).

In Chapter 5, we noticed that the RL agent never drives slower than 20 m/s because `target_speeds` is set to `[20,25,30]` by default. This configuration led to collisions when there are front vehicles in all lanes, which are currently driving slower and making overtaking impossible. Consequently, we make some adjustments compared to Chapter 5. We allow both agents to choose speeds between 0 and 40 m/s, with target speeds in

[1] Commit hash: `2721d360e7d345411d582be3ff888a068b1833ea` in `https://github.com/hhu-stups/reinforcement-learning-b-models`

steps of 5 m/s, i.e., we set `target_speeds` to `[0, 5, 10, 15, 20, 25, 30, 35, 40]`. Furthermore, we linearly scale the reward for the speed from 0 to 40 m/s, i.e., we configure the `reward_speed_range` as `[0, 40]`. Initially, we noticed that this caused the agent to drive slower than any other vehicle. In the environment, the agent eventually drives alone on the highway. To avoid this, we increase the `high_speed_reward` from `0.4` to `1` for both agents.

We use the same configuration for training as in Chapter 5: We apply DQN to train neural networks with two hidden layers of 256 neurons each and a learning rate of 0.0005, and a discount factor of 0.9. We also train the agents with 20 000 training steps, which takes around 15 minutes for each.

6.3.2. Results

Similar to Chapter 5, we apply Monte Carlo simulation and statistical validation techniques with SimB to validate Base and Higher Penalty. We run Monte Carlo simulation for Base and Higher Penalty in the configurations (1) without a safety shield, (2) with the safety shield from Chapter 5, and (3) with the RSS shield developed by Michael Leuschel, each with 1000 runs. Each run lasts 60 seconds, and the response time of both agents is one second, i.e., the agents observe the environment and perform an action every second. Similar to Chapter 5, a run might end earlier than 60 seconds when a crash occurs. In the following, we will refer to the safety shield from Chapter 5 as the *naive (safety) shield*.

Table 6.1 shows the percentages of crash-free runs for Base and Higher Penalty, i.e., the percentages of runs where the agent fulfills **SAF**. The results show that the adapted reward function already increases the percentage of crash-free runs to 93 % and 98 % for the Base and the Higher Penalty, compared to 45.4 % and 78.5 % in Chapter 5, respectively. Similar to the results in Chapter 5, the naive shield from Chapter 5 increases safety. With the naive shield, both agents achieve 99.9 % of crash-free runs. RSS reduces the percentage of crash-free runs to 100 % for both agents.

Table 6.1.: Percentage of Crash-Free Runs (percentages for fulfilling **SAF**) for Base, Higher Penalty, and Adversarial, each (1) unshielded, (2) with naive shield (from Chapter 5), and (3) with RSS shield

	no shield	naive shield	RSS shield
Base	93.0 %	99.9 %	100.0 %
Higher Penalty	98.0 %	99.9 %	100.0 %
Adversarial	0.5 %	23.9 %	98.7 %

With the adjusted training configuration, the agents' safety has significantly improved. Consequently, we cannot conclude from the results of the Base and the Higher Penalty that the RSS shield is safer than the naive shield. Additionally, we train an adversarial agent (Adversarial) to demonstrate the efficiency of the RSS shield (compared to the

naive shield). We reward ADVERSARIAL for driving fast, changing lanes, and even for collisions. Here, we adapt the parameters to `2` for the highest speed, `0.5` for changing lanes, and `1` for collisions. Table 6.1 also shows the results for ADVERSARIAL.

Only 0.5 % of the execution runs are crash-free for ADVERSARIAL unshielded. ADVERSARIAL with the naive shield only achieves 23.9 % of crash-free runs. The RSS shield increases the percentage of crash-free runs to 98.7 %. The remaining collisions with the RSS shield are mostly scenarios where ADVERSARIAL drives at maximum velocity, while the braking distance is less than the perception distance. These results demonstrate the significant improvement of the RSS shield compared to the naive shield in Chapter 5.

We also evaluate several metrics related to the agents' behavior, similar to Chapter 5. Here, we analyze BASE and HIGHER PENALTY (1) unshielded, (2) with the naive shield, and (3) with the RSS shield. The metrics contain average values for the episode length, the velocity, the distance traveled, the time spent on the right lane, and the total reward per episode.

The results for BASE and HIGHER PENALTY are shown in Table 6.2 and Table 6.3, respectively. Both agents achieve a higher average episode length with the RSS shield due to fewer collisions. Both agents also drive slower on average with the RSS shield, as it is stricter with safety distances. The cumulative reward and average distance traveled increase with both safety shields due to fewer collisions. Interestingly, the time spent in the right lane reduces with the RSS shield.

Table 6.2.: Results of Applying SimB Validation Techniques to Estimate Metrics for BASE. Values represent average values with standard deviation.

Metric (from Chapter 5)	no shield	naive shield	RSS shield
Episode Length	58.13 ± 8.18	59.95 ± 1.74	60.00 ± 0.00
Velocity [m/s]	21.00 ± 0.76	20.83 ± 0.76	20.80 ± 0.68
Distance [m]	1218.92 ± 177.02	1246.44 ± 58.52	1245.82 ± 41.47
On Right Lane [s]	50.83 ± 18.02	48.58 ± 20.59	46.15 ± 22.16
Total Reward	44.62 ± 6.49	45.74 ± 1.66	45.70 ± 1.05

Table 6.3.: Results of Applying SimB Validation Techniques to Estimate Metrics for HIGHER PENALTY. Values represent average values with standard deviation.

Metric (from Chapter 5)	no shield	naive shield	RSS shield
Episode Length	59.31 ± 5.43	59.94 ± 1.77	60.00 ± 0.00
Velocity [m/s]	20.66 ± 0.65	20.59 ± 0.68	20.56 ± 0.65
Distance [m]	1223.65 ± 119.29	1232.30 ± 54.76	1231.42 ± 39.76
On Right Lane [s]	53.65 ± 14.92	50.70 ± 17.95	46.25 ± 21.97
Total Reward	50.11 ± 4.71	50.45 ± 1.73	50.29 ± 1.06

Another interesting result is that with both shields, BASE drives as safely as HIGHER PENALTY while driving a longer distance. However, due to the training configuration, BASE without a safety shield drives more aggressively than HIGHER PENALTY because it is less penalized for collisions, leading to more collisions (see Table 6.1).

Threats to Validity. The newly trained RL agents with adjusted configurations for BASE and HIGHER PENALTY act much more safely than our evaluation in Chapter 5. Although this is desirable, it also outlines possible threats to validity. The results for BASE and HIGHER PENALTY might create the impression that the RSS shield performs as safely as the naive shield. However, when evaluating the shields on ADVERSARIAL, we observe that the RSS shield is significantly safer than the naive shield.

In this section, we tested critical behavior with an adversarial agent rewarded for dangerous behavior. With the adversarial agent, we discovered that the RSS shield cannot always prevent collisions when the braking distance is less than the perception distance.

There are some related approaches: Reimann et al. [199] present an approach to stress-test an AI system by generating critical scenarios based on temporal logic formulas. Reimann et al. also utilize the RSS technique in the automotive domain to test whether those critical scenarios fulfill the safety distances. Another approach by Scher et al. [204] focuses on finding errors with fewer Monte Carlo simulation runs.

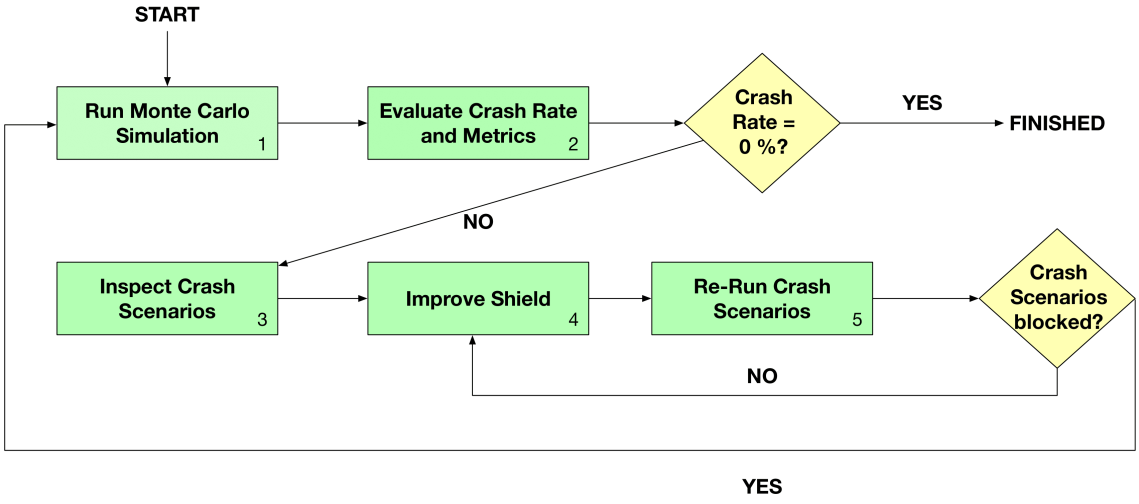


Figure 6.1.: Workflow: Validation-Driven Development with Monte Carlo Simulation, Estimation of Crash Rate and Metrics, Trace Replay for Inspection of Crash Scenarios, and Improvement of Shield

6.4. Validation-Driven Development

This section describes the workflow while developing the formal RSS model. The workflow aligns with *validation-driven development* presented by Stock et al. [216] and is

an extended presentation of using trace replay (initial ideas in Section 5.4.4) with Monte Carlo simulation and estimation of likelihood to enrich the validation process.

Figure 6.1 illustrates the workflow, which is as follows: First, we run Monte Carlo simulation to estimate the likelihood of a crash. If there are no crashes, we consider the safety shield sufficient to avoid collisions. In this context, it is crucial to test critical scenarios (as discussed in Section 6.3.2).

Otherwise, we can inspect the crash scenarios with trace replay and reason about the errors that led to crashes. One can then improve the safety shield to avoid the crash scenarios we have inspected before. To validate that the improved safety shield avoids these crashes, we proceed to trace replay. The improved safety shield should block decisions that lead to crashes. Otherwise, the changes were ineffective, and one must improve the safety shield again. Finally, the workflow returns to step 1 by re-running Monte Carlo simulation, expecting the crash rate to decline with the improved safety shield. Otherwise, the changes might have worsened the safety.

In particular, this workflow helped to debug the formal model, identify errors and weaknesses, and improve it step-by-step. I have done steps 1–3 and 5 in Figure 6.1, whereas Michael Leuschel has done step 4. The results from the 3rd and 5th steps provided helpful feedback to improve the formal model. Within the fourth step, a modeler improves the formal model, i.e., the safety shield.

Part III.

Code Generation

7. Model Checking B Models via High-Level Code Generation

Abstract. We present a new approach to improve the model checking performance for B models. We build on the high-level code generator B2Program, which unlike B's original code generators can already be applied at an early stage to high-level B models. We extend B2Program to generate efficient model checkers in Java and C++. The generated model checkers are customized and compiled for specific B models and include features like parallelization and caching. We evaluate the approach on a wide range of B models, comparing the performance to existing B model checkers. The results show that for some models we can obtain significant performance improvements, while for others ProB remains the tool of choice. For lower-level models, our new approach improves upon the existing TLC backend. In summary, the B2Program model checker is a very useful new tool addition for the B method.

Keywords. Code generation, B method, Model checking

Funding. The works of Fabian Vu and Michael Leuschel are part of the IVOIRE project, which is funded by "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF) grant # I 4744-N.

7.1. Introduction and Motivation

When using formal methods, software is often modeled step-by-step until all desired features are encoded. During each development step, the model is verified, e.g., by model checkers such as ProB [152] or by provers such as AtelierB [51]. In the B method [4], the model is refined until reaching an implementable subset of the language, called B0, before code generation is feasible. Thus, code generation is applied at the end of the development cycle to generate executable code. AtelierB [51] contains several code generators, which translate B0 code to C and Ada. In an earlier work [233], we presented the code generator B2Program[1] which generates code from high-level B specifications to Java and C++. In contrast to AtelierB, B2Program is capable of code generation from models using high-level data structures such as sets and relations. However, B2Program is not meant for generating code for safety-critical embedded systems, as it uses dynamic

[1] Available at: `https://github.com/favu100/b2program`

heap allocation. Nevertheless, the main advantage is that code generation, e.g., for efficient simulation, is feasible at an early stage without refining to B0.

The idea of this paper is to use B2Program for efficient explicit-state model checking. Indeed, existing model checkers like ProB or TLC [246] *interpret* the model, and using *compiled generated* code could lead to significantly improved performance. In this article, we extend B2Program to generate customized model checkers for high-level B models. A difficulty is that a model checker has to compute *all* enabled transitions of a model (and not just one), and has to be able to switch from one arbitrary state to another (and not just from one state to a successor state). Our main motivation is to achieve high performance, but the new model checker can also be used as a second toolchain with ProB to safe-guard against bugs in the tools (discussed in [233] and [100]).

Section 7.2 explains how B2Program is extended for model checking. Section 7.3 discusses the limitations of B2Program. Section 7.4 evaluates the performance compared to ProB and TLC (translation to TLA+ by TLC4B [100]). Finally, we compare this work with existing code generators and model checkers in Section 7.5, and conclude in Section 7.6.

7.2. Code Generation for Model Checking

This section presents how we extended B2Program for model checking. In the previous work [233], we generated code from a verified model for execution, while here we generate code for verification (or model checking to be precise).

```
MACHINE TrafficLight
SETS colors = {red, redyellow, yellow, green}
VARIABLES tl_cars, tl_peds
INVARIANT tl_cars : colors & tl_peds : {red, green} &
         (tl_peds = red or tl_cars = red)
INITIALISATION  tl_cars := red || tl_peds := red
OPERATIONS
cars_ry = SELECT tl_cars = red & tl_peds = red THEN tl_cars := redyellow END;
cars_y = SELECT tl_cars = green & tl_peds = red THEN tl_cars := yellow END;
cars_g = SELECT tl_cars = redyellow & tl_peds = red THEN tl_cars := green END;
cars_r = SELECT tl_cars = yellow & tl_peds = red THEN tl_cars := red END;
peds_r = SELECT tl_peds = green & tl_cars = red THEN tl_peds := red END;
peds_g = SELECT tl_peds = red & tl_cars = red THEN tl_peds := green END
END
```

Listing 7.1. Example of a Traffic Light Controller in B

A B specification is composed of B machines, each with its constants and variables. The state of a B machine consists of the values of the constants and variables; the latter can be modified by operations. Operations contain guards or preconditions which are used to define whether the operation is enabled. An important feature of B are the invariants, which often encode important safety properties. A running example (of a traffic light controlling cars and pedestrians) is shown in Listing 7.1.

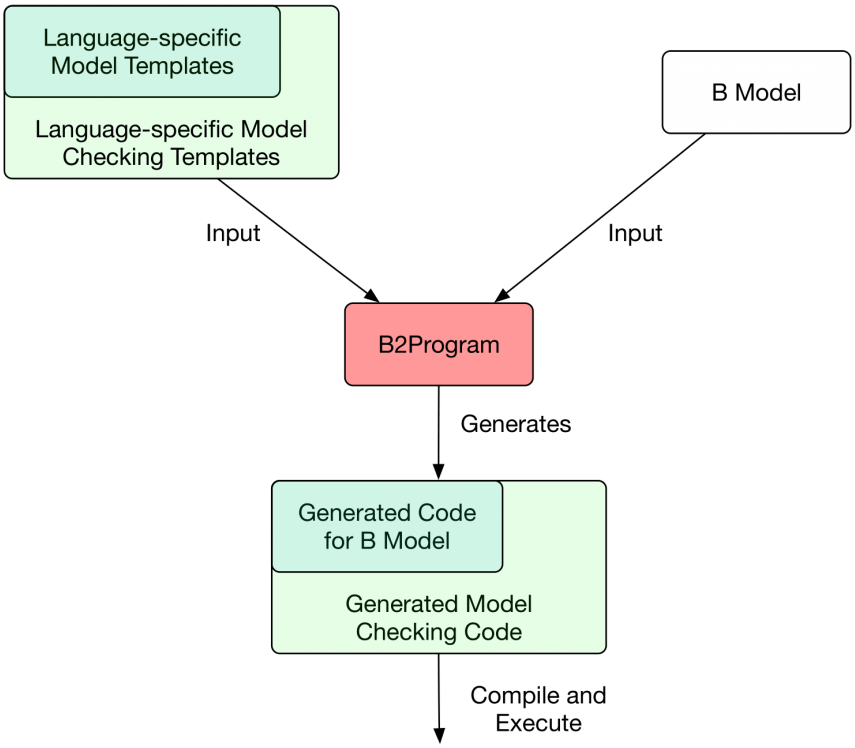


Figure 7.1.: Workflow of Model Checking

7.2.1. Extension of Generated Code

B2Program uses the StringTemplate [188] engine which makes it possible to generate code for multiple languages. In this article, we focus on Java and C++, but other target languages (like TypeScript/JavaScript [234], Rust, and Python) are being added to B2Program. The earlier work [233] provided templates for B's operators and constructs; by instantiating and assembling these templates one obtains the target code for the given model. In the previous work, code was generated from the model's operations, while the execution had to be controlled by a manually implemented main function. Thus, there was no computation of enabled operations (all parameters were provided by the main function). Furthermore, it was assumed that the model was already verified, i.e., code generation was not applied for constructs that are relevant for verification such as the invariant or preconditions.

In this article, we have added language-specific model checking templates which are weaved into the target code (see Figure 7.1). These templates contain the model checking algorithm, the computation of all enabled operations, and the evaluation of the invariant. A user can then verify invariants and deadlock-freedom in the model by compiling and executing the generated target code. When finding a violation, a counter-example is displayed showing a trace with states and executed events between them. By writing templates for model checking, it is possible to keep model checking code generation generic, i.e., to generate code for any B model (in B2Program's supported subset of B; discussed in Section 7.3) and for several languages with a single code generator.

This avoids implementing a model checker for each B model. For Java and C++, we implemented the model checkers with placeholders for the model's constructs.

In the following, we will first describe the evaluation of the invariant, and the computation of enabled operations. Afterward, we will explain the features that are implemented in the model checking algorithm.

Checking Invariant. The invariant predicate is decomposed into its conjuncts, each translated to a Boolean function. By splitting the invariant, it is possible to implement invariant caching (discussed later in this section). The generated Java code for checking the invariant of Listing 7.1 is shown in Listing 7.2.

```
public boolean _check_inv_1() {
  return new BBoolean(_colors.elementOf(tl_cars).booleanValue()).booleanValue();
}
public boolean _check_inv_2() {
  return new BBoolean(new BSet<colors>(colors.red, colors.green)
         .elementOf(tl_peds).booleanValue())
         .booleanValue();
}
public boolean _check_inv_3() {
  return new BBoolean(tl_peds.equal(colors.red).booleanValue() ||
                      tl_cars.equal(colors.red).booleanValue()).booleanValue();
}
```

Listing 7.2. Generated Java Code from `INVARIANT` of Listing 7.1

Computing Enabled Operations. Relevant B constructs for computing enabled transitions are `PRE`, `SELECT`, `ANY`, `CHOICE` substitutions, non-deterministic assignments, and high-level `PROPERTIES` constraining the possible values for the model's constants. In the previous work [233], we could treat any non-deterministic constructs in a simplified manner, such that only one possible execution path was chosen. In the context of model checking, it is necessary to cover *all* possible execution branches. It is, however, difficult to treat non-deterministic constructs deep within a B statement in Java or C++ code. A failed guard deep within a B statement could in principle be translated to an exception, but supporting all non-deterministic constructs is more difficult. Hence, we currently only allow non-determinism in top-level `PRE` and `SELECT` constructs; so the B model has to be rewritten to move non-determinism and guards to the top level. [2]

For operations without parameters, the guard or precondition is translated to a Boolean function evaluating whether the operation is enabled. Such a translation for `cars_ry` in Listing 7.1 to Java is shown in Listing 7.3.

```
public boolean _tr_cars_ry() {
  return new BBoolean(tl_cars.equal(colors.red).booleanValue() &&
                      tl_peds.equal(colors.red).booleanValue()).booleanValue();
}
```

Listing 7.3. Generated Java Code to Compute Enabledness of `cars_ry` in Listing 7.1

[2] In the absence of the WHILE loop, such a rewriting is always possible (cf. the normal form for substitutions in Chapter 6 of [4]).

In contrast, the computation of enabled transitions for operations with parameters is more difficult. Here, B2Program calculates the set of parameters for which the operation is enabled. Let $p_1, \ldots, p_n$ be the operation's parameters constrained by the precondition or guard P. It is then translated similarly as the set comprehension $\{p_1, \ldots, p_n \mid P\}$. [3] The rules after which quantified constructs are translated can be found in Section 3.5 of [233] and ensure that B2Program can create code to enumerate all quantified values.

As an example, the function for computing all parameter values to make the operation `SetCruiseSpeed(vcks, csam) = PRE vcks : BOOL & csam : BOOL & CruiseAllowed = TRUE THEN ... END` enabled is generated as shown in Listing 7.4.

```
public BSet<BTuple<BBoolean, BBoolean>> _tr_SetCruiseSpeed() {
  BSet<BTuple<BBoolean, BBoolean>> _ic_set_1 =
    new BSet<BTuple<BBoolean,BBoolean>>();
  for(BBoolean _ic_vcks_1 : BUtils.BOOL) {
    for(BBoolean _ic_csam_1 : BUtils.BOOL) {
      if((CruiseAllowed.equal(new BBoolean(true))).booleanValue()) {
        _ic_set_1 = _ic_set_1.union(new BSet<BTuple<BBoolean, BBoolean>>
                          (new BTuple<>(_ic_vcks_1, _ic_csam_1)));
      }
    }
  }
  return _ic_set_1;
}
```

Listing 7.4. Generated Java Code to Compute Transitions for `SetCruiseSpeed`

Copy Machine. From any given state there can be multiple enabled operations. When executing a single transition, the machine's current state is modified. It is then necessary to restore the previous state to execute another transition. To achieve this, we copy the machine's state before executing a transition during model checking.

```
public TrafficLight_MC(colors tl_cars, colors tl_peds) {
  this.tl_cars = tl_cars;
  this.tl_peds = tl_peds;
}
public TrafficLight_MC _copy() {return new TrafficLight_MC(tl_cars, tl_peds);}
```

Listing 7.5. Generated Java Code to Copy Machine from Listing 7.1

On the technical side, a copy constructor and a copy function are generated returning a new instance of the machine. Here, only references and not the data itself are copied. Note that B2Program is designed such that operations on data structures are applied immutably. An example of a copy constructor and a copy function for Listing 7.1 is shown in Listing 7.5.

7.2.2. Model Checking Features

The core of the algorithm is standard and follows [18]. So far, B2Program implements explicit-state model checking, verifying invariants and deadlock-freedom. LTL and

[3] Note that top-level preconditions are treated as similar to guards, and we only allow top-level guards and preconditions as non-determinism.

symbolic model checking are not supported. When a violation is found, a counter-example is displayed showing a trace with states and executed events between them. Additionally, we implemented parallelization as well as invariant and guard caching to improve the performance.

Invariant and Guard Caching. The techniques implemented here are inspired by the work of Bendisposto and Leuschel [28], and Dobrikov and Leuschel [64], but for our purposes, we only use lightweight caching techniques without using proof information or performing semantic analyses.

Taking a look at Listing 7.1: If an operation does not modify the variable `tl_peds` (e.g. `cars_y`), the model checker does not have to check the invariant `tl_peds` : {red, green} after applying the operation (provided no invariant violation has been found thus far). Similarly, if a guard of an operation is not affected by an executed operation, the model checker does not have to check this guard in the following state. Before applying model checking, B2Program extracts some static information from the model:

- For each operation, it extracts which variables are written.
- For each guard, it extracts which variables are read.
- For each invariant conjunct, it extracts which variables are read.

From this, we derive a table about which event can affect which guard and invariant: a guard or invariant p depends on operation *op* if there exists a variable that is read by p and written by *op*.

As B2Program stops as soon as an invariant violation is encountered, we do not have to cache each invariant's status; they must be true. The model checker only needs to know how a state is reached. When reaching a state s_2 from a state s_1 via an operation *op*, only invariants modified by *op* are checked. As for guards, the algorithm caches each guard's status for each visited state. Furthermore, the model checker copies the values of guards from s_1 which do not depend on *op*.

Parallelization. B2Program is also capable of multi-threaded model checking. Here, the model checker consists of a user-specified number of worker threads with one additional thread acting as a coordinator. Figure 7.2 shows the corresponding workflow, which is as follows: The coordinator takes the next state to be processed (depending on the search strategy), and assigns it to a worker thread. The worker checks the provided state for deadlocks and invariant violations, and computes the enabled operations. Successor states that have not been visited are then added to the list of queued states. The coordinator continues assigning tasks as long as there are unvisited states in the queue. Otherwise, the coordinator waits for a worker's notification. For this, a worker notifies the coordinator upon completion of a task when either (1) the global queue contains states for processing, or (2) the worker is the only one running at the moment. In case

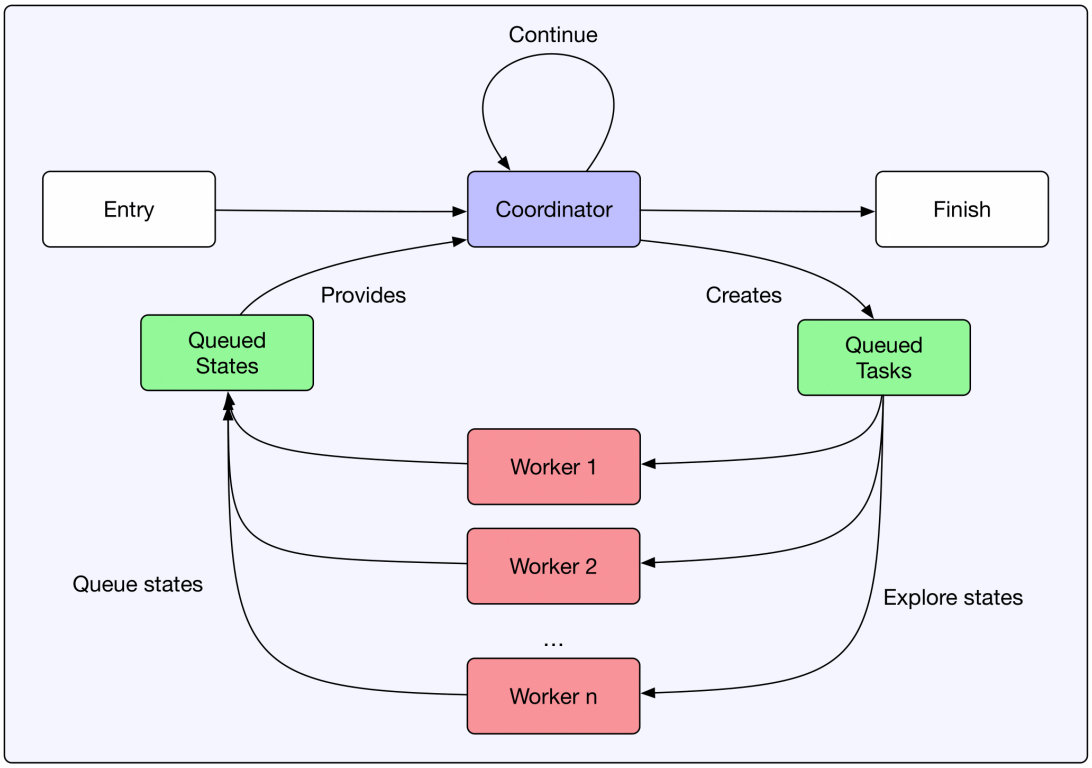


Figure 7.2.: Workflow: Multi-threaded Model Checking

(2), the coordinator can finish model checking, in case (1) it can (again) assign tasks to workers.

7.3. Limitations of High-level Code Generation

Below, we discuss the limitations of model checking with B2Program. Compared to existing B0 code generators, B2Program supports high-level constructs such as sets, relations, and quantified constructs like set comprehensions, quantified predicates, or lambda expressions. While B2Program is high-level compared to B0 code generators, it still has restrictions compared to ProB.

Currently, the required format for quantified constructs is quite restrictive. Indeed, quantified variables $v_1, \ldots, v_n$ must be constrained by a predicate P where the i-th conjunct of P must be constraining v_i (as described in previous work [233] to enable finite enumeration of v_i's possible values). E.g., assuming that z is a machine's state variable, $\{x, y \mid x \in 1..10 \land x \neq z \land y \in x..10\}$ must be rewritten to $\{x, y \mid x \in 1..10 \land y \in x..10 \land x \neq z\}$. This restriction will disappear in the future. TLC4B can cater for interleaved pruning predicates like $x \neq z$ while ProB has no restriction on the order of the conjuncts at all.

Comparing code generation for model checking with simulation, we have already discussed limitations regarding inner guards and non-deterministic constructs in Sec-

tion 7.2.1. Here, the user has to re-write the model to move guards and non-determinism to the top level of an operation. For now, constants must also be constrained to have a single possible value. (ProB does allow an arbitrary number of valuations for the constants.) `DEFINITIONS` must currently be inlined in the code by hand as well.

B2Program supports simple predicates using infinite sets on the right-hand side of $\in$, e.g., `x ∈ NATURAL`. Internally, the generated code checks whether `x` is greater than or equal to `0`. Slightly more complicated constructs such as `x ∈ NATURAL → NATURAL` are supported as well. Here, the generated code performs the same check on each element in the domain and the range. Additionally, it checks whether `x` is a total function. Note that the expression on the left-hand side of the operation is finite. Currently, B2Program also allows (partial and total) function operators including injection, surjection, and bijection together with $\in$. Nevertheless, we have disallowed constructs where it might be necessary to evaluate infinite sets or function operations explicitly. For example, we have disallowed the assignment of a variable to an infinite set like `x := NATURAL`. Furthermore, nested uses of infinite sets such as `x ∈ NATURAL → (NATURAL → NATURAL)` are not supported. Similarly, this work does not support assignment to function operations, e.g., `x := 1 ... m → 1 ... n`. The latter could be supported in the future, but the cardinality of those sets can grow very large.

On the technical side, there are also language-specific restrictions, e.g., the sum of B variables and constants must be less than 255 for generated Java code. To overcome this Java restriction, one would need to adapt the code generation technique to avoid creating large classes with too many parameters.[4]

However, as the next section shows, we can apply B2Program to a large number of examples. We also hope that many of the above restrictions will disappear in the future.

7.4. Empirical Evaluation of the Performance

This section evaluates the model checking performance compared to ProB, and TLC (via TLC4B). The generated Java model checking code is executed on *OpenJDK*.[5] The generated C++ code is compiled using *clang*[6] with `-O1`. For Java and C++, we benchmarked both *with* and *without* invariant and guard caching (cf. Section 7.2.2). We execute ProB in *1.12.0-nightly*,[7] and TLC4B in 2.06. Regarding ProB, we benchmark *with* and *without* operation re-use (a new technique described in [150]). Furthermore, we benchmark multi-threaded model checking (with six threads) for Java and C++ and compare the results with TLC. We have also measured the startup overhead time including parsing, translation, and compilation. The experiments in Table 7.2 (see Section 7.7), Table 7.3 (see Section 7.7), and Table 7.1 show the respective results for single-threaded, multi-threaded model checking, and the overhead. For single-threaded

[4] Note that TLC also has problems when the number of variables of a model increases, in terms of stack consumption and runtime degradation.

[5] 64-Bit Server VM (build 15+36-1562, mixed mode, sharing)

[6] Apple clang version 13.0.0 (clang-1300.0.29.30)

[7] Revision b6d1b600dbf06b7984dd2a1dd7403206cfd9d394

model checking, we show the bar chart relative to ProB (see Figure 7.3). For multi-threaded model checking, we show the bar charts relative to TLC (see Figure 7.4) and relative to the single-threaded speedups (see Figure 7.5). A more detailed overview and more benchmarks (including C++ with `-O2` optimization) are available in B2Program's repository. They are run on a MacBook Air with 8 GB of RAM and a 1.6 GHz Intel i5 processor with two cores. Each model checking benchmark is run ten times with a timeout of one hour, and afterward, the median runtime and the median memory consumption (maximum resident set size) are measured. Regarding the overhead, we measure the median runtime. We omit the C++ `-O2` benchmarks in this paper because the *clang++* compiler cannot optimize further for model checking.

The benchmarked models vary both in their complexity and in the focus of how they are modeled: Counter is a modified version of Lift from [233], consisting of operations to increment and decrement the counter between 0 and 1 000 000. It serves as a baseline benchmark for simple models with large state spaces.

The Volvo Cruise Controller uses mainly Boolean variables with many logical operations and assignments. The Landing Gear model (originally from Event-B [135]) also contains many logical operations and assignments, in addition to a large number of set operations.

Train and CAN Bus use set and relational operations. To keep the runtimes reasonable we benchmark a modified version [151] of the Train interlocking (with ten routes) from [5], where partial order reduction is applied manually. While ProB and TLC can be used directly, it is necessary to rewrite some constructs for B2Program. As ProB and TLC handle the original versions better, we benchmarked those for ProB and TLC.

We also benchmarked Nokia's NoTa (network on terminal architecture) model [183] which has many set operations. Here, it was necessary to rewrite the model to apply B2Program. The rewritten version leads to a reduced number of transitions, but does not affect the performance of ProB and TLC negatively. Compared to the other models, there are more power sets and quantified constructs. Also, its invariant contains more involved function type checks.

sort_1000 is a B model (originally from Event-B [200]) of an insertion sorting algorithm with 1000 elements.

As an opposite to the Counter model, we benchmark a B model of the N-Queens problem with $N = 4$. The model contains a B operation to solve the N-Queens puzzle and the state space will consist of all solutions to the puzzle (i.e., 2 for $N = 4$). While ProB and TLC apply to the original model, it is necessary to rewrite the model for B2Program. Similar to Train, we thus benchmarked the original model for ProB and TLC, and the rewritten model for B2Program.

In the previous work [233], code generation to Java and C++ for simulation was up to one or two magnitudes faster than ProB. Now, one can see in Figure 7.3 that this is still the case for several models when model checking compared to ProB *without* the new operation caching feature. For the Cruise Controller, the runtimes are similar and N-Queens is the only model where ProB outperforms B2Program in all configurations. This is obviously due to ProB's constraint solving capabilities. With the operation caching feature [150], the situation changes somewhat. ProB is now faster than generated C++ model checkers for NoTa and faster than the Java model checkers for Train. The

speedups obtained by the generated model checkers are still significant, but less than a factor of two for a few models.

Counter is the only model where the generated code strongly outperforms ProB (up to two magnitudes). One can also see that for some models, it is necessary to choose the right setting to outperform ProB with operation reuse. For example, model checking Train with B2Program only leads to a better runtime, when C++ with caching is chosen.

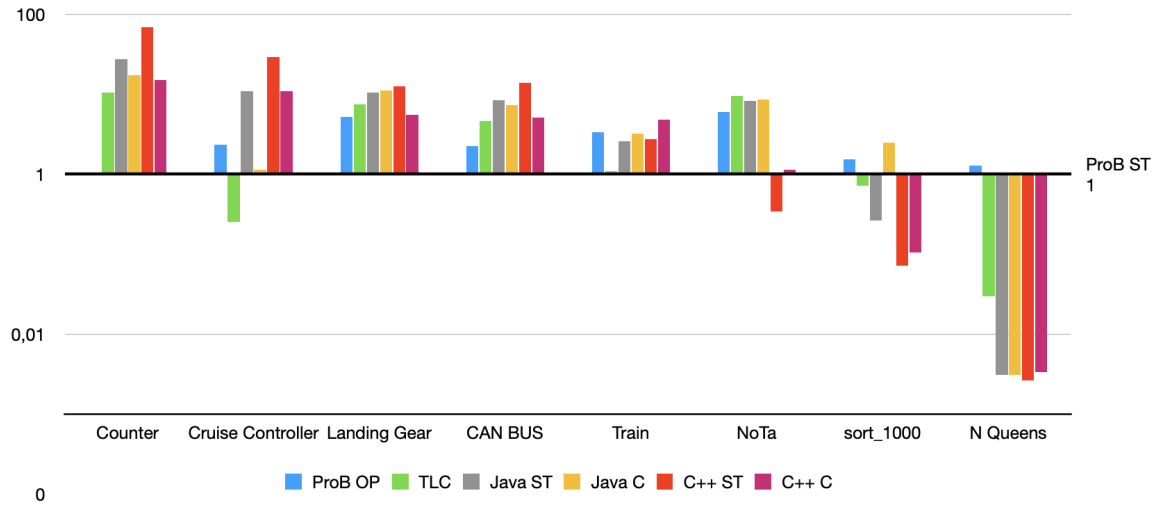


Figure 7.3.: Single-threaded Speedups relative to ProB ST as Bar Charts; ST = Standard, OP = Operation Reuse, C = Caching

Regarding model checking with TLC, there are models where ProB performs better and vice versa. Code generation to Java and C++ makes it possible to outperform TLC for most benchmarks (also for multi-threaded model checking as shown in Figure 7.4). For NoTa, the generated Java model checkers have a similar performance to TLC, while C++ is much slower.

TLC can find all solutions for N-Queens faster than the generated model checkers, but slower than ProB. Similar to B2Program, TLC also lacks constraint solving features. The reason why TLC possibly performs better could be the restrictions of constraining predicates as discussed in Section 7.3. For both ProB and TLC, the generated code only performs better for sort_1000 if Java is chosen together with caching. In particular, the translation of the invariant generates large sets which could be avoided by caching successfully.

As shown in Figure 7.5, parallelization makes it possible to improve the performance further for most benchmarks. For sort_1000 and Train, the additional speedup is around two. In some cases, e.g. Counter, CAN BUS or NoTa, the overhead results in a slowdown. Regarding the first two machines, this overhead can also be seen in TLC.

The implemented caching features in B2Program lead to overhead for most benchmarks. Nevertheless, a speedup could also be achieved for some models, e.g., Train. The reason

could be the complex invariants and guards in both models. For sort_1000, caching only improves Java's runtime. One can also see (in Table 7.2 and Table 7.3 in Section 7.7) that our caching implementation increases memory usage significantly. In contrast, ProB manages to keep memory consumption low when using operation reuse together with state compression [150]. A significantly increased memory consumption only occurs for Train. Overall, the operation reuse feature in ProB is not only more complex, but is also much more efficient than the caching technique implemented in this work.

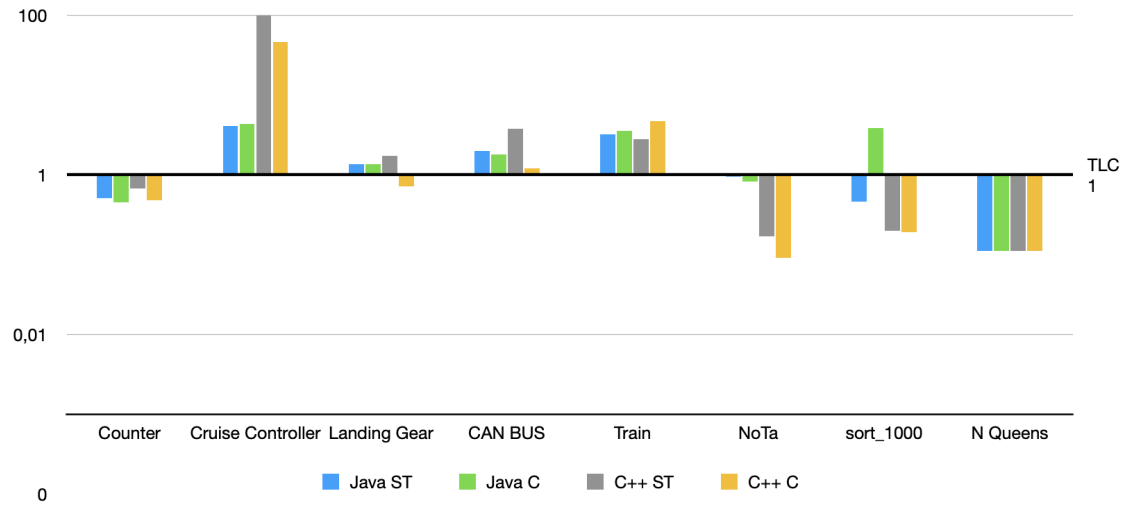


Figure 7.4.: Multi-threaded (6 Threads) Speedups relative to TLC as Bar Charts; ST = Standard, C = Caching

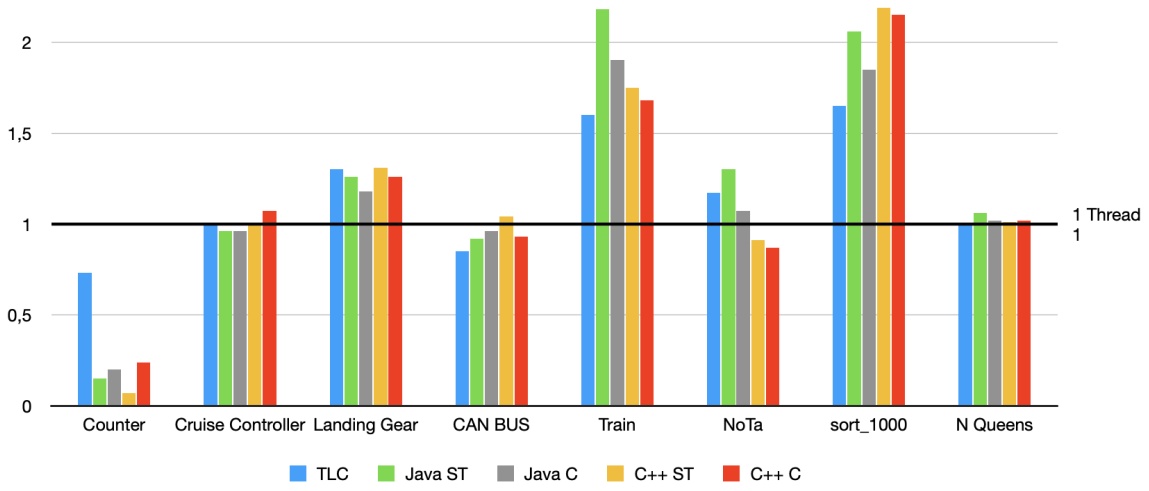


Figure 7.5.: Multi-threaded Speedups relative to Single-threaded Speedups as Bar Charts; ST = Standard, C = Caching

Table 7.1.: Startup Overhead in Seconds (including Parsing, Translation and Compilation) of ProB, TLC, and Generated Code in Java, and C++

Counter	ProB	TLC	Java	C++ -O1	C++ -O2
Parsing/Translation	2.52	2.95	1.32	1.35	1.35
Compiling	-	-	2.15	6.22	7.31
Cruise Controller (Volvo)	ProB	TLC	Java	C++ -O1	C++ -O2
Parsing/Translation	3.05	3.97	2.34	2.52	2.52
Compiling	-	-	3.33	20.84	37.48
Landing Gear [135]	ProB	TLC	Java	C++ -O1	C++ -O2
Parsing/Translation	3.04	3.91	2.53	2.74	2.74
Compiling	-	-	3.61	26.01	42.3
CAN BUS (J. Colley)	ProB	TLC	Java	C++ -O1	C++ -O2
Parsing/Translation	2.85	3.45	1.87	2.05	2.05
Compiling	-	-	3.05	16.17	23.45
Train (ten routes) [151, 5]	ProB	TLC	Java	C++ -O1	C++ -O2
Parsing/Translation	2.9	3.59	2.07	2.21	2.21
Compiling	-	-	2.84	15.52	20.05
NoTa [183]	ProB	TLC	Java	C++ -O1	C++ -O2
Parsing/Translation	2.94	3.65	2.14	2.32	2.32
Compiling	-	-	3.03	29.23	39.76
sort_1000 [200]	ProB	TLC	Java	C++ -O1	C++ -O2
Parsing/Translation	2.61	3.1	1.53	1.58	1.58
Compiling	-	-	2.26	8.51	10.37
N-Queens with N=4	ProB	TLC	Java	C++ -O1	C++ -O2
Parsing/Translation	2.61	3.07	1.45	1.5	1.5
Compiling	-	-	2.2	9.11	11.33

Table 7.1 shows a small startup overhead for ProB, TLC, and B2Program with Java. Note that the ProB and Java times also contain the startup time of the ProB CLI, and the JVM (twice: generating code with B2Program, and compiling) respectively. The JVM's startup time is not included in the TLC overhead times.

The C++ startup time, however, can be considerable. In some cases, the compilation time is greater than the model checking time. Thus, in a setting where one wants to repeatedly modify and re-verify a model, the C++ overhead would be prohibitive. B2Program with C++ is thus only effective for long-running model checking such as Train. In the future, one could pre-compile some libraries to reduce the C++ compilation times.

7.5. More Related Work

The present work enabled us to make use of some of ProB's validation tests, namely the tests where counter-examples to over 500 mathematical laws are sought after using model checking [27]. This indeed uncovered several issues in B2Program, and helped us improve the stability of the core of B2Program.

Code Generators. There are several code generators for various formalisms such as Event-B [172, 49, 200, 69, 80], ASM [33], or VDM [119]. As far as we know, none of these code generators supports model checking (yet).

Model Checkers. We have already compared B2Program with the explicit-state model checkers ProB [152], and TLC [246] via TLC4B [100].

In the following, we add a few more points. In general, there are some limitations compared to ProB, as discussed in Section 7.3. Furthermore, B2Program only supports B, while ProB also supports Event-B, Z, TLA + CSP, or CSP || B. In the context of explicit-state model checking ProB supports various features such as state compression, efficient state hashing, use of proof information, partial order reduction, partial guard evaluation, invariant and operation caching. Still, B2Program leads to a faster runtime for most benchmarks thanks to custom model checkers without interpretation overhead. But especially, the operation reuse feature [150] makes it possible for ProB to keep up with TLC's and B2Program's performance. ProB has the advantage that model checking counter-examples are represented as traces and can be loaded into the animator for inspection. Furthermore, ProB also supports other techniques such as LTL model checking which are not available to B2Program yet. Again, ProB is also capable of visualization features, e.g., visualizing the state space.

As discussed in Section 7.3, parameters or quantified variables must be enumerated for B2Program in the exact order they are defined first, before additional predicates can be checked. In contrast, TLC allows interleaving of pruning predicates. Unlike TLC, B2Program supports code generation for sequential substitutions. The parallel model checking approach in TLC is implemented similar to B2Program. Within TLC, a worker always takes a state before processing it. A main feature of TLC is the efficient storage of the state space on disk. As result, TLC can handle very large state spaces, while the generated Java model checking code with B2Program depends on the JVM's memory. Nevertheless, there are also state collisions in TLC, which can lead to erroneous results, although they occur rarely. Unlike TLC, collisions between states are handled. When analyzing the performance (for both single-threaded and multi-threaded model checking), we have encountered that model checking with B2Program leads to a speedup compared to TLC.

Another toolset is LTSmin [120] which supports explicit-state model checking and LTL model checking, as well as symbolic model checking. LTSmin also supports parallelization and partial order reduction. As presented by Körner et al. [127] and Bendisposto et al. [26], LTSmin was integrated into ProB, leading to a significant speedup. It, however,

also has some drawbacks (see [150]) and the predicates and operations themselves are still computed by the ProB interpreter.

PyB is a second tool-chain of ProB, which can also model check B models. Similar to TLC and this work, PyB lacks constraint solving. Set operations in PyB were relatively slow, while integer operations can be applied efficiently [244].

The idea of generating code for model checking has already been implemented in SPIN [109]. Here, a problem-specific model checker in C is generated from a Promela model. SPIN supports features such as state compression, bitstate hashing, partial order reduction, and LTL model checking. SPIN is a very efficient explicit-state model checker, but operates on a much lower-level language.

JavaPathfinder [169] is a model checker which runs executable Java bytecode on the JVM to check a program for race conditions and deadlocks. To cover all possible execution paths of a Java program, JavaPathfinder is implemented with backtracking features. B2Program currently only supports non-determinism for top-level guards and preconditions.

There are also bounded model checkers for C and Java, named CBMC [130] and JBMC [53] respectively. Both model checkers are capable of verifying memory safety, checking for exceptions, checking for various variants of undefined behavior, and checking user-specified assertions. In contrast to CBMC and JBMC, the main purpose of our work is to generate Java and C++ code for verification, not verifying Java and C++ programs themselves.

7.6. Conclusion and Future Work

In this work, we extended the high-level B code generator B2Program to generate specialized model checkers. One goal was to provide a baseline for model checking benchmarks, using tailored model checkers, compiled for each B model. The hope was to achieve fast model checking, exceeding TLC's (interpreter-based) performance and overcoming some of the limitations.

One major challenge was to adapt B2Program so that it produced *all* enabled operations (and not just one). This was achieved for top-level guards, parameters and preconditions but not yet for nested guards or preconditions and some nested non-deterministic constructs. In general, we have discovered some limitations of B2Program compared to ProB. In particular, some B constructs are too high-level for code generation to Java and C++. Furthermore, ProB also supports more formalisms than B2Program, which only supports B. The TLC4B approach shares some limitations of B2Program, e.g., the need for explicit enumeration predicates. There are, however, limitations of TLC which can be handled by B2Program and vice versa.

Our empirical evaluation has provided some interesting insights. We found out that code generation to Java and C++ leads to a speedup up to one magnitude wrt. ProB for certain interesting benchmarks. However, there are also models where ProB performs better, either due to its constraint solving capabilities or due to the recent operation caching technique [150]. In contrast, the invariant and guard caching in B2Program

only improve performance in some cases (for very complex invariants or guards) and its overhead is not worthwhile in general. Parallelization improves the generated code's performance for most experiments. Code generation to Java and C++ outperforms TLC for most benchmarks. The fact that TLC performs better for some models could be caused by the restrictions of B2Program. Indeed, these restrictions also mean that ProB will quite often perform significantly better than B2Program for (original) models which have not been adapted for B2Program.

In future, we would like to remove the above-mentioned restrictions of B2Program. We will also improve the feedback, e.g., show which parts of the invariant are violated, or more information about the coverage. Furthermore, it would also be possible to improve the performance, e.g., by improving the state space's storage, or caching features (such as presented by Leuschel [150]). One could also generate code for existing model checkers, such as LTSmin, SPIN, SpinJA [58] (integrated into LTSmin [225]), JBMC [53], or JavaPathfinder [169]. This would enable features, such as LTL or symbolic model checking, without re-implementing them. Another main issue for the future is model checking non-deterministic parts deep in the specifications. To address non-determinism, B2Program could also be extended to target Prolog.

Acknowledgements. We would like to thank Florian Mager and Klaus Sausen. They have been working on a student's model checking project from which some ideas have emerged for this work. We would also like to thank Lucas Döring, who is currently improving B2Program's model checking performance. We would also like to thank Sebastian Stock for proofreading this paper and anonymous referees for their feedback.

7.7. Appendix: Benchmarks

Table 7.2.: **Single-threaded** Runtimes of ProB, TLC, and Generated Code in Java, and C++ (Compiled with -O1) in Seconds with State Space Size, Speed-Up Relative to ProB, Memory Usage in KB, OP = Operation Reuse, ST = Standard, C = Caching

Counter		ProB OP	ProB ST	TLC	Java ST	Java C	C++ ST	C++ C
(1 000 001 states,	Runtime	90.06	87.98	8.52	3.24	5.16	1.29	5.88
2 000 001 transitions)	Speed-up	1	1.02	10.67	27.84	17.47	70.08	15.33
	Memory	1 151 604	1 151 556	325 420	421 034	654 880	217 920	878 754
Cruise Controller		ProB OP	ProB ST	TLC	Java ST	Java C	C++ ST	C++ C
(Volvo,	Runtime	0.75	1.74	6.89	1.6	1.53	0.06	0.16
1360 states,	Speed-up	1	0.11	0.15	0.47	0.49	12.5	4.69
26 149 transitions)	Memory	174 954	174 247	172 016	121 832	113 110	2722	10 912
Landing		ProB OP	ProB ST	TLC	Java ST	Java C	C++ ST	C++ C
Gear [135]	Runtime	36.85	188.87	25.68	18.23	17.22	15.1	34.38
(131 328 states,	Speed-up	1	0.2	1.51	2.12	2.44	2.56	1.12
884 369 transitions)	Memory	476 783	469 995	681 308	751 508	985 684	186 736	1 053 604
CAN BUS		ProB OP	ProB ST	TLC	Java ST	Java C	C++ ST	C++ C
(J. Colley,	Runtime	23.13	52.11	11.42	6.23	7.21	3.77	10.29
132 599 states,	Speed-up	1	0.44	2.03	3.71	3.2	6.14	2.25
340 266 transitions)	Memory	353 338	352 125	461 096	450 596	562 440	196 762	677 544
Train [151, 5]		ProB OP	ProB ST	TLC	Java ST	Java C	C++ ST	C++ C
(with ten routes,	Runtime	776.81	2564.03	2373.16	1004.45	799.37	940.32	533.78
672 174 states,	Speed-up	1	0.3	0.33	0.77	0.97	0.83	1.46
2 244 486 transitions)	Memory	2 995 244	1 278 929	896 422	1 267 960	2 317 640	1 228 082	2 995 064
NoTa [183]		ProB OP	ProB ST	TLC	Java ST	Java C	C++ ST	C++ C
(80 718 states	Runtime	29.89	178.82	18.78	21.89	20.9	88.51	157.8
1 797 353 transitions)	Speed-up	1	0.17	1.59	1.37	1.43	0.34	0.19
	Memory	947 413	946 857	883 470	974 294	1 063 392	189 306	993 818
sort_1000 [200]		ProB OP	ProB ST	TLC	Java ST	Java C	C++ ST	C++ C
(500 501 states,	Runtime	234.97	359.23	505.1	1365.79	146.72	3288.73	3468.77
500 502 transitions)	Speed-up	1	0.65	0.47	0.17	1.6	0.07	0.07
	Memory	833 697	602 163	374 906	521 224	1 314 720	303 293	947 840
N-Queens		ProB OP	ProB ST	TLC	Java ST	Java C	C++ ST	C++ C
with N=4	Runtime	0.15	0.19	6.46	61.97	61.19	57.05	57.02
(4 states	Speed-up	1	0.79	0.02	0.002	0.002	0.003	0.003
6 transitions)	Memory	166 608	166 574	170 972	351 168	349 608	48 892	48 886

Table 7.3.: **Multi-threaded** (6 Threads) Runtimes of TLC, and Generated Code in Java, and C++ (Compiled with -O1) in Seconds with State Space Size, Speed-Up Relative to TLC and Relative to Single-Threaded, Memory Usage in KB, TH = Thread, ST = Standard, C = Caching

Counter		TLC	Java ST	Java C	C++ ST	C++ C
(1 000 001 states,	Speed-up to TLC	1	0.51	0.45	0.67	0.48
2 000 001 transitions)	Speed-up to 1 TH	0.73	0.15	0.2	0.07	0.24
	Memory	294 664	398 248	728 514	218 086	878 910
Cruise Controller		TLC	Java ST	Java C	C++ ST	C++ C
(Volvo,	Speed-up to TLC	1	4.1	4.3	114.33	45.73
1360 states,	Speed-up to 1 TH	1	0.96	0.96	1	1.07
26 149 transitions)	Memory	172 032	147 498	142 246	3048	10 994
Landing		TLC	Java ST	Java C	C++ ST	C++ C
Gear [135]	Speed-up to TLC	1	1.36	1.36	1.71	0.72
(131 328 states,	Speed-up to 1 TH	1.3	1.26	1.18	1.31	1.26
884 369 transitions)	Memory	808 976	954 956	1 179 980	195 566	1 056 952
CAN BUS		TLC	Java ST	Java C	C++ ST	C++ C
(J. Colley,	Speed-up to TLC	1	1.98	1.78	3.72	1.21
132 599 states,	Speed-up to 1 TH	0.85	0.92	0.96	1.04	0.93
340 266 transitions)	Memory	337 404	498 644	574 292	204 528	687 058
Train [151, 5]		TLC	Java ST	Java C	C++ ST	C++ C
(with ten routes,	Speed-up to TLC	1	3.22	3.51	2.76	4.65
672 174 states,	Speed-up to 1 TH	1.6	2.18	1.9	1.75	1.68
2 244 486 transitions)	Memory	1 077 022	1 456 166	2 340 918	1 261 254	3 164 336
NoTa [183]		TLC	Java ST	Java C	C++ ST	C++ C
(80 718 states	Speed-up to TLC	1	0.95	0.82	0.17	0.09
1 797 353 transitions)	Speed-up to 1 TH	1.17	1.3	1.07	0.91	0.87
	Memory	898 580	1 100 700	1 138 802	220 648	1 035 806
sort_1000 [200]		TLC	Java ST	Java C	C++ ST	C++ C
(500 501 states,	Speed-up to TLC	1	0.46	3.84	0.2	0.19
500 502 transitions)	Speed-up to 1 TH	1.65	2.06	1.85	2.19	2.15
	Memory	503 360	520 850	1 894 494	304 048	948 392
N-Queens		TLC	Java ST	Java C	C++ ST	C++ C
with N=4	Speed-up to TLC	1	0.11	0.11	0.11	0.11
(4 states	Speed-up to 1 TH	1.0	1.06	1.02	1.01	1.02
6 transitions)	Memory	170 934	360 534	350 706	49 026	48 920

8. Generating Interactive Documents for Domain-Specific Validation of Formal Models

Abstract. Especially in industrial applications of formal modeling, *validation* is as important as *verification*. Thus, it is important to integrate the stakeholders' and the domain experts' feedback as early as possible. In this work, we propose two approaches to enable this: (1) a static export of an animation trace into a single HTML file, and (2) a dynamic export of a classical B model as an interactive HTML document, both based on domain-specific visualizations. For the second approach, we extend the high-level code generator B2Program by JavaScript, and integrate VisB visualizations alongside SimB simulations with timing, probabilistic and interactive elements. An important aspect of this work is to ease communication between modelers and domain experts. This is achieved by implementing features to run simulations, sharing animated traces with descriptions, and giving feedback to each other. This work also evaluates the performance of the generated JavaScript code compared with existing approaches with Java and C++ code generation as well as the animator, constraint solver, and model checker ProB.

Keywords. Code Generation, Validation, B Method, Domain-Specific, Interactive, Visualization

8.1. Introduction and Motivation

Verification shows the correctness of software, thus tackling the question "Are we building the software correctly?" [116]. During the verification process, it might indicate errors. Just as important is validation, which checks whether the stakeholders' requirements are fulfilled and thus tackles the question "Are we building the right software?" [116].

An important aspect of validation is the dialogue between modelers and stakeholders or domain experts. The latter are usually not familiar with the formal method and notation, while the modeler only has limited knowledge about the domain. Animation, simulation, and visualization of scenarios are important enabling technologies: a domain expert can grasp the behavior of a model by looking at visualizations, without having to understand the underlying mathematical notation. Even for modelers, visualization is important; for instance, some errors are immediately obvious in a visual rendering, while they can remain hidden within the mathematical, textual counterpart (see various case

studies, e.g., Vehicle's Light System [154], Landing Gear [135], ETCS Hybrid Level 3 [101], Air Traffic Control Software [88]).

In this paper, we tackle one further hurdle that domain experts or stakeholders face: in addition to lacking knowledge and experience with formal notations, they typically also lack the knowledge to drive the particular tool, or possibly even install it. Even when a domain expert successfully installs such a tool, they have to work with features, techniques, and notations they usually are not familiar with. In this article, we implement two solutions to this:

- a static export of an animation trace into a single HTML file, that can be sent by email and rendered in any current browser. This export is available for all models supported by the animator, constraint solver, and model checker ProB [152, 153], and enables the user to navigate within the trace.

- a dynamic export of a classical B model (and optionally pre-configured traces), to an HTML document which can also be rendered in a current browser. This export uses the high-level B code generator B2Program [233] which is extended by JavaScript. While not applicable to all models, the export is completely dynamic: a user can freely navigate the model's state space, not just one pre-configured trace. Furthermore, a user can even run various simulations automatically and modify descriptions of traces. The dynamic export includes a domain-specific VisB visualization [243] and supports timed probabilistic simulation with SimB [237] (including user interaction [236]). This allows a domain expert to interact with a prototype in VisB, and experience probabilistic and timing behavior.

In both solutions, one just needs to open a browser and the HTML document. A domain expert can then interact with the trace or model with a domain-specific visualization and familiar techniques.

First, we give some background in Section 8.2. We then present the validation workflow in Section 8.3. Section 8.4 describes the static export of an animation trace into a single HTML file. In Section 8.5, we describe a dynamic export of a classical B model to an interactive HTML document. Section 8.6 demonstrates how this work improves the validation of requirements by domain experts, and communication between modelers and domain experts. We also evaluate and discuss the applicability of the dynamic export, including the performance in Section 8.7. Finally, we compare our work with related work in Section 8.8, and conclude in Section 8.9.

This paper is an extended version of the FMICS 2022 paper [234]. For this, we implemented new features such as (1) timed probabilistic simulation with SimB [237], (2) interactive simulation [236], and (3) model checking support [231] for JavaScript for the performance analysis. In this extension, we also allow domain experts to give more feedback on execution traces by writing description texts. Furthermore, we describe the domain experts' validation process and the generation of certain GUI components in more detail. We have also demonstrated the effectiveness of the SimB features and the feedback through descriptions using the existing case studies.

```
1:  MACHINE Sieve
2:  VARIABLES numbers, cur, limit
3:  INVARIANT
4:    numbers <: INTEGER & cur:NATURAL1 & limit:NATURAL1
5:  INITIALISATION numbers := {} || cur := 1 || limit := 1
6:  OPERATIONS
7:    StartSieve(lim) = PRE cur=1 & lim > MINLIM & lim <= MAXINT THEN
8:      numbers := 2..lim ||
9:      cur := 2 ||
10:     limit := lim
11:   END;
12:
13:   prime <-- TreatNumber(cc) =
14:   PRE cc=cur & cur>1 & cur*cur<= limit THEN
15:     IF cc:numbers THEN
16:        numbers := numbers - ran(%n.(n:cur..limit/cur|cur*n))
17:       || prime := TRUE
18:     ELSE
19:        prime := FALSE
20:     END ||
21:     cur := cur+1
22:   END;
23:
24:   r <-- Finish = PRE cur*cur>limit THEN
25:     cur := 1 || r := card(numbers)
26:   END
27: END
```

Listing 8.1. Example of Prime Number Sieve in B

8.2. Background

The B method was introduced by Jean-Raymond Abrial, and is a formal method for specifying and verifying software systems [4]. The B method includes the formal B modeling language which bases on *first-order logic* and *set theory*. Within the B language, a component is a *machine* which contains *constants*, *variables*, *sets*, together with the *initialization*, and *operations*. While *variables* represent the model's current state, the *initialization* and *operations* can be defined with *substitutions* (aka statements) which change the *variables* and thus also the state. Usually, an operation consists of a *guard* and a *substitution* which means that the *substitution* is applied when the *guard* is true. Furthermore, each model contains an *invariant* which is a predicate that must always be true.

Listing 8.1 shows an example of Eratosthenes sieve modeled in B. It has three variables: `numbers` (the candidates for prime numbers), `cur` (the current number being processed) and a `limit` for stopping the algorithm. The model has three operations, one for starting the sieve, one for treating the next number and one for finishing. As shown in Listing 8.1, one can see that the model consists of arithmetic, logical, and set operations.

The formal B language is a refinement-based modeling language. This means that

the development chain consists of multiple machines which are gradually refined by further details. If the modeler intends to generate code for embedded systems from this, then at some point it will have to be refined to B0, which is a subset of B. B0 contains constructs from B which are at the implementation level. Those constructs are close to imperative programming languages, such as `WHILE` loops or `IF-THEN-ELSE` substitutions. The use of sets and relations is only restricted here. For example, one can use elements of enumerated sets or define total functions for arrays. Set and relation operators, however, are not allowed. High-level constructs such as nondeterminism or set definitions are also no longer allowed.

The main application fields of the B method are safety-critical systems. For example, railway systems such as the Paris Métro Line 14 [65], and the New York Canarsie Line [71] have been modeled and verified with the B method, and afterward, code generation has been applied. Another use case of the B model in the railway domain is the ETCS Hybrid Level 3 where formal B models are used at runtime [101]. Moreover, the B method has also been used for other industrial-motivated case studies in safety-critical domains, such as automotive [154], and aviation [135].

ProB [152, 153] is an animator, constraint solver, and model checker for formal methods, such as B, Event-B, Z, TLA+, CSP, and Alloy. ProB's core which also includes interpreters for various formalisms (including B) is implemented in SICStus Prolog [48]. Consequently, B models are interpreted during execution, animation and model checking.

ProB2-UI [25] is a JavaFX-based graphical user interface which has been developed on top of ProB by using ProB's Java API [126]. The following features of ProB2-UI are especially relevant for this work:

- the persistent storage and replay of traces [25],
- domain-specific VisB visualization [243],
- timed probabilistic SimB simulation [237] with user interaction [236].

ProB2-UI supports many techniques to create traces: animation (also via interactions in VisB), test case generation, model checking, or simulation with SimB. Once a trace is created, it is possible to add description text corresponding to each step. Later, a trace can be replayed (1) to check if the scenario is still feasible and (2) to perform additional checks on a trace. The corresponding description text helps to communicate with domain experts.

VisB [243] is a component of ProB to create interactive visualizations of formal models using SVG images and a glue file. The VisB glue file defines the main SVG image, as well as observers and click listeners which link the graphical elements with the model's state. Using VisB, a user can view the model's current state graphically, and execute operations by clicking on visual elements. An overview (also proposed in Figure 1 of [243]) is shown in Figure 8.1. Many features have been added in response to feedback from academic and industrial uses since VisB's original publication [243]. New features include iterators for groups of related SVG objects, multiple click events for SVG objects, dynamic SVG object creation, and SVG class manipulation for hovers. Furthermore,

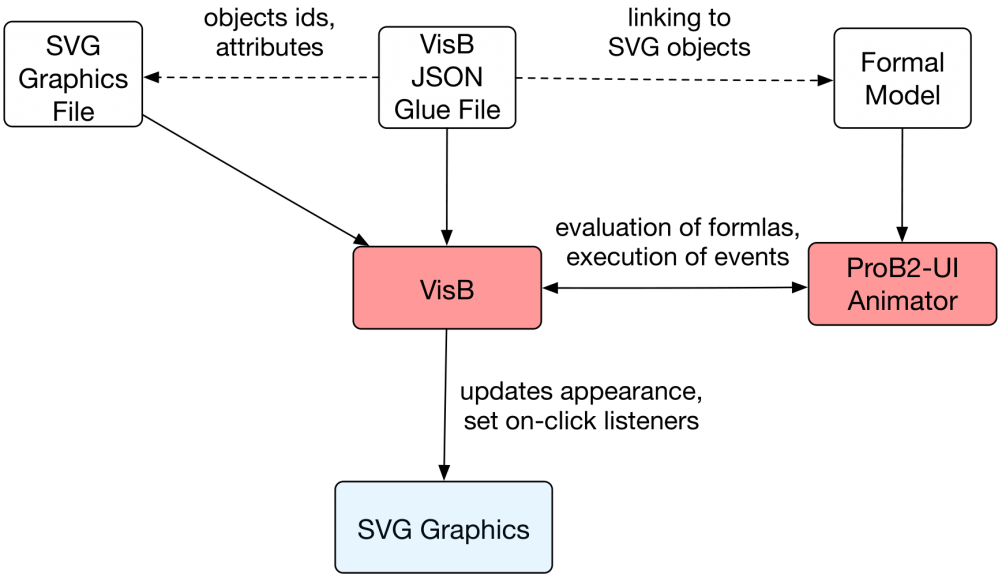


Figure 8.1.: Architecture of VisB and ProB2-UI (also proposed in Figure 1 of [243]); idea related to Model-View-Controller Pattern (MVC) [239]; SVG graphics file and VisB glue file are loaded in VisB, while formal model is loaded in ProB2-UI's animator; The VisB glue file connects the SVG graphics with the formal model; ProB2-UI's animator is used to evaluate formulas and execute events affecting the SVG objects' appearances; formal model events can also be executed via VisB.

VisB's core has been re-implemented in Prolog and integrated into ProB's core. Thus, VisB can now be used from ProB's command-line interface directly (without ProB2-UI [25]).

SimB [237] is a simulator for formal models which is part of ProB2-UI [25]. Using SimB, a modeler can annotate a formal model with timing and probabilistic elements for simulation. These annotations take the form of an *activation diagram* which describes how events trigger each other with delays and probabilities. As a result, SimB helps to validate requirements with timing and probabilistic aspects. More recently, SimB has been extended by a feature called *interactive simulation*, which allows user interaction to trigger a system response in a real-time simulation [236]. *Interactive simulation* can also be linked to responding to manual interaction in VisB visualisations. This helps a user or domain experts to validate user requirements more easily, as user interaction and system reaction can be better understood. In general, SimB helps to create prototypes for formal models with timing, probabilistic, and interactive behavior, emulating real-world behavior. For example, in this work, we used a visualization of a user interface for a vehicle's light system [154].

B2Program [233] is a code generator for high-level B models, which targets Java, C++, Python, Rust [66], and also TypeScript/JavaScript now. Unlike other B code

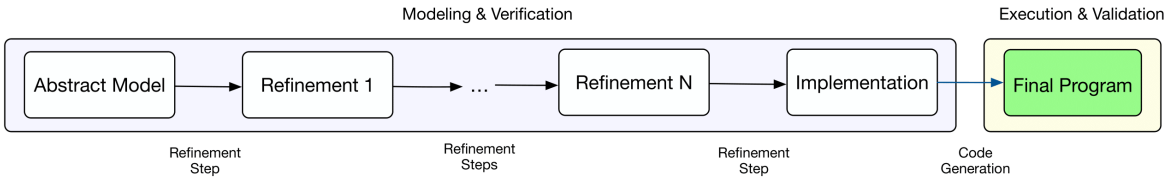


Figure 8.2.: Typical Formal Methods Workflow with Refinement; each refinement step adds more detail (represented by events, variables, etc.) to the previous abstraction level; the final refinement step refines the model to B0 from which low-level B0 code generators are applied, e.g., for usage in embedded systems.

generators, the model does not have to be refined to B's low-level subset B0. Instead, B2Program enables code generation from a formal B model at various abstraction levels for validation and demonstration purposes. This also means that B2Program allows code generation for constructs such as set operation, set comprehensions, relation operations, non-determinism, etc. Consequently, B2Program cannot be used for embedded systems because memory consumption cannot be verified for these constructs, especially due to the use of external libraries. B2Program also supports code generation of *specialized* model checkers [1] for a machine [231]. The generated code for model checking builds up the entire state space to check for invariant violations and deadlocks. To explore the complete state space, this code generates functions to compute all outgoing transitions, and thus all succeeding states. Those functions are invoked for each explored state until they cover the complete state space. B2Program is implemented using the StringTemplate [188] engine which allows targeting multiple languages with a single code generator. This is achieved by mapping each construct to a template which is rendered to the target code.

8.3. Validation Workflow

In the following, we compare the typical formal methods workflow with the one that is enabled in this research, i.e., by code generation for validation.

Figure 8.2 shows a typical formal methods workflow with refinement: A system or software is modeled step-by-step until all requirements are encoded. Furthermore, the model is refined until reaching an implementable subset of the modeling language (e.g. B0 in the B method). Each development step of the model is verified by provers such as AtelierB [51], or by model checkers such as ProB. After finishing the modeling process, a low-level code generator (e.g. an AtelierB B0 code generator) is applied to generate the final program from a verified model.

[1] Model checking is a technique that checks whether a system (modeled by a formal model) meets a certain specification, i.e., property. To do this, model checking computes all possible states and execution paths of the system. There are exhaustive approaches where the entire system is executed in all possible states, and symbolic approaches which over-approximate the state space [18].

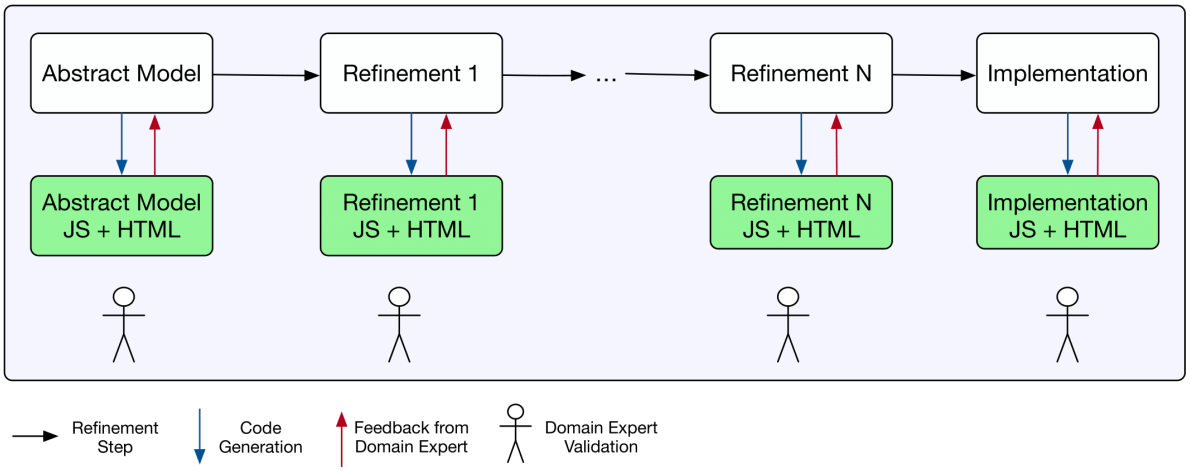


Figure 8.3.: Workflow: Code Generation for Validation with Refinement; each refinement step adds more detail (represented by events, variables, etc.) to the previous abstraction level; code generation can be applied at each abstraction level for validation purposes; high-level constructs are supported for code generation, but memory usage cannot be verified and thus usage in embedded systems is not possible.

A disadvantage of this typical workflow is that software is often validated too late during the development process, possibly after generating the final code. Figure 8.3 describes the approach followed by this work: We extend the high-level B code generator B2Program [233] by JavaScript generation and supporting validation techniques such as animation, trace replay, VisB visualizations, and SimB simulations. In particular, an HTML document is generated, supporting early-stage validation with the aforementioned techniques by a domain expert. As a result, domain experts are integrated into the development process at an early stage.

While Figure 8.3 is also feasible with existing animators like ProB, our approach enables communication via “interactive validation documents”, where the model’s formal aspects are hidden and no formal methods tool has to be installed by the domain expert.

As a simple example of a refinement-based development approach, let us consider a lift which is modeled as follows: the abstract level model’s the lift’s movement, the first refinement models the doors, and the second refinement introduces the lift’s buttons. According to Figure 8.2, a modeler can then refine further to B0 to apply code generation for the embedded system to be used in a real lift. Validation would then be applied at the final stage when code is already generated. Following our approach, as shown in Figure 8.3, we can generate prototypes for validation (rather than embedded systems) for each refinement level when developing the lift. The purpose of code generation in this work is to enable us to check whether requirements have been implemented correctly in each, especially in earlier development stages. A domain expert can then already inspect whether the lift’s movement is correctly modeled at the abstract level.

8.4. Static VisB HTML Export

In this section, we present another new feature of VisB to export a trace as a standalone HTML file containing the visualization of the entire trace. This approach is supported by all formalisms in ProB. The trace can either be constructed interactively in the animator or automatically by other techniques such as test case generation, model checking, or simulation. The HTML file enables the user to navigate the trace, and inspect the visualization of each state in the trace, without installing ProB. The model's variables and constant values are also accessible. Furthermore, the trace can be replayed automatically at different speeds. An example export can be seen in Figure 8.4.[2]

This feature has been used for the communication of modelers with domain experts, e.g., in follow-on projects of the ETCS Hybrid Level 3 [101]. In particular, we (as modelers) animated traces which contain critical behavior. Those were traces we animated to validate important behaviors, or traces where we suspected errors. We then created static exports for these traces, and sent them to domain experts. The domain experts were then able to open them in the browser directly, and give feedback via E-Mail. With the dynamic export (later explained in Section 8.5), domain experts can provide feedback as description texts into the traces directly.

When exporting the trace to an HTML file, a JavaScript function is generated for each state, hard-coding the SVG objects' changed attributes. Listing 8.3 shows parts of the function that is generated for the state shown in Figure 8.4. Focusing on the VisB item for the SVG object `occupied_ttd_polygon` (see Listing 8.2), one can see its hard-coded value for the state. When a domain expert steps through the trace, the visualization is updated according to the current state by executing the corresponding function. Figure 8.4 also contains meta-information. Thus, a stored HTML trace is also a standalone snapshot of the model. One can later compare the stored visualization and variables with the current model.

```
1: {
2:  "id":"occupied_ttd_polygon",
3:  "attr":"points",
4:  "value":"svg_set_polygon(OCC_TE,100.0/real(TrackElementNumber+1),100.0,2.0)"
5: }
```

Listing 8.2. VisB Item for Occupied Section on Track

```
1: function visualise14(stepNr) {
2:    setAttr("visb_debug_messages","text","Step "+stepNr+"/7, State ID: 14");
3:    setAttr("occupied_ttd_polygon","points","0.0,0 0.0,2.0 42.30769230769231,2.0
          42.30769230769231,0 100.0,0");
4:    ...
5:    highlightRow(stepNr);
6: }
```

Listing 8.3. JavaScript Function for Visualizing a Particular State in Figure 8.4

[2] A more complex one is available at `https://www3.hhu.de/stups/models/visb/train_4_POR_mch.html`.

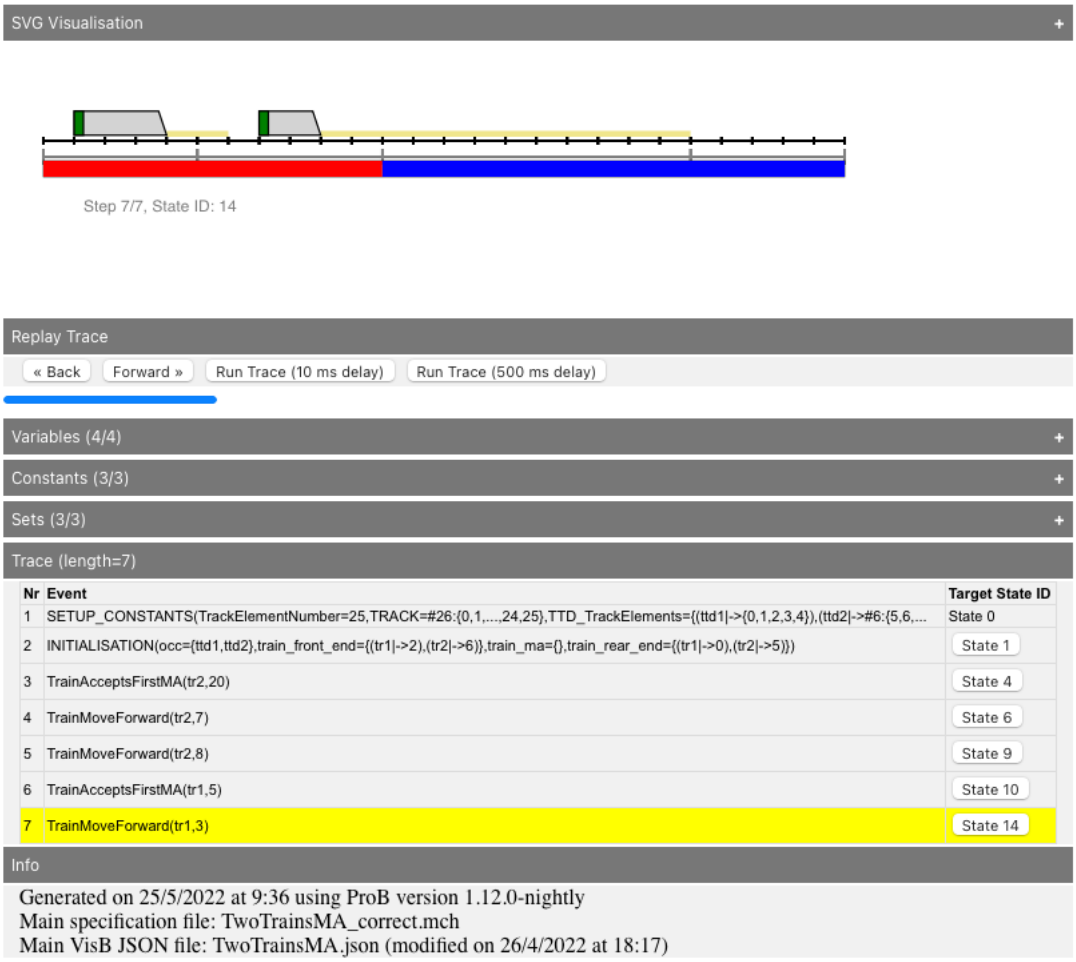


Figure 8.4.: Static VisB Export of Trace from Railway Domain; static export consists of the domain-specific VisB visualization, variables', constants', and sets' values, the trace consisting of events + parameters that were executed, and metadata; case study shows two trains that are driving on the same track; no block must be occupied by more than one train.

8.5. Dynamic HTML Export: Code Generation to HTML and JavaScript

Instead of generating a static HTML file consisting of a single trace, we now present a second approach which allows a domain expert to interact with the model. This approach is only supported for (a subset of) classical B. In this section, we explain the implementation of the dynamic export which was the main effort of this work. For this, we use the model of a vehicle's light system by Leuschel et al. [154] which was modeled using the specification by Houdek and Raschke [110]. This model encodes a subset of requirements from the specification which contains the key ignition, the pitman arm, and the vehicle's light system consisting of the blinking lights and the hazard warning lights. Later in Section 8.6, we demonstrate how modelers and domain experts can work with the dynamic export for the light system.

Within the dynamic export, state values are **computed** in JavaScript dynamically. This makes it possible for a domain expert to explore alternate paths, and not just the exported one. The dynamic export supports animation, domain-specific visualization in VisB, timed probabilistic simulation in SimB, and import/export of scenarios with descriptions. For the dynamic export, we also implemented model checking code generation (after [231]) with some features being used for animation and SimB. Those features contain functions for evaluating enabled transitions, and functions to compute the invariant. The complete model checking algorithm is not accessible to the user; but is later used to evaluate the performance of animation (see Section 8.7).

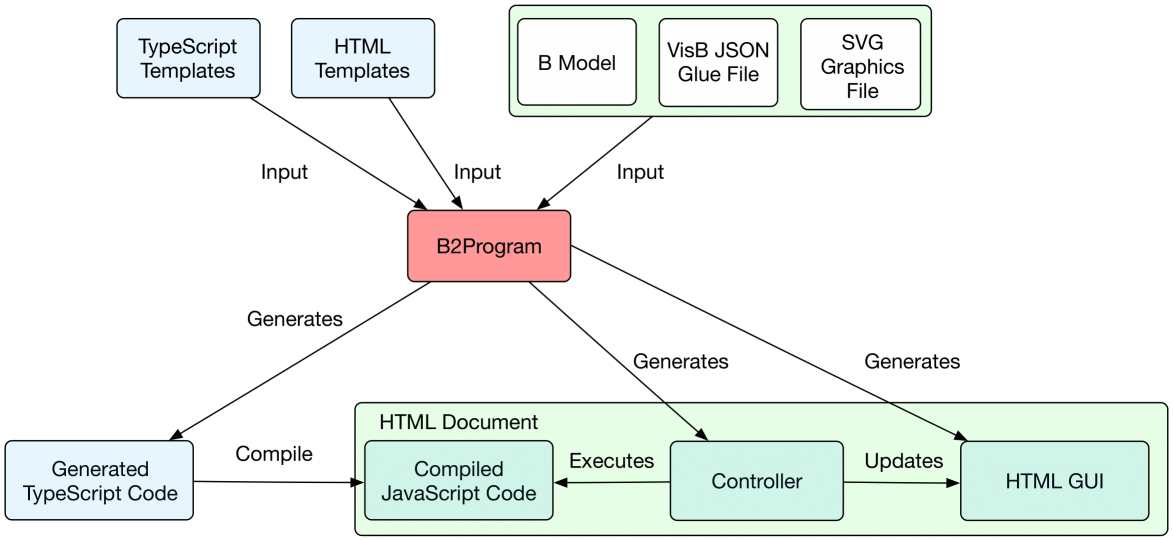


Figure 8.5.: Code Generation from B Model and VisB to HTML and JavaScript; templates are used as input to generate the TypeScript code for the B model, and the HTML GUI and its controller; the generated TypeScript code for the B model is compiled to JavaScript.

Figure 8.5 shows the infrastructure for code generation to HTML and JavaScript. In addition to the B model, B2Program also expects the VisB glue file and the associated SVG visualization as input. To support JavaScript, we extend B2Program by TypeScript following the approach described in our previous work [233]. Here, we decided not to generate JavaScript directly, but to generate TypeScript code as an intermediate step, which is then transpiled to JavaScript. We consider TypeScript as easier to debug than JavaScript, as there are fewer implicit type casts due to a stricter type system. Furthermore, many errors are already detected at compile time when transpiling from TypeScript to JavaScript (with more detailed error messages). Following the steps described in [233], we first implement TypeScript templates, and the B data types in TypeScript.

```
1: initialization(..., body, ...) ::= <<
2: ...
3: constructor() {
4:     <body>
5: }
6: >>
```

Listing 8.4. Parts of TypeScript Template for `INITIALISATION`

Listing 8.4 shows parts[3] of a TypeScript template which is used for code generation from the `INITIALISATION` clause. Generating code from the `INITIALISATION` clause shown in Listing 8.5 results in the TypeScript code shown in Listing 8.6.

```
1: INITIALISATION
2:     hazardWarningSwitchOn := switch_off ||
3:     pitmanArmUpDown := Neutral ||
4:     keyState := KeyInsertedOnPosition ||
5:     engineOn := FALSE
```

Listing 8.5. INITIALISATION clause of Sensors machine in Light System Model

```
1: constructor() {
2:     this.hazardWarningSwitchOn = new SWITCH_STATUS(enum_SWITCH_STATUS.switch_off);
3:     this.pitmanArmUpDown = new PITMAN_POSITION(enum_PITMAN_POSITION.Neutral);
4:     this.keyState = new KEY_STATE(enum_SWITCH_STATUS.KeyInsertedOnPosition);
5:     this.engineOn = new BBoolean(false);
6: }
```

Listing 8.6. Generated TypeScript Code from `INITIALISATION` clause of Sensors machine shown in Listing 8.5

By using the StringTemplate engine in B2Program, we could utilize the majority of B2Program's implementation for TypeScript/JavaScript without additional extensions. The main effort was to implement the B data types including the B operators in TypeScript which has to be done by a programmer manually. Those B data types include integers, booleans, strings, tuples, structs, sets, and relations together with their operators. It would also be possible to implement those data types in JavaScript directly; but due to the aforementioned reasons, we decided to implement them in TypeScript and then transpile to JavaScript.

[3]This part of the template is used for code generation without constants and copy constructor.

In addition to TypeScript templates, we also implemented HTML templates from which the graphical user interface (GUI) is generated. B2Program also generates a controller for the GUI and the translated B model. The controller's task is to execute operations in the translated model, and to update the GUI based on the model's current state.

8.5.1. Validation by Domain Expert

In the following, we describe how a domain expert can work with the dynamic export to support the validation of formal models. The steps are illustrated in Figure 8.6.

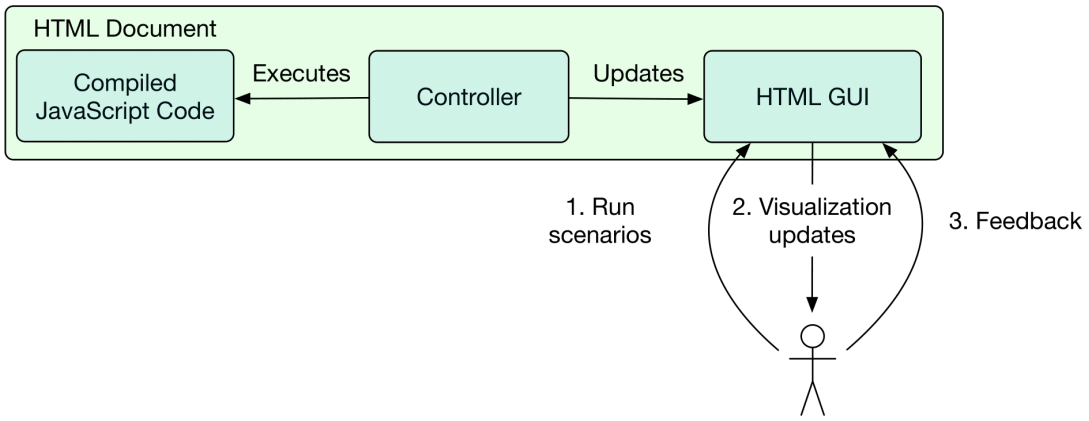


Figure 8.6.: Validation with HTML Document by Domain Expert; steps consisting of running scenarios, inspecting visualization updates, and giving feedback

In the first step, a domain expert can run various scenarios to check whether the model behaves as desired, i.e., whether the requirements are fulfilled. With the dynamic export, a domain expert can run scenarios via animation, trace replay, or SimB simulation. Animation allows a domain expert to explore a new scenario and store it as a trace. It is also possible for a domain expert to re-play an existing trace. The trace could either have been created by the domain expert itself, or it could have been supplied by the modeler together with the dynamic export. SimB simulation makes it possible to run a scenario with timing, probabilistic, and interactive aspects. To achieve this, a modeler must encode a SimB simulation as an *activation diagram* (currently in a JSON representation). The challenge for domain expert here is also that one has to familiarize oneself with the syntax and semantics of SimB's activation diagrams to model them.

During the execution of a scenario, the VisB visualization might update, i.e., its appearance might change. A domain expert can then check from their perspective whether the system described by the underlying formal model behaves as desired.

In the last step, a domain expert can give feedback to the modeler, e.g., by writing description text into the executed trace. The trace can finally be exported to a modeler who can load the trace in ProB2-UI.

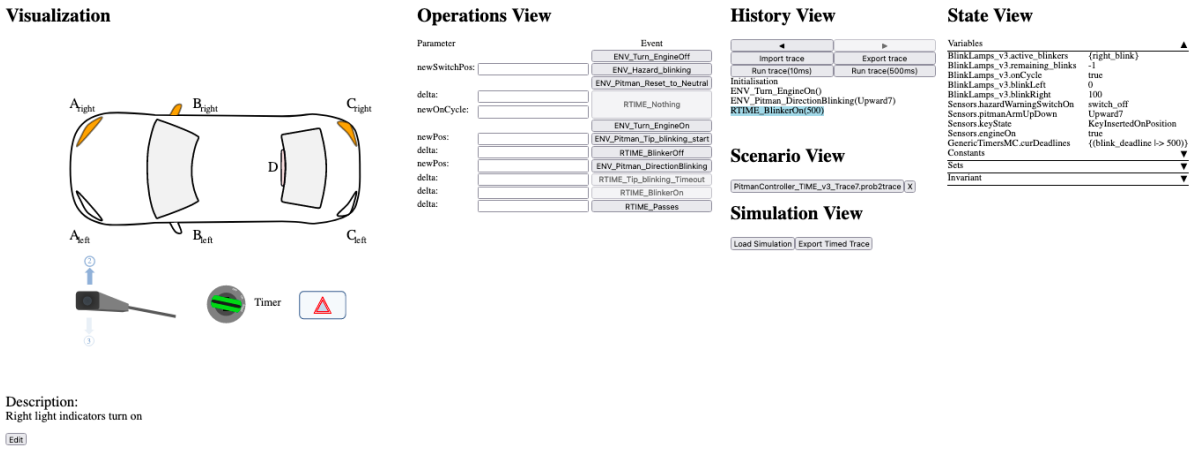


Figure 8.7.: Light System Web GUI with Domain-Specific VisB Visualization + Description Text, Operations View, History View, Scenario View, Simulation View, and State View

In our design, we decided to generate code from the VisB visualization, while generating interpreters for replaying traces and running SimB simulations. Thus, the VisB visualization cannot be changed in the HTML document. In contrast, it is possible to customize SimB simulations and run various traces. We have decided on this design because a domain expert usually only has one specific view, but wants to run multiple scenarios. Thus, it would be too inflexible if it is only possible to run one SimB simulation.

8.5.2. Graphical User Interface

In the following, we describe the GUI of the dynamic export which is generated from the Light System model (see Figure 8.7). [4] The GUI is inspired by ProB2-UI [25] and consists of its main views. Besides describing the generation of the GUI, we will also focus on the challenges. Those challenges particularly occur when using B expressions dynamically, e.g., in the operations view, or when loading traces or SimB simulations.

VisB View. On the left-hand side of Figure 8.7, one can see the domain-specific VisB visualization. Its features include (1) a graphical representation based on the model's current state, and (2) interaction with the model, i.e., executing an operation by clicking on a graphical object.

Listing 8.7 shows a VisB item defining an observer on the model's state. Particularly, this is a VisB item which defines the color of the right indicator at the vehicle's front. Assuming that the right blinks are active, the SVG object `A-right` should be filled in `#ffe6cc` (light orange) when the lamps are off, or `orange` when the lamps are on. When the right blinks are not active, `A-right` should be filled `white`.

[4] The example is also available at `https://favu100.github.io/b2program/visualizations/LightModel/PitmanController_TIME_MC_v4.html`.

```
{
  "id":"A-right",
  "attr":"fill",
  "value":"IF right_blink : active_blinkers THEN
      IF blinkRight=lamp_off
      THEN \"#ffe6cc\" ELSE \"orange\" END
    ELSE \"white\" END"
}
```

Listing 8.7. Example of VisB Item defining the color of the right indicator at the vehicle's front

```
_svg_vars["A-right"] = document.getElementById("LichtUebersicht_v4").contentDocument.
    getElementById("A-right")
_svg_vars["A-right"].setAttribute("fill",
  (_machine._BlinkLamps_v3._get_active_blinkers().elementOf(new DIRECTIONS(
      enum_DIRECTIONS.right_blink)).booleanValue() ?
    (_machine._BlinkLamps_v3._get_blinkRight().equal(_machine._BlinkLamps_v3.
        _get_lamp_off()).booleanValue() ?
    new BString("#ffe6cc") : new BString("orange")) :
  new BString("white")).getValue());
```

Listing 8.8. JavaScript Code Generation from Listing 8.7

As described in Section 8.4, values for the graphical objects' appearances are hard-coded in the static HTML export. To allow interactive animation, the visualization has to be updated based on the current state dynamically. For this purpose, the B expression is translated to JavaScript, and is thus evaluated at runtime (and not statically hard-coded as described in Section 8.4). The code generated for Listing 8.7 is shown in Listing 8.8.

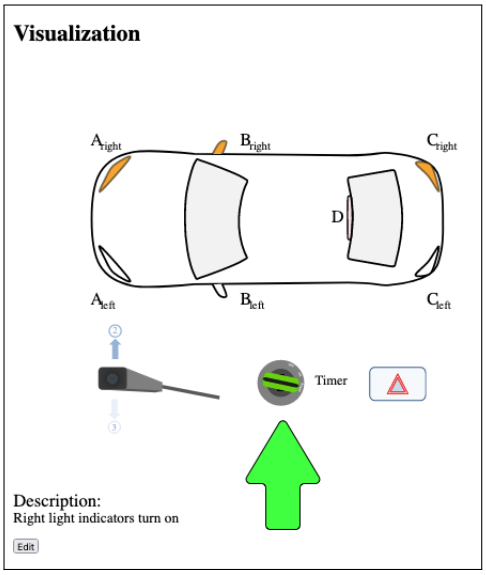
For a VisB event, B2Program generates a click listener on the SVG object which checks whether the corresponding B event is enabled, and executes it afterward. This makes it possible to interact with the model by clicking on the graphical element. `{"id":"engine-start", "event":"ENV_Turn_EngineOff"}` defines a click event on `engine-start`, triggering the `ENV_Turn_EngineOff` event. The generated code is shown in Listing 8.9.

```
_svg_events["ENV_Turn_EngineOff"] = document.getElementById("LichtUebersicht_v4").
    contentDocument.getElementById("engine-start");
_svg_events["ENV_Turn_EngineOff"].onclick = function() {
    transition = _machine._tr_ENV_Turn_EngineOff();
    ... // Check whether transition is feasible
    var parameters = [];
    var returnValue = _machine.ENV_Turn_EngineOff(...parameters);
    ... // Update views and internals
}
```

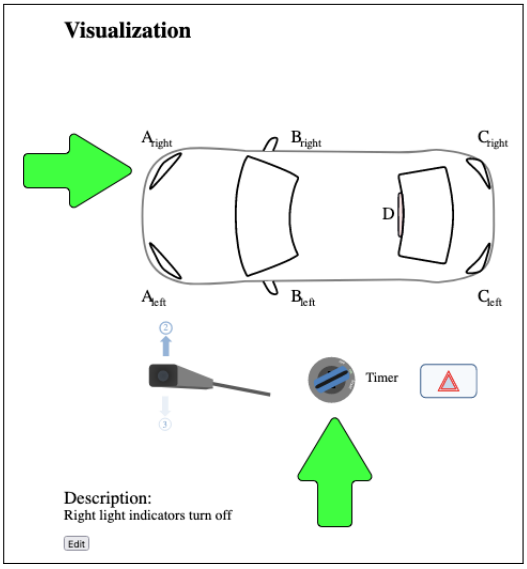
Listing 8.9. JavaScript Code Generation from `{"id":"engine-start", "event":"ENV_Turn_EngineOff"}`

The VisB view also includes a text area which allows a domain expert to provide feedback for the executed transition, and describe the current state. This description is saved when exporting the trace (see history view). It can be used by a modeler or another domain expert as valuable feedback. Description texts are also important as one might not see changes immediately when the current state changes.

Figure 8.8 shows the VisB visualization for the Light System model. In Figure 8.8a, one can see that the right light indicators are active, i.e., they are orange after executing the generated code shown in Listing 8.8. When the user presses the engine button (`engine-start`; also directed by the green arrow) in the state shown in Figure 8.8a, the engine and the right light indicators turn off (see Figure 8.8b).



(a) Light System with Right Light Indicators turned on



(b) Light System with Right Light Indicators turned off after turning off engine

Figure 8.8.: VisB Visualization (+ Description Texts) of Light System with Pitman Arm, Key Ignition, Warning Lights Button

Operations View. The *operations view* allows the execution of operations as an alternative to the VisB view.

Within the operations view, B2Program generates functionalities of an animator, i.e., a user receives suggestions of which operations (with which parameters) are enabled. In particular, B2Program generates a button for each operation in the formal model, and a text field for each corresponding parameter. Each button is enabled exactly when the operation is enabled; otherwise, the button is enabled. Furthermore, the text fields store a list of options for parameters the operation is feasible for execution. To compute the operations' enabledness and the feasible parameters, B2Program uses functions to compute enabled transitions which are originally generated for model checking [231]. The computation is done when reaching another state in the model, and works as follows:

1. Invoke the function to compute outgoing transitions for the operation.

2. There are outgoing transitions, i.e., the function returns `true` or a non-empty set of tuples. Then check the inner guards by trying to execute the computed transitions.

Enable the button for the operation, and use the set of tuples to fill the list of possible options for parameters.

3. Otherwise, there is no outgoing transition, i.e., the function returns `false` or an empty set. Then disable the button for the operation. Clear the list of possible options for parameters.

Code generation for executing an operation via the *operations view* is done similarly to the VisB events (see Listing 8.10). To achieve better user-friendliness, the user does not have to explicitly specify the parameters here; by default, the first possible combination of parameters is used for execution.

```
1: _machine_events["ENV_Turn_EngineOff"] = document.createElement("button");
2: _machine_events["ENV_Turn_EngineOff"].onclick = function() {
3:     transition = _machine._tr_ENV_Turn_EngineOff();
4:     ... // Check whether transition is feasible
5:     var parameters = [];
6:     var returnValue = _machine.ENV_Turn_EngineOff(...parameters);
7:     ... // Update views and internals
8: }
```

Listing 8.10. JavaScript Code Generation for Button to execute `ENV_Turn_EngineOff` Event

Since values for parameters are entered dynamically in the operations view, we encountered a problem that these values have to be evaluated. Compared to animators like ProB, we only allow constant values, e.g., 1, TRUE, or red (whereas red is a set element). These constant values still have to be parsed in a lightweight manner, in order to transform them from a string representation to a feasible representation in B2Program. For example, `1` is transformed to `new BInteger(1)`. In the case that we would try to allow expressions in general, e.g., 1+1+a, it would be necessary to embed a complete B parser and evaluator for expressions. This would contradict the idea of code generation; thus we decided to implement it in a lightweight manner only.

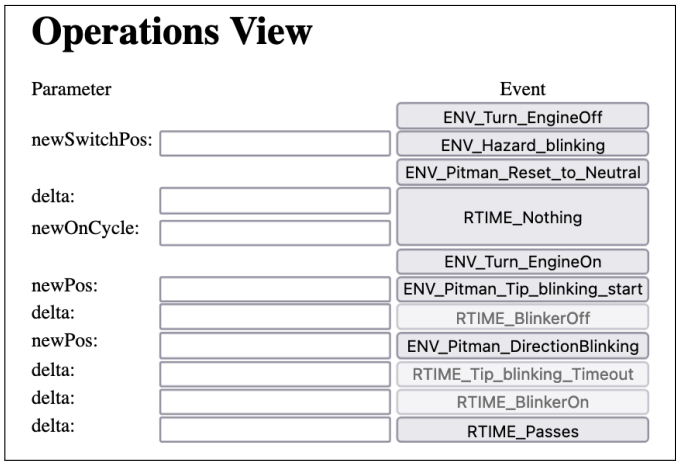


Figure 8.9.: Example: Operations View for Light System

An example of the operations view in the dynamic export is shown in Figure 8.9. This operations view shows all buttons and text fields for the Light System model, including the button for `ENV_Turn_EngineOff` which was discussed before.

State View. Within the *state view*, one can view the model's current state in mathematical notation. Although mathematical notations are difficult for a domain expert to understand, it can still be important to debug the model. We display the *set*, *variable*, *constant*, and *invariant* sections textually. For better readability, B2Program splits the invariant into its conjuncts. This feature was also implemented for model checking to achieve better performance [231].

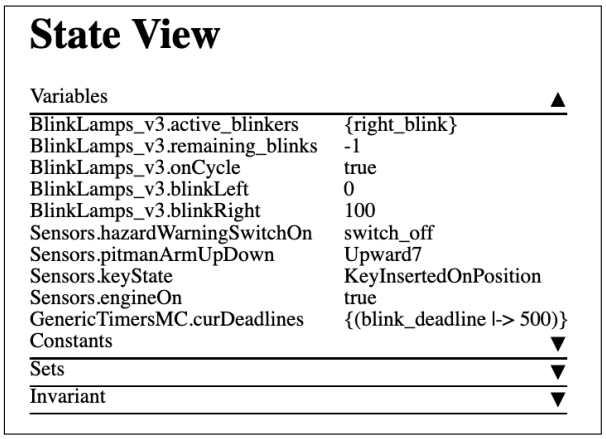


Figure 8.10.: Example: State View for Light System

Figure 8.10 shows an example of the state view for the Light System model. This state corresponds to the one shown in Figure 8.8a.

History View. The *history view* shows the currently animated trace. When executing an operation (via the operations view or by clicking inside the VisB visualisation), the corresponding transition with input/output parameters is displayed in the history view. At the same time, this transition is saved in a list together with the model's state. By default, an empty description text is created which can be modified by a domain expert in the VisB view. Those data are used to generate a ProB2-UI trace.

Within the history view, there are buttons to import, and export an animated trace represented in ProB2-UI's format. In Section 8.6, we demonstrate how this, together with the *scenario view*, improves communication between modelers and domain experts.

Figure 8.11 shows an example of the history view for the Light System model which contains an animated trace. This trace could have been animated via the VisB view or the operations view by hand, or loaded via the `Import trace` button.

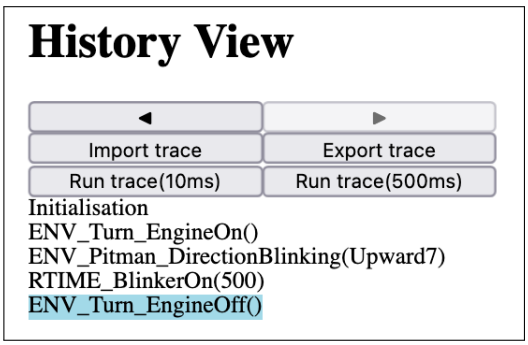


Figure 8.11.: Example: History View for Light System

Scenario View. Within the *scenario view*, a domain expert can store a set of traces. Each trace is stored in a ProB2-UI trace file. Replaying a trace is done by iterating over its transitions; the ProB2-UI trace files contain for each transition the operation name and parameter values. Using the other views described above, a domain expert can then step through the scenario, check whether the system behaves as desired, and add a description text.

Figure 8.12 shows an example of the scenario view for the Light System model. The scenario view shows a set of traces that have been loaded via the `Import trace` button in the history view. By clicking on such a trace, it is loaded into the history view and set as the currently animated trace.

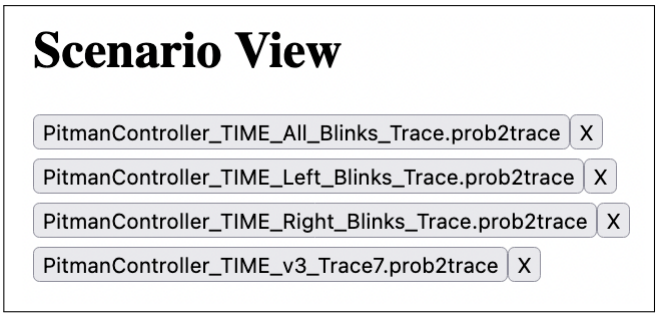


Figure 8.12.: Example: Scenario View for Light System with a set of traces

Simulation View. The simulation view enables a user to perform real-time simulations using SimB files that can be loaded in the *simulation view*. As mentioned in Section 8.2, SimB is based on activation diagrams describing how events activate each other with timing and probabilistic behavior.

Our generated Javascript code re-implements the SimB algorithm as described by Vu et al. [237]. There, activations of events are managed in a scheduling table which stores the times until the next activation. In particular, our implementation performs the following steps [237]:

1. Let time pass until reaching the next scheduled activations
2. Update the activation times in the scheduling table
3. Iterate over the activations in order of their priority
   - If the activation's time has expired then
     a) Execute the operation if it is enabled
     b) Activate SimB activations that are triggered by this activation
     c) Remove the activation from the scheduling table
   - Otherwise, ignore the activation
4. Compute the (minimal) time until the next activations shall be executed
5. Specifically for the GUI: Update all views

Both steps Item 3a and Item 3b can also take into account probabilistic behavior. SimB's *interactive simulation* [236] is also supported in the dynamic export, i.e., allowing a user to trigger additional events.

To support SimB in the dynamic export, we considered two options:

1. Either, to generate code for a specific SimB simulation, similar to the B model and the VisB visualization.
2. Or to provide a SimB interpreter which allows loading several SimB simulations.

Here, we decided to implement the second possibility, to allow a domain expert to load several simulations, and not only specific ones. However, to support the full power of SimB activation diagrams, we would have to provide an evaluator for general B expressions. For now, we thus only allow constant values for those B expressions. In the future, we could reconsider pursuing the first option, i.e., to generate code for SimB activation diagrams. But as mentioned above, this means that the user can only choose from specific SimB simulations that were generated together with the B model.

Figure 8.13 shows an example of the simulation view of the Light System. There, a modeler has loaded two simulations, one which simulates the driver's and the vehicle's behaviors automatically, and a second one which simulates the vehicle's behaviors as a reaction to the driver's input (which has to be operated manually). In the dynamic export, it is also possible to export a simulated trace with timing behavior. This is called a *timed trace*, and corresponds to SimB's representation of an activation diagram where within the trace each event triggers the next one with time elapsing in between. Note that timed traces only contain constant values in their activation diagram; thus, a domain expert can re-play all timed traces with the SimB interpreter.

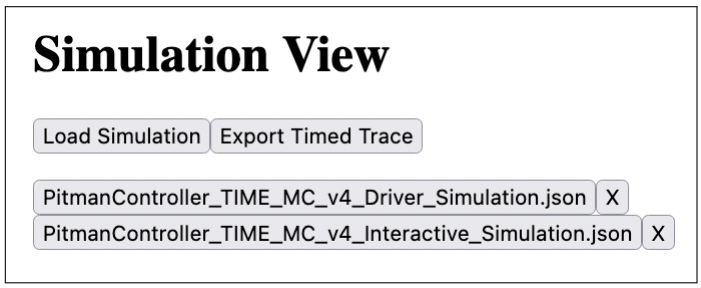


Figure 8.13.: Example: Simulation View for Light System

8.6. Case Studies

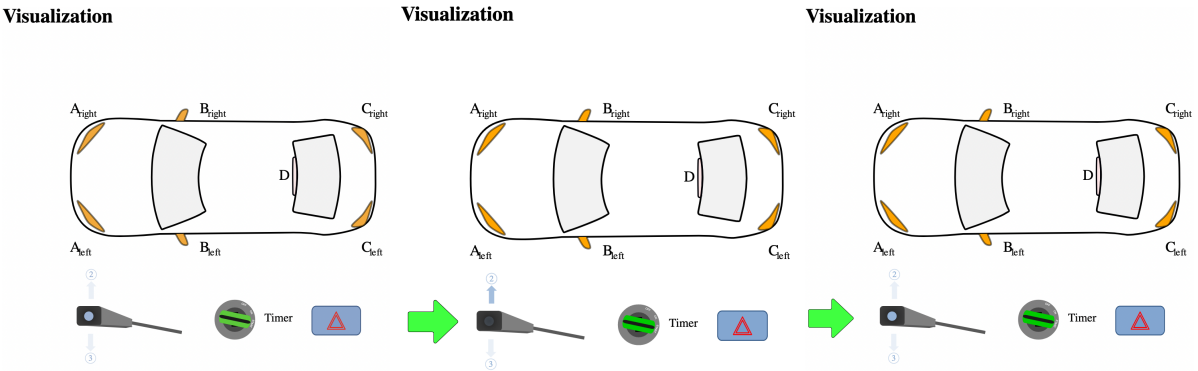
This section demonstrates how this work (1) makes it possible for a domain expert to validate requirements, and (2) improves communication between the modelers and the domain experts. We will study two case studies: a vehicle's light system from the automotive domain [154], and a landing gear from the aviation domain [135]. On the one hand, this section shows that our approach applies to different domains. On the other hand, the first case study focuses on the communication between domain experts and modelers, while the second case study focuses on the communication between domain experts with different perspectives.

Vehicle Light System. For the light system case study, domain experts provide a set of validation sequences (aka scenarios). The dynamic export allows a domain expert to run scenarios directly, and then communicate with modelers afterward. In the following, we focus on *sequence 7* which is given in the specification [110]. *Sequence 7* validates the turn indicator's and the hazard light's behaviors. In particular, events for tip blinking, direction blinking, and the hazard warning lights are executed, and the desired behavior is checked afterward.

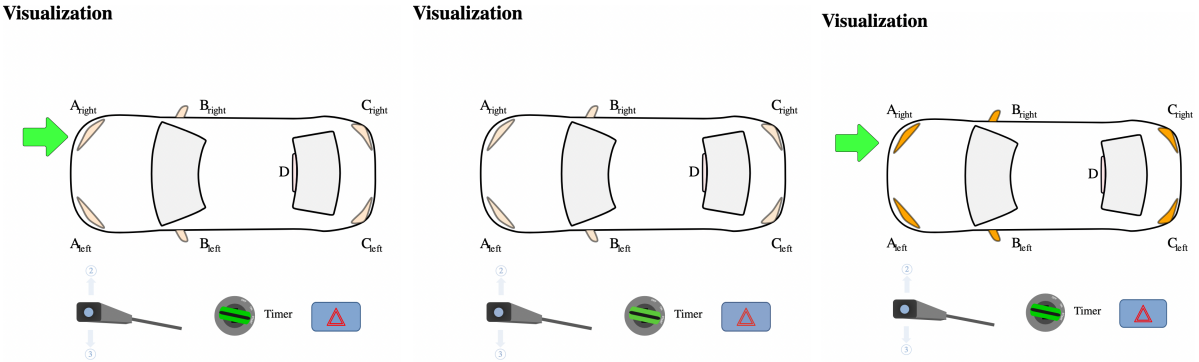
Figure 8.14 shows parts of *sequence 7* as domain-specific visualizations in the dynamic export. First, a modeler animates a trace in ProB2-UI to validate sequence 7 (see Figure 8.15). The sequence's feasibility in the model has already been shown by Leuschel et al. in [154]. Based on this sequence, we outline how our approach helps to improve communication between modelers and domain experts.

To ensure that the modeler has not misunderstood the requirements, they can then export the trace to a domain expert, who could load this trace into the generated HTML document (see Figure 8.16). The domain expert can then inspect whether the correct behavior was indeed implemented by the modeler.

A critical point in the sequence is to validate that "if the warning light is activated, any tip-blinking will be ignored or stopped if it was started before" (requirement **ELS-13** in [110]). This part of the animation is shown in steps (a) – (f) in Figure 8.16 which corresponds to Figure 8.14. With the help of the domain-specific visualization (see Figure 8.14), the domain expert can easily approve that the desired behavior has indeed been implemented. For example, in step (f) of Figure 8.16, a domain expert can check



(a) *System*: Time passes (b) *User*: Activates tip blinking to `Upward5` (c) *User*: Resets pitman arm to `Neutral`



(d) *System*: All light indicators turn off (e) *System*: Timeout for tip blinking with `Upward5`; no changes in visualization (f) *System*: All lights indicators turn on

Figure 8.14.: Domain-Specific Visualization of States after Executing (a) – (f) in Figure 8.16; green arrows show changes compared to previous state.

the aforementioned behavior, and add or modify a description text for a modeler (see Figure 8.17).

Furthermore, the dynamic export allows a domain expert to inspect alternate paths for the same requirement, thereby establishing a stronger guarantee of whether a requirement is fulfilled. This process is supported by the new SimB implementation in this work. In particular, a user/domain expert can simulate user inputs and the vehicle's system reactions with timing and probabilistic behavior automatically. For more precise control over the user inputs, one can also use *interactive simulation*. Here, user input is applied manually, while the system reaction afterward is simulated automatically. For instance, a user/domain expert could execute user interactions in Figure 8.14 (marked with *User*), whereafter the system's reaction (marked with *System*) could be observed with delay.

In our previous work [236], we already validated other requirements about the light system (in particular **ELS-1**, **ELS-8**, and **ELS-12**) by executing the user interaction, and observing the system's reaction afterward. Those validations could now also be done by a user or domain expert via the dynamic export of this work; and not only via ProB2-UI.

Position ▲	Transition
103	RTIME_BlinkerOff(delta=500)
104	RTIME_BlinkerOn(delta=500)
105	RTIME_BlinkerOff(delta=500)
106	RTIME_BlinkerOn(delta=500)
107	RTIME_Passes(delta=100)
108	RTIME_Passes(delta=100)
109	ENV_Pitman_Tip_blinking_start(newPos=Upward5)
110	RTIME_Passes(delta=100)
111	ENV_Pitman_Reset_to_Neutral
112	RTIME_BlinkerOff(delta=200)
113	RTIME_Tip_blinking_Timeout(delta=200)
114	RTIME_BlinkerOn(delta=300)
115	RTIME_Passes(delta=100)
116	RTIME_Passes(delta=100)
117	ENV_Hazard_blinking(newSwitchPos=switch_off)
118	RTIME_Nothing(delta=300, newOnCycle=FALSE)
119	RTIME_Nothing(delta=100, newOnCycle=FALSE)
120	RTIME_Nothing(delta=100, newOnCycle=FALSE)
121	RTIME_Nothing(delta=100, newOnCycle=FALSE)
122	**RTIME_Nothing(delta=100, newOnCycle=FALSE)**

Figure 8.15.: Parts of Sequence 7 in the History View of ProB2-UI

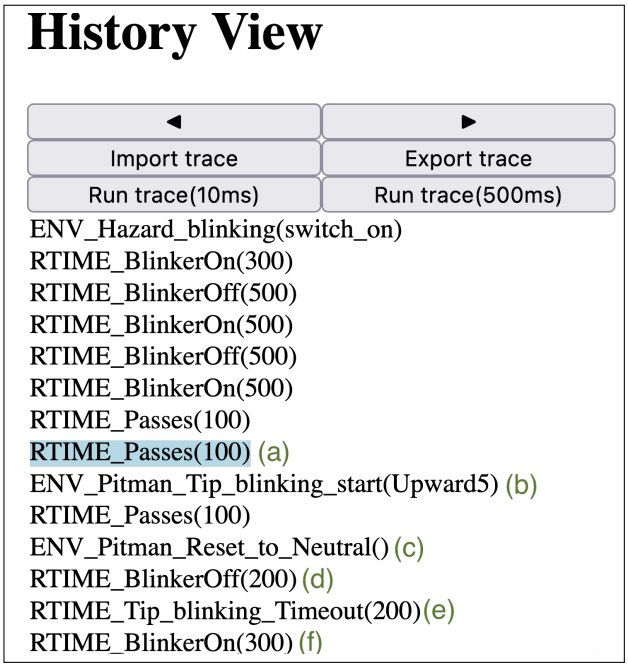


Figure 8.16.: Parts of Sequence 7 in the History View of the Interactive Validation Document; (a) – (f) added manually; (a) – (f) corresponds to steps 108 – 114 in Figure 8.15.

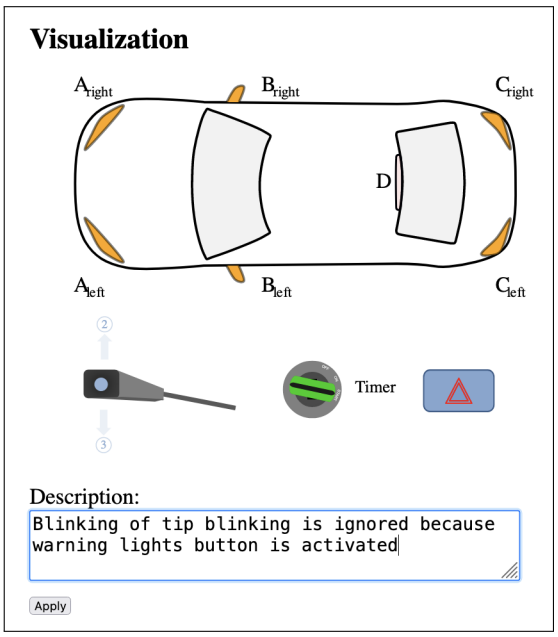


Figure 8.17.: Modifying Description for Step (f) in Figure 8.16

Regarding the SimB features in the dynamic export, one can also export (timed) traces to a modeler again. As mentioned in Section 8.5.2, a timed trace is a special case of a SimB simulation. Consequently, a domain expert can export a timed trace which can then be re-played by a modeler in real-time.

Landing Gear. The landing gear model [135] by Ladenberger et al. is modeled based on the specification by Boniol [39]. For the demonstration, we use the refinement level which includes gears, doors, handle, switch, and electro-valves. To be able to use B2Program, we have manually translated the Event-B model to classical B. Figure 8.18 shows parts of the generated GUI from the landing gear model which contains the VisB view and the history view. The domain-specific VisB view shows a hydraulic circuit consisting of the handle, the switch, the electro-valves, and the cylinders.

Using the operations view (which we omitted here due to space reasons), a domain expert can animate traces representing desired requirements. In this example, the domain expert has animated the *retraction sequence* from the specification. This trace can then be exported for ProB2-UI, to be used by a modeler. It can also be converted for use by another domain expert more focused on other aspects of the model. For instance, Figure 8.19 shows an alternate domain-specific visualization with gears and doors, and description text provided by the first domain expert. The second domain expert can import the trace created from Figure 8.18.

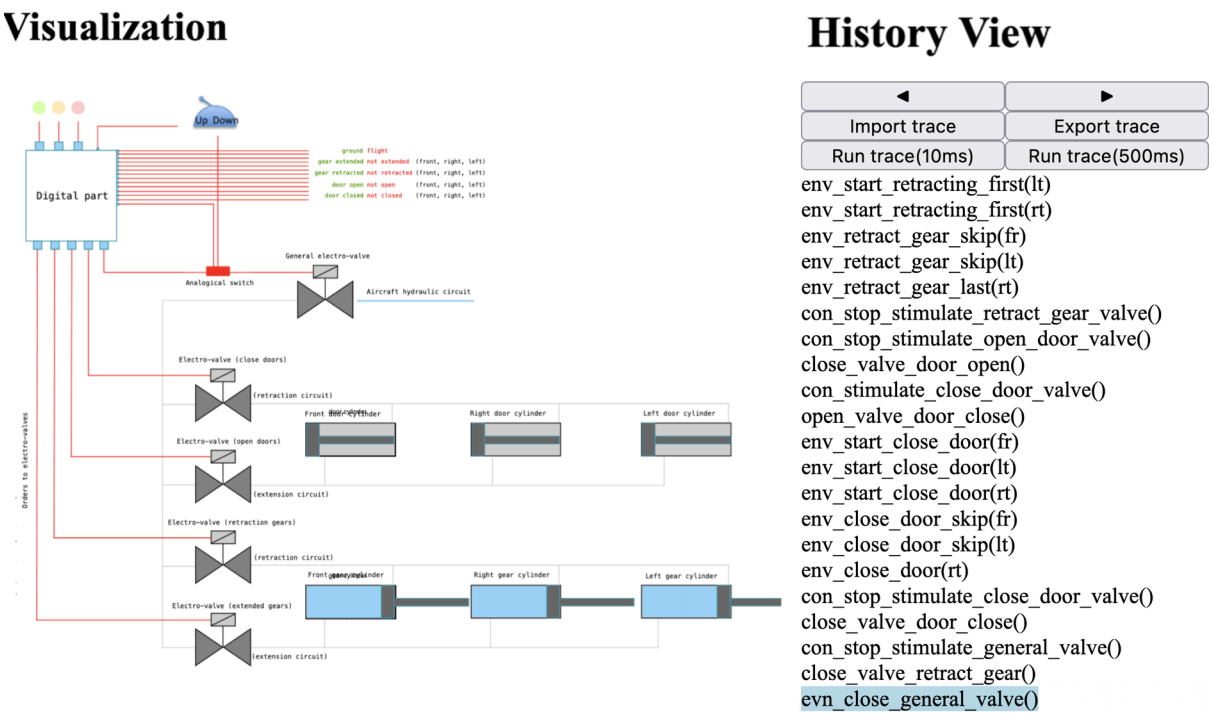


Figure 8.18.: Retraction Sequence (also shown in History View) with Hydraulic Circuit as Domain-Specific Visualization; Hydraulic Circuit contains the handle, the switch, the electro-valves, and the cylinders.

Comparing Figure 8.18 and Figure 8.19, one can see that a pressurized door cylinder is equivalent to a closed door, and an unpressurized gear cylinder is equivalent to a retracted gear. [5] Thus, our approach does not only improve communication between modelers and domain experts, but also between domain experts from different perspectives.

To achieve more realistic user interaction, a modeler can provide interactive timed SimB simulations to both domain experts. Both domain experts can then push up or push down the pilot's *handle* manually, and check whether the respective *retraction sequence* or *outgoing sequence* is executed within 15 seconds automatically (R_{11} and R_{12} from the specification [135]). In comparison to animation, users simply need to perform user actions, allowing them to experience and validate timing properties.

[5] A partially pressurized door cylinder is equivalent to a moving door. An unpressurized door cylinder is equivalent to an opened door. A pressurized gear cylinder is equivalent to an extended gear. A partially pressurized gear cylinder is equivalent to a moving gear.

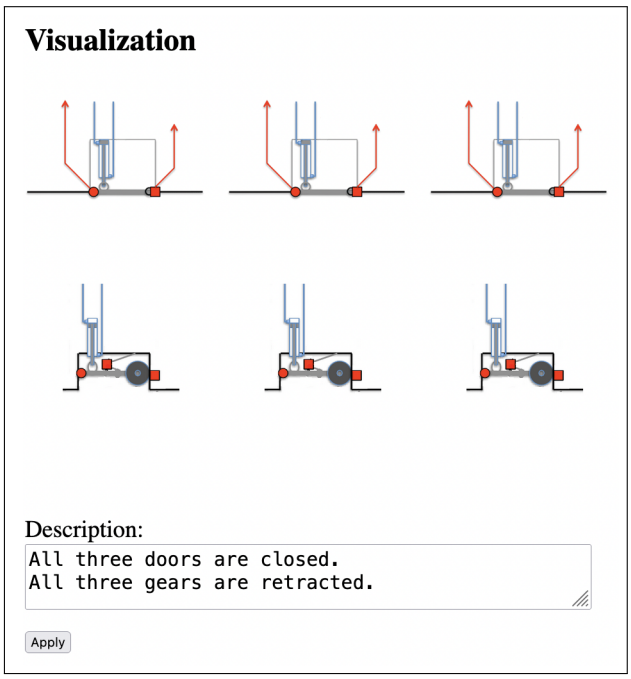


Figure 8.19.: Retraction Sequence with Gears and Doors as Domain-Specific Visualization, and Description Text

8.7. Applicability of JavaScript Code Generation

Another important aspect is the applicability of JavaScript code generation. In this section, we focus on the limitations and the performance.

Limitations. As the JavaScript code generator is based on B2Program, it shares the same restrictions that are discussed in [233, 231].

B2Program has strong restrictions for quantified constructs. Indeed, for bounded variables $a_1 \dots a_n$ constrained by a predicate P, the first n conjuncts of P must constrain the bounded variables in the exact order they are defined [233]. As discussed in [231], we plan to loosen this restriction in future, e.g., by allowing pruning predicates to reduce the enumeration size. Currently, B2Program iterates over all possible values before other predicates are formulated. The conjuncts in P can take the following form and are treated as follows [233]:

- $a = E$ is translated by assigning the value of E to the bounded variable a.
- $a \in S$ is translated to a for loop where a is constrained while iterating over the set S.
- $a \subset S$, $a \subseteq S$ are translated to a for loop where a is constrained while iterating over the (strict) superset of S.

B2Program also forbids set operations on infinite sets, or storing them in variables [233]. We do not plan to support all operations on infinite sets as some might require embedding a constraint solver, against which we decided to do [233].

B2Program chooses just one execution path for non-deterministic constructs such as `ANY`, `CHOICE`, or non-deterministic assignments [233]. Thus, models with those constructs can not be animated exhaustively. For precise animation and model checking of `ANY`, `CHOICE`, and non-deterministic assignments, it will be necessary to compute all choice points. Regarding animation, the user requires more control over the desired choice point for execution, while for model checking, it will be necessary to cover all choice points.

An `ANY` substitution is of the form:

ANY $v_1, \ldots, v_n$ WHERE *predicate* THEN *substitutions* END

This means that for (local) variables $v_1, \ldots, v_n$ where *predicate* is true, execute *substitutions*. Otherwise, the entire operation in which the `ANY` substitution is used is not executable. As the *predicate* might constraint multiple values for $v_1, \ldots, v_n$, this substitution is non-deterministic.

A `CHOICE` substitution is of the form:

CHOICE *substitutions1* OR *substitutions2* END

This means that either *substitutions1* or *substitutions2* is executed. As there are two possibilities to choose from, this substitution is non-deterministic.

B2Program only allows top-level `PRE` and `SELECT` as non-determinism [231] for model checking. Inner guards, e.g., inner `SELECT`s cause problems when calculating enabled transitions (discussed in [231]). Regarding animation in this work, a superset of possible transitions is computed from the top-level guards first; inner guards are checked during execution of the transition. To support inner `PRE` and `SELECT` for model checking, it would be necessary to adapt the algorithm slightly so that precondition violations are detected, and a state is discarded when an inner guard is not true. This is already the case for animation.

A `SELECT` substitution is of the form:

SELECT *guard* THEN *substitutions* END

This means that, when the *guard* is true, then execute *substitutions*. Otherwise, the entire operation in which the `SELECT` substitution is used is not executable. Furthermore, this substitution is often used at the top level to constrain possible values for the parameters.

A `PRE` (precondition) substitution is of the form:

PRE *predicate* THEN *substitutions* END

This means that, when the *predicate* is true, then execute *substitutions*. Otherwise, the entire operation in which the `PRE` substitution is used leads to a *precondition violation*. Top-level `PRE` substitutions behave similarly to `SELECT`. `PRE` substitutions are also often used at the top level to constrain possible values for the parameters.

In conclusion, some models must be rewritten according to these rules; still, there are also models where it is not possible. Note that B2Program supports a significantly larger subset than B0 code generators. So, B2Program can be used at an early development stage; but especially at a very early stage, some models are too high-level for B2Program. One must then refine the model further to enable B2Program for validation, or use the static export from Section 8.4.

Performance. In the previous work [233], we already compared Java and C++ code generation with ProB. To achieve good performance, we implemented the B data types `BSet` and `BRelation` using persistent data structures (similar to Java and C++, see [233]). For this, we used the Immutable[6] Javascript library, which also makes use of structural sharing [17]. Furthermore, we used primitive integers in the generated Java, JavaScript, and C++ here. As already explained in Section 8.5, code generation is similar to the one for the previously supported languages Java and C++: we adapted the existing templates for TypeScript, but still use the same implementation for generating code. Afterward, the generated TypeScript code for the B model is transpiled to JavaScript. Below we investigate how much the change of target language and libraries affects the performance of the generated code.

We have benchmarked the models from [233] and [231] for ProB[7], Java[8], and C++[9] again and compared them with JavaScript [10]. As explained in [233] and [231], those selected models range from small to large ones, covering various performance aspects. Due to the small number of states, we replaced the simulation benchmarks Lift, Traffic Light, and Sieve with the following machines for model checking: a Counter to one million, Landing Gear, NoTa, and N-Queens (with N = 4). As explained in [231], some models were rewritten to make B2Program applicable; for ProB, we benchmarked the original versions. Landing Gear was originally modeled by Ladenberger et al. [135], and then translated to classical B to make B2Program applicable (see Section 8.6). N-Queens also has few states, but computing transitions without constraint solving is very time-consuming. Compared to the earlier performance analyses, further optimizations were made in the generated Java, JavaScript, and C++ code. The complete benchmark set can be found in the B2Program repository[11]. Each benchmark is run five times on a MacBook Pro (16 GB RAM, Apple M1 Pro Chip with eight cores[12]) with a timeout of one hour, and afterward, the median runtime is taken.

Table 8.1 shows the simulation benchmarks comparing ProB, Java, JavaScript, and C++. Following the approach in [233], we execute operations in a long-running `while` loop. For ProB, we used the `-execute` command to just execute the first enabled

[6] `https://immutable-js.com/`
[7] ProB CLI 1.12.2-nightly built with SICStus 4.8.0 (arm64-darwin-20.1.0)
[8] OpenJDK 64-Bit Server VM (build 18.0.2+0, mixed mode, sharing)
[9] Compiled with clang, version 13.0.0 (clang-1300.0.29.30); `-O1` for model checking benchmarks (`-O2` did not optimized further for model checking [231]), `-O2` for simulation benchmarks.
[10] NodeJS 19.9.0
[11] `https://github.com/favu100/b2program`
[12] Six performance cores, two efficiency cores.

		ProB	Java	JavaScript	C++
Lift					
(2×10^9 op calls)	Runtime	> 3600	5.85	13.86	0.07
	Speed-up	1	> 615.38	> 259.74	> 51 428.57
Traffic		ProB	Java	JavaScript	C++
Light	Runtime	> 3600	3.06	17.81	0.08
(1.8×10^9 op calls)	Speed-up	1	> 1176.47	> 202.13	> 45 000
Sieve		ProB	Java	JavaScript	C++
(1 op call,	Runtime	49.83	2.86	21.94	4.65
primes until 2 Million)	Speed-up	1	17.42	2.27	10.72
Scheduler		ProB	Java	JavaScript	C++
(9.6×10^6 op calls)	Runtime	158.27	2.17	2.78	1.98
	Speed-up	1	72.94	56.93	79.93
Cruise		ProB	Java	JavaScript	C++
Controller (Volvo,	Runtime	> 3600	6.68	10.66	0.21
136.1×10^6 op calls)	Speed-up	1	> 538.92	> 337.71	> 17 142.86
CAN Bus		ProB	Java	JavaScript	C++
(J. Colley,	Runtime	199.66	1.61	1.65	0.61
15×10^6 op calls)	Speed-up	1	124.01	121.01	327.31
Train (ten routes) [151, 5]		ProB	Java	JavaScript	C++
(940×10^3 op calls)	Runtime	45.16	2.41	3.54	1.66
	Speed-up	1	18.74	12.76	27.20
sort_m2_		ProB	Java	JavaScript	C++
data1000 [200]	Runtime	7.67	0.44	0.13	0.10
(500.5×10^3 op calls)	Speed-up	1	17.43	59	76.7

Table 8.1.: Simulation Runtimes (ProB, Generated Java, Generated JavaScript, and Generated C++ Code) in Seconds with Number of Operation Calls (op calls), Speed-Up Relative to ProB; Models from [233] were re-benchmarked for ProB, Java, and C++ with another device.

transitions and avoid exploring the state space. Nevertheless, ProB always performs variant checking for `while` loops which cannot be turned off.

Here, one can see that the generated JavaScript code outperforms ProB. For most benchmarks, JavaScript is one or two orders of magnitude faster than ProB. Sieve is a model with many set operations where JavaScript is less than one magnitude faster than ProB [13]. The slower runtime of ProB could be explained by the use of an interpreter implemented in Prolog. In contrast, the B syntax is compiled to JavaScript with B2Program. In addition, for the ARM processor used in the experiments, SICStus Prolog lacks the JIT compiler [14]. In our previous work [233], the JIT compiler was available and the performance gap between ProB and the generated Java code is less pronounced. Finally, ProB supports unlimited precision integers, while we used primitive integers for Java, JavaScript, and C++ here. Although JavaScript is an interpreted language, our new backend for B2Program performs very well: the JavaScript and Java runtimes are usually within an order of magnitude and it seems that the JIT compiler

[13] This Sieve model is a slightly different, more low-level version compared of the one in Listing 8.1.
[14] `https://sicstus.sics.se/download4.html`

in NodeJS optimizes effectively. As already discussed in prior work, C++ leads to a speedup compared to Java regarding simulation for all benchmarks except Sieve [233]. Compared to JavaScript, C++ is even faster for all simulation benchmarks we have considered (see Table 8.1). Especially in the simulation case where we specialize the operations' input specifically, *clang*'s `-O2` optimization can optimize strongly [233].

Note, however, that Table 8.1 contains benchmarks for simulation or trace-replay, *not* for animation, i.e., we measure the performance of executing the model on long-running paths where operations parameters are provided explicitly. In animation — as in model checking — the tools need to compute *all* enabled transitions and present them to the user. To analyze the performance for this, we analyze the model checking performance of the generated JavaScript code. In prior work, model checking code generation was implemented in B2Program for Java and C++ [231]. In this work, we have extended B2Program's model checking code generation for JavaScript. For Java, JavaScript, and C++, we benchmarked both with and without caching. When activating B2Program's caching, the operations' guards and the invariant conjuncts are only computed if they contain variables that are changed by the operation that is executed to reach this state [231]. Although caching is not yet implemented in the animator of the dynamic export, this feature could be implemented later easily. For ProB we activated the operation reuse feature together with state compression [150] (`-p OPERATION_REUSE full` and `-p COMPRESSION TRUE`) which is an efficient caching strategy for animation and model checking. Invariant checking is also activated as it is displayed to the user of the dynamic export.

Note that we do not benchmark code generation and compilation time of B2Program. In the use case of this work, *interactive validation documents* are usually generated once, and can then be used by a domain expert. However, for the verification use case with model checking, B models might be compiled more frequently, e.g., when the encoding of the B model changes. Compilation for C++ might be significantly more time-consuming than model checking [231]. More details regarding code generation and compilation runtime with B2Program are discussed in [231].

The model checking results are shown in Table 8.2. Here, one can see that for most JavaScript benchmarks, we achieved runtimes within an order of magnitude as Java and C++. An exception here is Sort without caching, where JavaScript is around one order of magnitude or more slower than C++, and at least one order of magnitude slower than Java. For a few models JavaScript is faster than Java or C++, while for others it is the other way around (see Table 8.2). As already analyzed in [231], ProB's operation reuse can improve the performance up to the same order of magnitude as Java and C++, and thus also JavaScript for some models. For example, the generated JavaScript code has better performance than ProB for CAN BUS, Landing Gear, or NoTa, but not for Train, Sort (without caching), or N-Queens. The poor performance of JavaScript for Sort without caching is due to the invariant checking. The N-Queens example shows that ProB's constraint-solving capability can make it much faster than B2Program (also discussed in [231]). A similar effect appeared in the automotive case study in Section 8.6, where ProB can be up to three orders of magnitude faster at computing all enabled transitions presented to the user. For Light System, we only measured the computation

Counter		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(1 000 001 states,	Runtime	65.23	0.67	0.80	1.21	1.78	0.36	0.63
2 000 001 transitions)	Speed-up	1	97.36	81.54	53.91	36.65	181.19	103.54
Cruise Controller (Volvo,		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
1360 states,	Runtime	0.37	0.46	0.46	0.13	0.16	0.06	0.07
26 149 transitions)	Speed-up	1	0.80	0.80	2.85	2.31	6.17	5.29
CAN BUS (J.Colley,		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
132 599 states	Runtime	14.98	1.31	1.49	2.34	3.58	1.14	1.02
340 266 transitions)	Speed-up	1	11.44	10.05	6.4	4.18	13.14	14.69
Landing Gear [135]		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(131 328 states,	Runtime	24.58	4.23	4.86	8.87	11.26	6.07	5.82
884 369 transitions)	Speed-up	1	5.81	5.06	2.77	2.18	4.05	4.22
NoTa [183]		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(80 718 states,	Runtime	16.26	4.17	3.70	9.41	10.66	11.18	10.31
1 797 353 transitions)	Speed-up	1	3.90	4.39	1.73	1.53	1.45	1.58
Train [151, 5] (ten routes,		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
672 174 states,	Runtime	408.51	240.55	207.57	830.49	828.12	253.00	186.54
2 244 486 transitions)	Speed-up	1	1.70	1.97	0.49	0.49	1.61	2.19
sort_1000 [200]		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(500 501 states,	Runtime	183.23	373.73	37.76	> 3600	106.30	820.43	112.88
500 502 transitions)	Speed-up	1	0.49	4.85	< 0.05	1.72	0.22	1.62
N-Queens with N=4		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(4 states,	Runtime	0.04	74.67	71.25	19.55	20.03	15.69	15.75
6 transitions)	Speed-up	1	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01

Table 8.2.: Model Checking Runtimes (ProB, Generated Java, Generated JavaScript, and Generated C++ Code) in Seconds with Size of State Space (states and transitions), Speed-Up Relative to ProB, OP = Operation Reuse; Models from [231] were re-benchmarked for ProB, Java, and C++ with another device.

of all enabled transitions for different states; due to the limitations presented before, the model is not yet feasible for model checking with B2Program. We plan to tackle this problem in the near future. Still, for all the case studies, the performance of B2Program was sufficient for interactive exploration; there were also no problems with memory usage as they are about the same order of magnitude as Java. Note that SICStus Prolog lacks the JIT compiler for the ARM processor used here; in our previous model checking benchmarks [231] the JIT compiler was available and the results of ProB are slightly better (e.g., ProB is actually faster for Train than Java, Java + cache and C++, but slower than C++ + cache) but not fundamentally different.

8.8. Related Work

In the following, we compare this work with existing tools that integrate domain experts in the software development process.

Requirements. Automatic translation of natural language requirements makes it possible to involve domain experts more directly in the validation process. An example is the requirements language FRETish [91] supported by the tool FRET [90]. Using FRET, the domain expert can write FRETish requirements in natural language which are translated

to linear temporal logic (LTL). To further improve communication between modeler and domain expert, FRET supports visualizing and simulating the underlying LTL formulas. A similar approach is followed by the tool SPEAR [73]. In contrast, our work does not yet enable the domain expert to directly validate formal properties. Instead, the domain expert can run scenarios for certain properties, and inspect the behavior in a domain-specific visualization.

Other works support writing high-level domain-specific scenarios for execution on a formal model, e.g., Gherkin and Cucumber for Event-B to run scenarios using the ProB animator [212, 74]. This allows a domain expert to write scenarios in natural language, execute them, and check the behavior afterward. As the base of communication, modelers and domain experts must agree on the events' meaning in natural language. Furthermore, the AValLa language was introduced to write domain-specific scenarios in ASMs, and run them using AsmetaV [47]. Another ASM tool is ASM2C++ which translates ASMs to C++, and AValLa scenarios to BDD code targeting the generated C++ code [36]. In our approach, the domain expert first creates scenarios by interacting with the domain-specific VisB visualization. More recently, the dynamic export also allows domain experts to write description text for each operation that is executed to describe the effect. Thus, our base of communication is the VisB visualization, and the import/export of scenarios with feedback (in the form of description text).

Documentation. ProB Jupyter [87] provides a notebook interface for formal models (in B, Event-B, TLA+, etc.). It also supports generating HTML, LaTeX, and PDF documents from Jupyter notebooks. This way, it is also possible to generate validation documentation with explanatory texts. More recently, ProB Jupyter supports VisB domain-specific visualizations as used in this article.

The LaTeX mode [148] of ProB can be used to produce LaTeX documents, and to generate documentation with explanatory texts, visualizations and tables. It does not support VisB and domain-specific visualizations have to be created via LaTeX.

Visualizations. This work has already outlined how important (domain-specific) visualizations are to validate a formal model.

There are more visualization tools for the B method like BMotionWeb [133], BMotion-Studio [134], AnimB[15], Brama [206], JeB [167], and the animation function [155] in ProB. A detailed comparison between these tools and VisB (together with SimB's interactive simulation) is described in [243] and [236]. An important novelty of our approach is that we create stand-alone artefacts for domain experts. However, B2Program used for the dynamic export only supports a subset of the B language.

State space projection was introduced by Ladenberger and Leuschel [136] to enable validation to focus on a sub-component or particular aspect of a system. In future, we would like to incorporate projection diagrams into our approach.

PVSio-Web [241] is a tool for visualizing PVS models and creating prototypes, especially human-machine interfaces. This enables the user to assemble an interactive visualization

[15] `http://wiki.event-b.org/index.php/AnimB`

for the model. In our approach, VisB visualizations are created manually, i.e., by creating an SVG image in an editor such as Inkspace, and by writing the VisB glue file. Similar to using VisB together with SimB's *interactive simulation*, PVSio-Web also supports simulation underneath.

There are also tools to create prototypes for VDM-SL models [181, 182]. Similar to our work, those works also allow domain-specific visualization, animation, simulation, and recording scenarios. In addition to validation by users and domain experts, the VDM-SL tools also incorporate UI designers as stakeholders.

Simulators. JeB [167] supports animation, simulation and visualization by generating HTML with JavaScript from an Event-B model. The user can encode functions by hand to enable the execution of complex models. To ensure the reliability of the simulated traces, JeB's approach introduced the notion of *fidelity*.

In our approach, it is also possible to write additional code by hand. Compared to JeB, B2Program supports easy import and export of traces. While JeB translates Event-B models to JavaScript constructs which are then run by an *interpreter*, B2Program *translates* B models nearly one-to-one to TypeScript classes.

This work also generates an interpreter for SimB's timed probabilistic simulation with user interaction. As discussed by Vu et al. [237], SimB is also related to simulation tools such as JeB, Uppaal [29], the co-simulation tool INTO-CPS [223], the ASM simulation tool AsmetaS [82, 83] in the Asmeta toolset [84], and the VDM simulation tool Overture [141]. With the implementation of SimB in this work, it is now possible to use SimB together with VisB as in ProB2-UI. Simulation scenarios can thus be exported by both modelers and domain experts, and shared between them.

OPEN/CÆSAR [81] is a language-independent open software architecture for concurrent systems which allows verification, simulation, and testing. One of its features is an interactive simulator which works similarly to the *animation* feature in the operations view of our work. Enabled transitions are computed and shown to the user from which the user can choose one for execution. Additionally, we allow interaction with more realistic prototypes by combining domain-specific VisB visualization and timed probabilistic SimB simulation. In our work, the user can execute operations via the VisB visualization which can also trigger a simulation in real-time. To extend OPEN/CÆSAR by another language, one must implement a C compiler for this language against OPEN/CÆSAR's interface. To extend B2Program, one would have to translate this language to B models compatible with B2Program. This principle is also applied in ProB, where some languages like TLA+ [99] or Alloy [129] are translated to B. (It is, however, also possible to provide the operational semantics as Prolog rules, as is done for CSP [43]).

Code Generators. Related code generators to B2Program are code generators for B [38, 51, 227], Event-B [172, 49, 200, 69, 80], ASM [33] and VDM [119]. Detailed comparisons have already been made in previous work by Vu et al. [233, 231]

Model Checkers. Our implementation and the associated performance analysis in this work resulted in an additional model checking tool as a by-product, generating JavaScript model checking code for a B model. Thus, model checkers such as ProB [152], TLC [246, 100], SPIN [109], pyB [244], LTSmin [120] are also related work. A detailed comparison of these model checkers with B2Program's model checking code generation is discussed by Vu et al. [231] In this work, we have achieved a satisfactory level of performance with JavaScript model checking code, reaching runtimes within an order of magnitude as Java for most benchmarks (see Section 8.7).

From the domain expert's view, only the animator, but not the model checker is available via the HTML document. We have made this decision as model checking is a technique a domain expert is usually not familiar with.

8.9. Conclusion and Future Work

In this work, we presented two solutions to improve the communication between modelers and domain experts by creating "interactive validation documents": (1) a (mostly) static export of a trace to an HTML file, and (2) a fully dynamic export of a classical B machine to an HTML document. While the static export works for all formalisms in ProB, the dynamic export only works for classical B machines supported by B2Program. The static export is suitable to analyze one scenario or trace, and allows the user to step through the saved trace and inspect the various states of the trace. In contrast, the dynamic export is suitable when domain experts have to animate or simulate traces, e.g., to modify existing traces, or to validate entire requirements.

Both approaches use domain-specific visualizations to help a domain expert reason about the formal model. For the dynamic export, we extended B2Program to generate HTML and JavaScript code while incorporating VisB visualizations. This makes it possible to interact with the model and check its behavior without the knowledge of the modeling language and its tools. Communication between modelers and domain experts is eased by features for importing/exporting scenarios and writing description texts. As new features for supporting the validation process, it is now possible to run (interactive) SimB simulations. A user or domain expert can now simulate scenarios with timing and probabilistic properties, or evaluate the system's reaction to a user interaction. Overall, this work enables involving domain experts in the development and validation process more actively. Those aspects have been demonstrated by two case studies: a light system model from the automotive domain, and a landing gear case study from the aviation domain.

Furthermore, we discussed the limitations of the dynamic export and analyzed the performance of the generated JavaScript code from B2Program. For most benchmarks in JavaScript, we achieved runtimes within an order of magnitude as Java and C++; a few models are faster in JavaScript, and for others, it is the other way around. We also encountered a benchmark where JavaScript is around one or more orders of magnitude slower than C++ and Java. Compared to ProB, the performance of simulation and trace replay seems to be significantly better. For animation and model checking, some

models can be processed with JavaScript faster, while for others ProB achieves faster runtimes. With the operation caching feature in ProB, a strong performance boost could be achieved [150]. Furthermore, ProB is particularly efficient in models where constraint solving can be used well. Overall, the performance of all our case studies was good enough to be able to interact with the model in dynamic export.

B2Program is available at:

`https://github.com/favu100/b2program`

Case studies are available at:

`https://github.com/favu100/b2program/tree/master/visualizations`[16]

In the future, one could support state diagrams which are an important technique for domain-specific validation. To support a larger subset of SimB simulation, one could think about generating code for SimB instead of generating a SimB interpreter. Another possible future work is generating other application formats such as standalone JavaFX applications.

Acknowledgements. We would like to thank anonymous reviewers for their constructive feedback.

Author Contribution.

- Fabian Vu: Conceptualization, Writing, Original Draft, Review and Editing, Implementation (Dynamic Export), Maintenance, Material Preparation, Data Collection, Analysis
- Christopher Happe: Conceptualization, Implementation (Dynamic Export), Data Collection
- Michael Leuschel: Implementation (Static Export), Writing, Review and Editing, Supervision, Funding Acquisition, Project Administration, Material Preparation, Analysis

Funding. This research is part of the IVOIRE project funded by the "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF); grant # I 4744-N. Section 8.4 is part of the KI-LOK project funded by the "Bundesministerium für Wirtschaft und Energie"; grant # 19/21007E.

Open Access funding is enabled and organized by Projekt DEAL.

[16] Also accessible via `https://favu100.github.io/b2program/`.

9. Additional Improvements and Benchmarks

This chapter presents the improvements to the implementation of B2Program in Chapter 7 and Chapter 8.

Section 9.1 describes how we lift some restrictions on quantified constructs. Section 9.2 describes how we rewrite predicates with $\in$ and $\subseteq$, aiming for more efficient execution. Furthermore, the results for the caching technique presented in Chapter 7 and evaluated in both Chapter 7 and Chapter 8 are unsatisfactory. In Section 9.3, we implement operation reuse by Leuschel [150] for B2Program, aiming for more efficient caching. Finally, we evaluate the performance in Section 9.4.

9.1. Lifting Restrictions on Quantified Constructs

This section describes how we lift B2Program's restrictions on quantified constructs.

Constraining Predicates. Before we present the restrictions lifted in this chapter, we first recap the notion of a *constraining predicate* in B2Program (discussed in Section 3.5 of [233]). Constraining predicates are predicates for constraining free variables in quantified constructs. In the following, we assume that x is a free variable.

In the first case, a constraining predicate can assign x to V, i.e., the constraining predicate is $x = V$.

In the second case, a constraining predicate can be $x \in S$, $x \subset S$, or $x \subseteq S$, where S is a finite set. In this case, B2Program translates the constraining predicate to a *for-loop* where S provides the domain for x. Listing 9.1 shows the generated Java code from $\{a|a \in 1..10\}$, where the constraining predicate is $a \in 1..10$.

```
1: BSet<BInteger> _ic_set_0 = new BSet<BInteger>();
2: for(BInteger _ic_a_1 : BSet.interval(new BInteger(1), new BInteger(10))) {
3:     _ic_set_0 = _ic_set_0.union(new BSet<BInteger>(_ic_a_1));
4: }
```

Listing 9.1. Generated Java Code from $\{a|a \in 1..10\}$

Quantified Constructs supported by B2Program. Quantified constructs in B contain free variables $v_1, \ldots, v_n$ and a predicate P to constrain their values. Universal quantified predicates in B require an implication $P \Rightarrow Q$ as a predicate, where P is relevant for constraining free variables. B2Program requires P to be a conjunction of sub-predicates

$p_1, \ldots, p_m$, i.e., $P = p_1 \wedge \ldots \wedge p_m$, with $m \geq n$. $m \geq n$ must hold because B2Program requires each free variable to be assigned/constrained with exactly one predicate.

The quantified constructs we optimize for B2Program are:

- Quantified predicates:
  - $\forall\, v_1, \ldots, v_n.(p_1 \ \wedge \ \ldots \ \wedge \ p_m \Rightarrow Q)$
  - $\exists\, v_1, \ldots, v_n.(p_1 \ \wedge \ \ldots \wedge p_m)$
- Set comprehensions: $\{v_1, \ldots, v_n \mid p_1 \wedge \ldots \wedge p_m\}$
- Lambda expressions $\lambda\, v_1, \ldots, v_n.(p_1 \wedge \ldots \wedge p_m \mid E)$
- Generalized unions and intersections: $\bigcup_{p_1 \wedge \ldots \wedge p_m}^{v_1,\ldots,v_n.} E$ and $\bigcap_{p_1 \wedge \ldots \wedge p_m}^{v_1,\ldots,v_n.} E$
- Set products and summations: $\prod_{p_1 \wedge \ldots \wedge p_m}^{v_1,\ldots,v_n.} E$ and $\sum_{p_1 \wedge \ldots \wedge p_m}^{v_1,\ldots,v_n.} E$
- ANY substitutions: $\texttt{ANY}\ v_1, \ldots, v_n\ \texttt{WHERE}\ p_1 \ \wedge \ \ldots \ \wedge \ p_m\ \texttt{THEN}\ S\ \texttt{END}$
- Assignments by predicate: $v_1 : (p_1 \ \wedge \ \ldots \ \wedge \ p_m)$

As described in Chapter 7, B2Program generates code that computes outgoing transitions for operations with parameters $v_1, \ldots, v_n$ and a guard P by treating them similarly to the set comprehension $\{v_1, \ldots, v_n \mid P\}$. The computation of $\{v_1, \ldots, v_n \mid P\}$ stores the parameter values that enable the operation. B2Program considers guards in top-level substitutions in operations:

- PRE substitutions: $op(v_1, \ldots, v_n) = \ \texttt{PRE}\ p_1 \ \wedge \ \ldots \ \wedge \ p_m\ \texttt{THEN}\ S\ \texttt{END}$
- SELECT substitutions: $op(v_1, \ldots, v_n) = \ \texttt{SELECT}\ p_1 \ \wedge \ \ldots \ \wedge \ p_m\ \texttt{THEN}\ S\ \texttt{END}$

In Chapter 7 and Chapter 8, B2Program required the i-th predicate p_i to constrain exactly the free variable v_i. v_i could be used in p_j to constrain v_j for $j > i$.

Lifting Restrictions. We now lift this restriction for B2Program, requiring v_i to be constrained by a predicate p_j, but v_i no longer needs to be constrained by p_i specifically. Assuming p_j is the constraining predicate of v_i, and v_k is a free variable different from v_i, i.e., $i \neq k$, appearing in p_j. Then, v_k must have been constrained by another conjunct p_l, with $l < k$.

Now, B2Program tries to reorder the conjuncts using topological sort [54] so that p_l comes before p_k, i.e., $l < k$. If multiple predicates could constrain v_i, B2Program uses the left-most predicate that fulfills the conditions for constraining v_i. For instance, $x \in 1..y$ is used for constraining x in $y = 2 \ \wedge \ x \in 1..y \ \wedge \ x \in 1..z \ \wedge \ z = 10$, whereas $x \in 1..z$ is used for constraining x in $z = 10 \ \wedge \ x \in 1..y \ \wedge \ x \in 1..z \ \wedge \ y = 2$.

After lifting this restriction in this chapter, B2Program allows *pruning predicates*, which we describe now. When v_k is constrained by p_l and v_i is constrained next by p_j, all

predicates in between, i.e., $p_{l+1}, \ldots, p_{j-1}$ are *pruning predicates*. If there is a predicate in $p_{l+1}, \ldots, p_{j-1}$ evaluated to `false`, then it is not necessary to constrain $v_i, \ldots, v_n$ anymore. In particular, one does not need to evaluate predicates after p_h with $p_h \in \{p_{l+1}, \ldots, p_{j-1}\}$, which is the left-most predicate evaluated to `false`. Consequently, the generated code cuts off the evaluation of all predicates after p_h, including constraining $v_i, \ldots, v_n$.

Example 1. Consider the following set comprehension:

$$\{a, b | a \in 1..10 \wedge a \; mod \; 2 = 0 \wedge b \in 1..5\}.$$

The predicate in the set comprehension does not meet the restriction in Chapter 7 and Chapter 8 and was thus not supported previously. One had to rewrite this set comprehension to $\{a, b | a \in 1..10 \wedge b \in 1..5 \wedge a \; mod \; 2 = 0\}$. Code generation results in the Java code shown in Listing 9.2. The generated code contains *for-loops* that iterate over all free variables before checking other predicates.

```
1: BRelation<BInteger, BInteger> _ic_set_0 = new BRelation<BInteger, BInteger>();
2: for(BInteger _ic_a_1 : BSet.interval(new BInteger(1), new BInteger(10))) {
3:   for(BInteger _ic_b_1 : BSet.interval(new BInteger(1), new BInteger(5))) {
4:     if((_ic_a_1.modulo(new BInteger(2)).equal(new BInteger(0))).booleanValue()) {
5:       _ic_set_0 = _ic_set_0.union(new BRelation<BInteger, BInteger>(new BTuple<>(
             _ic_a_1, _ic_b_1)));
6:     }
7:   }
8: }
```

Listing 9.2. Generated Java Code from $\{a, b | a \in 1..10 \wedge b \in 1..5 \wedge a \; mod \; 2 = 0\}$

After lifting this restriction, B2Program can generate code for this set comprehension without manual rewriting. The if statement in Listing 9.3 skips the iteration for `b` when `a mod 2 = 0` is false, i.e., `a mod 2 = 0` is a pruning predicate.

```
1: BRelation<BInteger, BInteger> _ic_set_0 = new BRelation<BInteger, BInteger>();
2: for(BInteger _ic_a_1 : BSet.interval(new BInteger(1), new BInteger(10))) {
3:   if((_ic_a_1.modulo(new BInteger(2)).equal(new BInteger(0))).booleanValue()) {
4:     for(BInteger _ic_b_1 : BSet.interval(new BInteger(1), new BInteger(5))) {
5:       _ic_set_0 = _ic_set_0.union(new BRelation<BInteger, BInteger>(new BTuple<>(
             _ic_a_1, _ic_b_1)));
6:     }
7:   }
8: }
```

Listing 9.3. Generated Java Code from $\{a, b | a \in 1..10 \wedge a \; mod \; 2 = 0 \wedge b \in 1..5\}$

Example 2. Let us take a look at another example where it is necessary to reorder conjuncts: $\{a, b | a \in 1..b \wedge a \; mod \; 2 = 0 \wedge b \in 1..5\}$. a is constrained by a predicate further to the left than b in this set comprehension, but B2Program needs the value of b for constraining a. B2Program rewrites the set comprehension to: $\{a, b | b \in 1..5 \wedge a \in 1..b \wedge a \; mod \; 2 = 0\}$, generating the code shown in Listing 9.4.

```
1: BRelation<BInteger, BInteger> _ic_set_0 = new BRelation<BInteger, BInteger>();
2: for(BInteger _ic_b_1 : BSet.interval(new BInteger(1), new BInteger(5))) {
3:   for(BInteger _ic_a_1 : BSet.interval(new BInteger(1), _ic_b_1)) {
4:     if((_ic_a_1.modulo(new BInteger(2)).equal(new BInteger(0))).booleanValue()) {
5:       _ic_set_0 = _ic_set_0.union(new BRelation<BInteger, BInteger>(new BTuple<>(
             _ic_a_1, _ic_b_1)));
6:     }
7:   }
8: }
```

Listing 9.4. Generated Java Code from $\{a, b | a \in 1..b \wedge a \ mod \ 2 = 0 \wedge b \in 1..5\}$

Predicate	Rewritten predicate	Negated predicate	Rewritten negation
$x \in \mathtt{NAT}$	$x \in \mathtt{0..MAXINT}$	$x \notin \mathtt{NAT}$	$x \notin \mathtt{0..MAXINT}$
$x \in \mathtt{NAT1}$	$x \in \mathtt{1..MAXINT}$	$x \notin \mathtt{NAT1}$	$x \notin \mathtt{1..MAXINT}$
$x \in \mathtt{INT}$	$x \in \mathtt{MININT..MAXINT}$	$x \notin \mathtt{INT}$	$x \notin \mathtt{MININT..MAXINT}$
$x \in \{\mathtt{e}_1, \ldots, \mathtt{e}_n\}$	$x = \mathtt{e}_1 \vee \ldots \vee x = \mathtt{e}_n$	$x \notin \{\mathtt{e}_1, \ldots, \mathtt{e}_n\}$	$x \neq \mathtt{e}_1 \wedge \ldots \wedge x \neq \mathtt{e}_n$
$x \in \mathtt{m..n}$	$x \geq m \wedge x \leq n$	$x \notin \mathtt{m..n}$	$x < m \vee x > n$
$x \in \mathtt{A} \cup \mathtt{B}$	$x \in \mathtt{A} \vee x \in \mathtt{B}$	$x \notin \mathtt{A} \cup \mathtt{B}$	$x \notin \mathtt{A} \wedge x \notin \mathtt{B}$
$x \in \mathtt{A} \cap \mathtt{B}$	$x \in \mathtt{A} \wedge x \in \mathtt{B}$	$x \notin \mathtt{A} \cap \mathtt{B}$	$x \notin \mathtt{A} \vee x \notin \mathtt{B}$
$x \in \mathtt{A} \setminus \mathtt{B}$	$x \in \mathtt{A} \wedge x \notin \mathtt{B}$	$x \notin \mathtt{A} \setminus \mathtt{B}$	$x \notin \mathtt{A} \vee x \in \mathtt{B}$
$x \in \mathbb{P}(\mathtt{A})$	$x \subseteq \mathtt{A}$	$x \notin \mathbb{P}(\mathtt{A})$	$x \not\subseteq \mathtt{A}$
$x \in \mathtt{FIN}(\mathtt{A})$	$x \subseteq \mathtt{A}$	$x \notin \mathtt{FIN}(\mathtt{A})$	$x \not\subseteq \mathtt{A}$
$x \in \mathbb{P}_1(\mathtt{A})$	$x \neq \varnothing \wedge x \subseteq \mathtt{A}$	$x \notin \mathbb{P}_1(\mathtt{A})$	$x = \varnothing \vee x \not\subseteq \mathtt{A}$
$x \in \mathtt{FIN}_1(\mathtt{A})$	$x \neq \varnothing \wedge x \subseteq \mathtt{A}$	$x \notin \mathtt{FIN}_1(\mathtt{A})$	$x = \varnothing \vee x \not\subseteq \mathtt{A}$
$x \in \mathtt{id}(\mathtt{A})$	$\mathtt{prj1}(x) \in \mathtt{A} \wedge$ $\mathtt{prj1}(x) = \mathtt{prj2}(x)$	$x \notin \mathtt{id}(\mathtt{A})$	$\mathtt{prj1}(x) \notin \mathtt{A} \vee$ $\mathtt{prj1}(x) \neq \mathtt{prj2}(x)$
$x \in \mathtt{A}^{\sim}$	$\mathtt{prj2}(x) \mapsto$ $\mathtt{prj1}(x) \in \mathtt{A}$	$x \notin \mathtt{A}^{\sim}$	$\mathtt{prj2}(x) \mapsto$ $\mathtt{prj1}(x) \notin \mathtt{A}$
$x \in \mathtt{A} \times \mathtt{B}$	$\mathtt{prj1}(x) \in \mathtt{A} \wedge$ $\mathtt{prj2}(x) \in \mathtt{B}$	$x \notin \mathtt{A} \times \mathtt{B}$	$\mathtt{prj1}(x) \notin \mathtt{A} \vee$ $\mathtt{prj2}(x) \notin \mathtt{B}$
$x \in \mathtt{A} \lhd \mathtt{r}$	$\mathtt{x} \in \mathtt{r} \wedge \mathtt{prj1}(x) \in \mathtt{A}$	$x \notin \mathtt{A} \lhd \mathtt{r}$	$\mathtt{x} \notin \mathtt{r} \vee \mathtt{prj1}(x) \notin \mathtt{A}$
$x \in \mathtt{A} \lhd \mathtt{r}$	$\mathtt{x} \in \mathtt{r} \wedge \mathtt{prj1}(x) \notin \mathtt{A}$	$x \notin \mathtt{A} \lhd \mathtt{r}$	$\mathtt{x} \notin \mathtt{r} \vee \mathtt{prj1}(x) \in \mathtt{A}$
$x \in \mathtt{r} \rhd \mathtt{A}$	$\mathtt{x} \in \mathtt{r} \wedge \mathtt{prj2}(x) \in \mathtt{A}$	$x \notin \mathtt{r} \rhd \mathtt{A}$	$\mathtt{x} \notin \mathtt{r} \vee \mathtt{prj2}(x) \notin \mathtt{A}$
$x \in \mathtt{r} \rhd \mathtt{A}$	$\mathtt{x} \in \mathtt{r} \wedge \mathtt{prj2}(x) \notin \mathtt{A}$	$x \notin \mathtt{r} \rhd \mathtt{A}$	$\mathtt{x} \notin \mathtt{r} \vee \mathtt{prj2}(x) \in \mathtt{A}$
$x \in \mathtt{A} \otimes \mathtt{B}$	$\mathtt{prj1}(x) \mapsto$ $\mathtt{prj1}(\mathtt{prj2}(x)) \in \mathtt{A} \wedge$ $\mathtt{prj1}(x) \mapsto$ $\mathtt{prj2}(\mathtt{prj2}(x)) \in \mathtt{B}$	$x \notin \mathtt{A} \otimes \mathtt{B}$	$\mathtt{prj1}(x) \mapsto$ $\mathtt{prj1}(\mathtt{prj2}(x)) \notin \mathtt{A} \vee$ $\mathtt{prj1}(x) \mapsto$ $\mathtt{prj2}(\mathtt{prj2}(x)) \notin \mathtt{B}$
$x \in \mathtt{A} \parallel \mathtt{B}$	$\mathtt{prj1}(\mathtt{prj1}(x)) \mapsto$ $\mathtt{prj1}(\mathtt{prj2}(x)) \in \mathtt{A} \wedge$ $\mathtt{prj2}(\mathtt{prj1}(x)) \mapsto$ $\mathtt{prj2}(\mathtt{prj2}(x)) \in \mathtt{B}$	$x \notin \mathtt{A} \parallel \mathtt{B}$	$\mathtt{prj1}(\mathtt{prj1}(x)) \mapsto$ $\mathtt{prj1}(\mathtt{prj2}(x)) \notin \mathtt{A} \vee$ $\mathtt{prj2}(\mathtt{prj1}(x)) \mapsto$ $\mathtt{prj2}(\mathtt{prj2}(x)) \notin \mathtt{B}$
$x \in \mathtt{seq}(\mathtt{A})$	$x \in \mathtt{1..card(x)} \rightarrow \mathtt{A}$	$x \notin \mathtt{seq}(\mathtt{A})$	$x \notin \mathtt{1..card(x)} \rightarrow \mathtt{A}$
$x \in \mathtt{seq1}(\mathtt{A})$	$x \neq \varnothing \wedge x \in \mathtt{1..card(x)} \rightarrow \mathtt{A}$	$x \notin \mathtt{seq1}(\mathtt{A})$	$x = \varnothing \vee x \notin \mathtt{1..card(x)} \rightarrow \mathtt{A}$
$x \in \mathtt{iseq}(\mathtt{A})$	$x \in \mathtt{1..card(x)} \rightarrowtail \mathtt{A}$	$x \notin \mathtt{iseq}(\mathtt{A})$	$x \notin \mathtt{1..card(x)} \rightarrowtail \mathtt{A}$
$x \in \mathtt{iseq1}(\mathtt{A})$	$x \neq \varnothing \wedge x \in \mathtt{1..card(x)} \rightarrowtail \mathtt{A}$	$x \notin \mathtt{iseq1}(\mathtt{A})$	$x = \varnothing \vee x \notin \mathtt{1..card(x)} \rightarrowtail \mathtt{A}$
$x \in \mathtt{perm}(\mathtt{A})$	$x \in \mathtt{1..card(x)} \twoheadrightarrow \mathtt{A}$	$x \notin \mathtt{perm}(\mathtt{A})$	$x \notin \mathtt{1..card(x)} \twoheadrightarrow \mathtt{A}$
$x \in \mathtt{A} \leftrightarrow \mathtt{B}$	$x \in \mathbb{P}(\mathtt{A} \times \mathtt{B})$	$x \notin \mathtt{A} \leftrightarrow \mathtt{B}$	$x \notin \mathbb{P}(\mathtt{A} \times \mathtt{B})$
$x \subseteq \mathtt{A}$	$\forall v.(v \in x \Rightarrow v \in A)$	$x \not\subseteq \mathtt{A}$	$\exists v.(v \in x \wedge v \notin A)$

Table 9.1.: Overview of rewrites for B predicates with $\in$, $\subseteq$, $\notin$, and $\not\subseteq$ (if they are not *constraining predicates*). Rewrites inspired by the fact that ProB [98] also rewrites predicates with $\in$, $\subseteq$, $\notin$, and $\not\subseteq$.

9.2. Rewriting Predicates with Set Membership and Subset of

This section describes how B2Program rewrites predicates with set membership ($\in$, $\notin$) and subset of ($\subseteq$, $\not\subseteq$). Some rewrites also lift limitations on infinite sets. Table 9.1 shows an overview of rewrites.

Note that B2Program does not allow x to store an infinite set. Therefore, `FIN(A)` and `FIN1(A)` can be treated similarly to $\mathbb{P}$(`A`) and $\mathbb{P}_1$(`A`), respectively. Furthermore, one can simply access `card(x)` without WD errors.

We rewrite a predicate only if it is not a *constraining predicate* (discussions in Section 9.1). For instance, B2Program does not rewrite $a \in 1..b$ in Listing 9.3 to $a \geq 1 \ \wedge \ a \leq b$ because B2Program (1) uses this predicate for constraining a and (2) requires a set on the right-hand side of $\in$ to constrain a free variable.

ProB also optimizes certain predicates with $\in$, $\subseteq$, $\notin$, and $\not\subseteq$ specifically to improve performance [98]. This fact inspired us to rewrite certain predicates with $\in$, $\subseteq$, $\notin$ and $\not\subseteq$ for B2Program to handle these constructs more efficiently.

Example 3. Let us consider an example with x as a variable:

$$x \in (\mathbb{N} \cap 1..5000) \cup (0..1000000) \wedge x \in \{1, 2, 3\}$$

The predicate contains a set operation on an infinite set: $\mathbb{N} \cap 1..5000$, which B2Program did not support in Chapter 7 and Chapter 8. Some large sets, i.e., 1..5000 and 0..1000000, would lead to a performance overhead. The predicate also contains set operations on large sets, such as $\cup$ and $\cap$. B2Program now rewrites this predicate as follows, before generating code:

$$\begin{aligned}
& x \in (\mathbb{N} \cap 1..5000) \cup (0..1000000) \wedge (x \in \{1, 2, 3\}) \\
\equiv\ & (x \in (\mathbb{N} \cap 1..5000) \vee (x \in 0..1000000)) \wedge (x = 1 \vee x = 2 \vee x = 3) \\
\equiv\ & ((x \in \mathbb{N} \wedge x \in 1..5000) \vee (x \geq 0 \wedge x \leq 1000000)) \wedge (x = 1 \vee x = 2 \vee x = 3) \\
\equiv\ & ((x \geq 0 \wedge x \geq 1 \wedge x \leq 5000) \vee (x \geq 0 \wedge x \leq 1000000)) \wedge (x = 1 \vee x = 2 \vee x = 3)
\end{aligned}$$

The rewritten predicate no longer contains infinite sets or large sets. Note that the computation of the rewritten predicate could be more expensive than the original predicate (assuming the infinite set is not there) if x is not a variable and its computation is costly, as x might need to be computed eight times in the worst case.

More Optimizations. As of Chapter 7 and Chapter 8, B2Program supported some predicates with $\mathtt{x} \in \mathtt{E}$ and $\mathtt{x} \notin \mathtt{E}$, without computing `E` explicitly: $\mathtt{x} \in \mathtt{A} \rightarrow \mathtt{B}$, $\mathtt{x} \in \mathtt{A} \nrightarrow \mathtt{B}$, $\mathtt{x} \in \mathtt{A} \rightarrowtail \mathtt{B}$, $\mathtt{x} \in \mathtt{A} \mapsto\!\!\!\!\!\nrightarrow \mathtt{B}$, $\mathtt{x} \in \mathtt{A} \twoheadrightarrow \mathtt{B}$, $\mathtt{x} \in \mathtt{A} \twoheadrightarrow\!\!\!\!\!\!\!+\ \mathtt{B}$, $\mathtt{x} \in \mathtt{A} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathtt{B}$, and their negated counterparts. Those predicates were not allowed as *constraining predicates*, as the generated code does not explicitly compute `E` for the generated *for-loop* to iterate over.

In this chapter, we optimize more predicates $\mathtt{x} \in \mathtt{E}$ and $\mathtt{x} \notin \mathtt{E}$, so that the generated code does not explicitly compute $\mathtt{E}$. In particular, the generated code directly checks whether $\mathtt{x}$ is in $\mathtt{E}$. With the optimizations in this chapter, B2Program treats some more predicates, i.e., $\mathtt{x} \in \mathtt{A}^*$, $\mathtt{x} \in \mathtt{A}^+$, $\mathtt{x} \in \mathtt{dom}(\mathtt{A})$, $\mathtt{x} \in \mathtt{ran}(\mathtt{A})$, $\mathtt{x} \in \mathtt{r}[\mathtt{A}]$, $\mathtt{x} \in \mathtt{A};\mathtt{B}$, and their negated counterparts, differently from Chapter 7 and Chapter 8. These improvements only apply when B2Program does not treat the predicate as a *constraining predicate*. Note that B2Program supports these expressions on the right-hand side of $\in$ in a *constraining predicate*, unlike the operators named at the beginning of the paragraph: $\rightarrow$, $\nrightarrow$, $\rightarrowtail$, etc.

Example 4. Let us look at the example $\mathtt{x} \in \mathtt{dom}(\mathtt{A})$. In Chapter 7 and Chapter 8, the generated code iterated over $\mathtt{A}$ to compute $\mathtt{dom}(\mathtt{A})$ and then checked whether $\mathtt{x} \in \mathtt{dom}(\mathtt{A})$. The generated Java code for $\mathtt{x} \in \mathtt{dom}(\mathtt{A})$ was `A.domain().elementOf(x)`.

In this chapter, the generated code checks $\mathtt{x} \in \mathtt{dom}(\mathtt{A})$ without computing $\mathtt{dom}(\mathtt{A})$. The libraries for B2Program implement *relations* with *hashmaps* [233]. Thus, the generated code checks whether $\mathtt{x}$ is in the *keyset* of $\mathtt{A}$. The generated Java code is `A.isInDomain(x)`.

9.3. Improvements on Caching

This section describes how B2Program implements the operation reuse technique by Leuschel [150] in the generated code. The caching technique applies to the operations' guards, the operations' effects, and the invariants.

In the following, we use the notion $s \xrightarrow{op(\alpha)} s'$ from [150] to describe a transition from state s to s' by executing the operation $\mathtt{op}$ with parameters α, i.e., $\mathtt{op}(\alpha)$.

9.3.1. Caching of Operation Effects

As of Chapter 7 and Chapter 8, the code generated by B2Program did not cache and reuse the effects of the operations. In the following, we explain how to cache the operations' effects according to Leuschel [150]. We use the following notions corresponding to [150]:

- $\mathtt{reads}(\mathtt{op})$ - the set of variables and constants read in the operation $\mathtt{op}$
- $\mathtt{writes}(\mathtt{op})$ - the set of variables and constants written in the operation $\mathtt{op}$
- $\Delta(\mathtt{s}', \mathtt{op}) = \mathtt{writes}(\mathtt{op}) \lhd \mathtt{s}'$ - the update function storing the effect of $\mathtt{op}$ when reaching $\mathtt{s}'$ after executing $\mathtt{op}$
- $\mathtt{Cache}_{\mathtt{op}}$ - a cache for each operation $\mathtt{op}$, storing operation updates
- $\mathtt{s}_{\mathtt{proj}} = \mathtt{reads}(\mathtt{op}) \lhd \mathtt{s}$ - a projection of a state $\mathtt{s}$ onto the read variables of $\mathtt{op}$

B2Program now implements caching of operation updates similarly to [150]: During model checking, a new state $\mathtt{s}'$ is reached from a previous state $\mathtt{s}$ by executing $\mathtt{op}(\alpha)$. To cache the effect of $\mathtt{op}(\alpha)$, we create a projection of $\mathtt{s}$ on $\mathtt{reads}(\mathtt{op})$, resulting in $\mathtt{s}_{\mathtt{proj}}$.

The generated code then checks whether $\mathtt{s}_{\mathtt{proj}} \in \mathtt{dom}(\mathtt{Cache}_{\mathtt{op}})$. If true, one can access the updates via $\mathtt{Cache}_{\mathtt{op}}(\mathtt{s}_{\mathtt{proj}})$ to compute $\mathtt{s}'$. Otherwise, execute $\mathtt{op}(\alpha)$ to compute $\Delta(\mathtt{s}', \mathtt{op})$. Finally, update $\mathtt{Cache}_{\mathtt{op}}(\mathtt{s}_{\mathtt{proj}}) := \Delta(\mathtt{s}', \mathtt{op})$.

Remark: If $\mathtt{s}_{\mathtt{proj}} \in \mathtt{dom}(\mathtt{Cache}_{\mathtt{op}})$ is true, then we know that $\mathtt{s}_{\mathtt{proj}}$ was added into $\mathtt{Cache}_{\mathtt{op}}(\mathtt{s}_{\mathtt{proj}})$ by another state $\mathtt{s}_{\mathtt{other}}$ where both the projection of $\mathtt{s}$ and $\mathtt{s}_{\mathtt{other}}$ result in the same projected state $\mathtt{s}_{\mathtt{proj}}$.

9.3.2. Caching of Evaluated Transitions

Like [150], we also cache the evaluation of the transitions. Our previous implementation caches evaluated transitions as follows (see Chapter 7): Given a transition $s \xrightarrow{op(\alpha)} s'$ where s' is visited the first time, and given $\mathtt{grd}_{\mathtt{op}}$, the *guard* of an operation $\mathtt{op}$: For each operation, we reuse the computation of outgoing transitions for $\mathtt{grd}_{\mathtt{op}}$ from s if executing $\mathtt{op}(\alpha)$ in state s does not modify the variables read in $\mathtt{grd}_{\mathtt{op}}$. However, the results in Chapter 7 showed that this technique is inefficient as caching only applies in this specific case. We use the following notions (similar to [150]) to describe the improved caching of evaluated transitions after [150]:

- $\mathtt{reads}_{\mathtt{grd}}(\mathtt{op})$ [150] - the set of variables and constants read in the *guard* of the operation $\mathtt{op}$
- $\mathtt{Cache}_{\mathtt{op,trans}}$ - a cache for each operation $\mathtt{op}$, storing outgoing transitions
- $\mathtt{s}_{\mathtt{grd}} = \mathtt{reads}_{\mathtt{grd}}(\mathtt{op}) \lhd \mathtt{s}$ [150] - a projection of a state $\mathtt{s}$ onto the read variables in the guard of $\mathtt{op}$
- $\mathtt{compute\_trans}(\mathtt{op}, \mathtt{s})$ - the computation of all outgoing transitions for operation $\mathtt{op}$ in state $\mathtt{s}$.

B2Program now implements caching of computed transitions similarly to [150]: When visiting a new state $\mathtt{s}$ during model checking, the algorithm computes all outgoing transitions to explore the succeeding states. To cache the computation of the outgoing transitions for an operation $\mathtt{op}$, we create a projection of $\mathtt{s}$ on $\mathtt{reads}_{\mathtt{grd}}(\mathtt{op})$, resulting in $\mathtt{s}_{\mathtt{grd}}$. The generated code then checks whether $\mathtt{s}_{\mathtt{grd}} \in \mathtt{dom}(\mathtt{Cache}_{\mathtt{op,trans}})$. If true, one can access the evaluated transitions via $\mathtt{Cache}_{\mathtt{op,trans}}(\mathtt{s}_{\mathtt{grd}})$. Otherwise, compute the outgoing transitions and update $\mathtt{Cache}_{\mathtt{op,trans}}(\mathtt{s}_{\mathtt{grd}}) := \mathtt{compute\_trans}(\mathtt{op}, \mathtt{s})$.

Remark: If $\mathtt{s}_{\mathtt{grd}} \in \mathtt{dom}(\mathtt{Cache}_{\mathtt{op,trans}})$ is true, then we know that $\mathtt{s}_{\mathtt{grd}}$ was added into $\mathtt{Cache}_{\mathtt{op,trans}}(\mathtt{s}_{\mathtt{grd}})$ by another state $\mathtt{s}_{\mathtt{other}}$ where both the projection of $\mathtt{s}$ and $\mathtt{s}_{\mathtt{other}}$ result in the same projected state $\mathtt{s}_{\mathtt{grd}}$.

9.3.3. Caching of Invariant Conjuncts

Like [150], we also cache the evaluation of each invariant conjunct. Our previous implementation caches invariant conjuncts as follows (see Chapter 7): Given a transition $s \xrightarrow{op(\alpha)} s'$ where s' is visited the first time, and given n invariant conjuncts $i_1, \ldots, i_n$: For

each i_j, we reuse the evaluation of i_j from s if executing $\texttt{op}(\alpha)$ in state s does not modify the variables read in i_j. However, the results in Chapter 7 showed that this technique is inefficient as caching only applies in this specific case. We use the following notions (similar to [150]) to describe invariant caching:

- $\texttt{reads}_{\texttt{pred}}(\texttt{inv})$ - the set of variables and constants read in the invariant conjunct `inv`
- $\texttt{Cache}_{\texttt{inv}}$ - a cache for each invariant conjunct `inv`, storing the evaluated value
- $\texttt{s}_{\texttt{inv}} = \texttt{reads}_{\texttt{pred}}(\texttt{inv}) \lhd \texttt{s}$ - a projection of a state `s` onto the read variables in the invariant conjunct `inv`
- $\texttt{eval}(\texttt{inv}, \texttt{s})$ - evaluation of invariant conjunct `inv` in state `s`

B2Program now implements caching of invariants similarly to [150]: When visiting a new state `s` during model checking, the algorithm checks the invariant in this state. To cache invariants, we create a projection of `s` on $\texttt{reads}_{\texttt{pred}}(\texttt{inv})$ for each conjunct `inv`, resulting in $\texttt{s}_{\texttt{inv}}$. The generated code checks whether $\texttt{s}_{\texttt{inv}} \in \texttt{dom}(\texttt{Cache}_{\texttt{inv}})$. If true, one can access the invariant value via $\texttt{Cache}_{\texttt{inv}}(\texttt{s}_{\texttt{inv}})$. Otherwise, evaluate `inv` on `s` and update $\texttt{Cache}_{\texttt{inv}}(\texttt{s}_{\texttt{inv}}) := \texttt{eval}(\texttt{inv}, \texttt{s})$.

Remark: If $\texttt{s}_{\texttt{inv}} \in \texttt{dom}(\texttt{Cache}_{\texttt{inv}})$ is true, then we know that $\texttt{s}_{\texttt{inv}}$ was added into $\texttt{Cache}_{\texttt{inv}}(\texttt{s}_{\texttt{inv}})$ by another state $\texttt{s}_{\texttt{other}}$ where both the projection of `s` and $\texttt{s}_{\texttt{other}}$ result in the same projected state $\texttt{s}_{\texttt{inv}}$.

9.4. Benchmarks on Improvements

With the improvements made in this chapter, we run the model checking benchmarks from Chapter 7 and Chapter 8 again. Additionally, we benchmarked more machines:

- A formal B model for a real-time ethernet protocol (rether) [226] that we manually translated from Event-B. The Event-B model is a translation by Marc Büngener of a model for DiViNe. The formal model contains many set operations, mainly applied to relations.
- A formal B model containing parts of a Mercury Planetary Orbiter developed by Space Systems Finland [115] that we manually translated from their Event-B model. The formal model contains many relations and operations on these relations. Furthermore, the formal model contains multiple solutions for the `PROPERTIES` clause. As B2Program requires assigning all constants with the = operator, we instantiated the formal model with a specific configuration.
- A formal B model for a Communications-based Train Control system (CBTC) [168] [1]. The formal model contains many integers, sequences of integers, as well as comparisons, arithmetic operations and function calls.

[1] Also available at `https://mars-workshop.org/repository/020-CBTC.html`

The optimizations presented in this chapter resulted in changes to the generated code compared to those in Chapter 7 and Chapter 8. First, we can support many constructs in the machines in their original form due to the lifting of some restrictions in Section 9.1 and Section 9.2. For example, we can keep the original order of conjuncts in many quantified constructs (including predicates for pruning) and allow more expressions on the right-hand side of $\in$ and $\subseteq$. Second, the optimizations implemented in Section 9.2 resulted in different generated code for a few constructs compared to those in Chapter 7 and Chapter 8. These changes affect the machines: Cruise Controller, CAN BUS, NoTa, N-Queens, Sort, Train, and Landing Gear. For two machines, the performance (without caching) appears worse than in Chapter 7 and Chapter 8, although this is not the case: The Sort machine has two more invariants omitted in Chapter 7 and Chapter 8. We manually rewrote a few constructs in the Train model in Chapter 7 and Chapter 8 for B2Program to generate more efficient code. These rewrites are more complex than those in Section 9.2.

We benchmark the runtimes for parsing/translation, compilation, and single-threaded and multi-threaded model checking, and measure the memory consumption of model checking.

In particular, we compare the results of single-threaded model checking via B2Program with ProB [153] and TLC [246] (B machines translated via TLC4B [100] to TLA+). [2] For multi-threaded model checking, we compare Java and C++ code generated by B2Program with TLC, with eight threads for all. [3] We also present benchmarks with six threads in Section 9.5 (for B2Program and TLC), as done in Chapter 7. However, we run the benchmarks using a computer different from the one used in Chapter 7. We observe improvements up to eight threads on the computer used for the benchmarks in this chapter. Furthermore, we benchmark ProB and Java, JavaScript and C++ code generated by B2Program in the standard configuration (without operation reuse) and with operation reuse. [4] Detailed results of the benchmarks are shown in Section 9.5.

We run the benchmarks with ProB[5], TLC4B in version 2.06, Java[6], JavaScript[7], and C++[8]. In particular, we run each benchmark five times on a MacBook Pro (16 GB RAM, Apple M1 Pro Chip with eight cores[9]) with a one-hour timeout and then take the median values.[10] Note that SICStus Prolog lacks the JIT compiler for the ARM processor[11]. In the following, we will refer to ST as standard, i.e., "without operation reuse", and OP as "with operation reuse" for better readability.

[2] We configure `-p TLC_WORKERS 1` for TLC with one thread.
[3] We configure `-p TLC_WORKERS 8` for TLC with eight threads.
[4] We configure `-p OPERATION_REUSE false -p COMPRESSION TRUE` for ProB without operation reuse, and `-p OPERATION_REUSE full -p COMPRESSION TRUE` for ProB with operation reuse.
[5] ProB CLI 1.13.1-nightly built with SICStus 4.8.0 (arm64-darwin-20.1.0)
[6] OpenJDK 64-Bit Server VM (build 18.0.2+0, mixed mode, sharing)
[7] NodeJS 23.10.0
[8] Compiled with clang, version 16.0.6
[9] Six performance cores, two efficiency cores.
[10] Commit hash: `8f0ebc974dd98f63e91f6356de0be286b4ab8904` in `https://github.com/favu100/b2program`
[11] `https://sicstus.sics.se/download4.html`

Table 9.2.: Startup Overhead in Seconds (including Parsing, Translation and Compilation) of ProB, TLC, and Generated Code in Java, JavaScript (JS), and C++

Counter	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.51	0.69	0.31	0.30	0.31	0.31
Compilation	-	-	0.44	1.06	1.44	1.55
Cruise Controller (Volvo)	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.53	0.78	0.53	0.52	0.59	0.59
Compilation	-	-	0.77	1.24	9.23	10.36
CAN BUS (J. Colley)	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.52	0.79	0.44	0.44	0.47	0.47
Compilation	-	-	0.71	1.15	7.56	8.39
Landing Gear [135]	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.54	0.85	0.59	0.65	0.69	0.69
Compilation	-	-	0.84	1.29	12.89	14.18
NoTa [183]	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.54	0.82	0.48	0.46	0.53	0.53
Compilation	-	-	0.70	1.16	11.41	12.64
rether [226]	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.53	0.76	0.39	0.39	0.42	0.42
Compilation	-	-	0.57	1.11	4.58	5.05
Mercury Orbiter [115]	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.54	0.78	0.50	0.50	0.55	0.55
Compilation	-	-	0.72	1.16	10.78	11.84
CBTC Controller [168]	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.56	0.91	0.60	0.68	0.64	0.64
Compilation	-	-	0.73	1.29	4.46	6.70
Train [151, 5] (ten routes)	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.53	0.93	0.60	0.66	0.62	0.62
Compilation	-	-	0.68	1.18	6.70	7.42
sort_1000 [200]	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.53	0.81	0.36	0.42	0.37	0.37
Compilation	-	-	0.54	1.13	2.59	2.84
N-Queens with N=4	ProB	TLC	Java	JS	C++ -O1	C++ -O2
Parsing/Translation	0.51	0.78	0.35	0.41	0.36	0.36
Compilation	-	-	0.48	1.10	2.16	2.35

Startup Results. Table 9.2 presents the results for the startup overhead (parsing, translation, and compilation) for ProB, TLC, and the Java, JavaScript and C++ code generated by B2Program.

The results for startup overhead align with Chapter 7: ProB, TLC, Java, and JavaScript have a short startup time, whereas C++ has a longer startup time due to longer compilation time (in many cases, even longer than model checking time).

As described in Chapter 7, a modeler often makes changes in the machine and, therefore, has to recompile before model checking again. C++ is thus only efficient when the model checking runtime is significantly higher than the compilation time. For use cases where the generated code is compiled only once, e.g., domain-specific validation documents

in Chapter 8, the startup overhead with parsing, translation, and compilation becomes negligible.

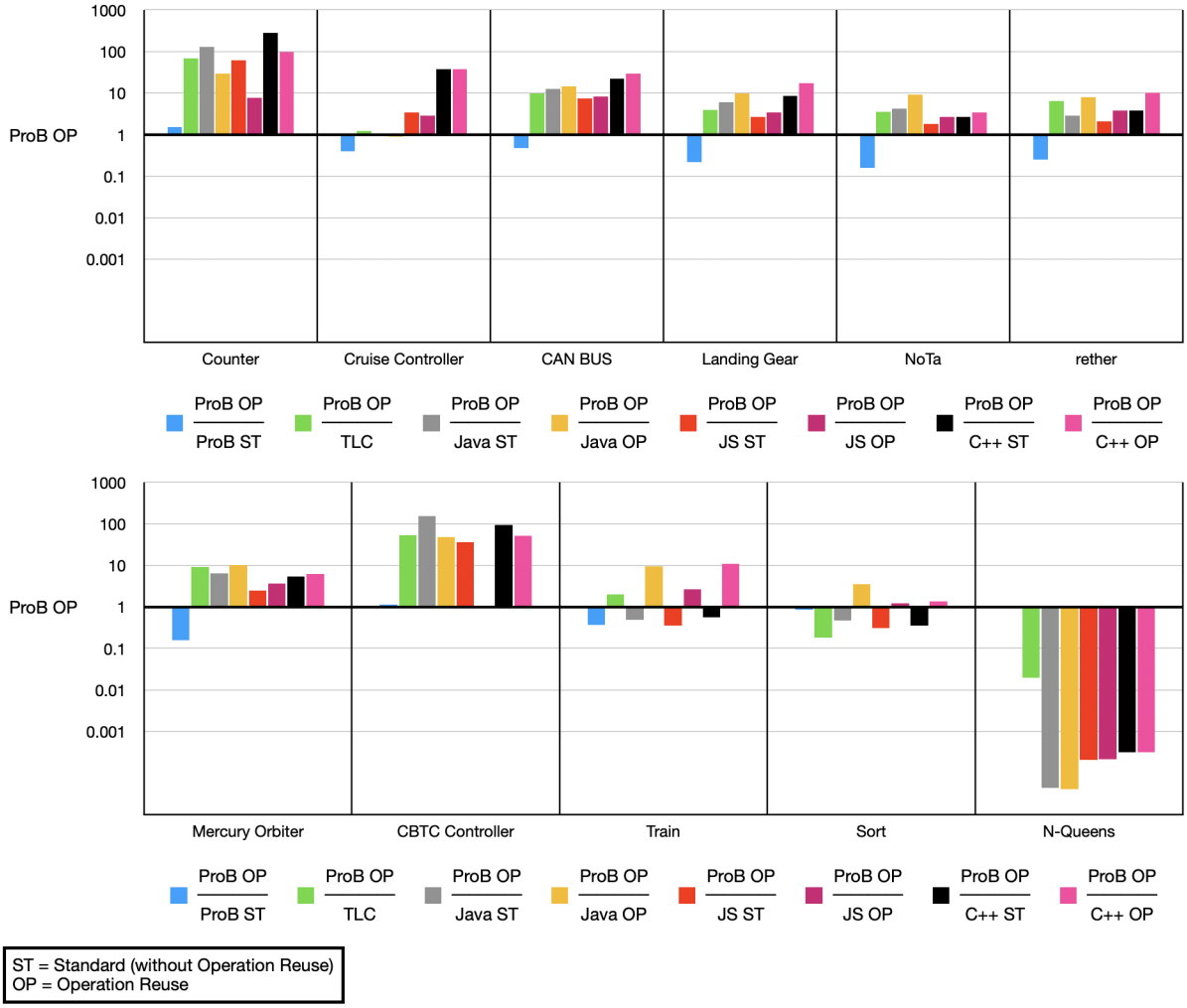


Figure 9.1.: Speedups of Single-threaded Model Checking for ProB ST, TLC, Generated Java, JavaScript (JS), and C++ Code Relative to ProB OP as Bar Charts

Single-Threaded Model Checking. Figure 9.1 shows the speedups of single-threaded model checking across various configurations: ProB ST, TLC, and generated Java, JavaScript, and C++ code via B2Program, all evaluated relative to ProB OP. For code generated by B2Program, we consider both configurations: ST and OP. Table 9.3 in Section 9.5 shows detailed results for runtimes and memory consumption.

Compared to ProB OP, the generated code achieves faster runtimes for most B machines. When activating operation reuse for the generated code, model checking with the generated code is even faster for most machines.

For Java, the detailed results are as follows:

- Java ST outperforms ProB OP for 7 out of 11 machines, achieving runtimes that are two orders of magnitude faster for 2 machines and one order of magnitude faster for another machine.
- ProB OP outperforms Java ST for 3 machines: Train, Sort, and N-Queens. For Train and Sort, Java OP achieves faster runtimes than ProB OP.
- Java OP outperforms ProB OP for 9 machines, achieving runtimes around one order of magnitude faster for 8 machines.

The generated JavaScript code achieves faster runtimes than ProB OP for most machines. Java outperforms JavaScript for most benchmarks. For JavaScript, the detailed results are as follows:

- JavaScript ST outperforms ProB OP for 8 machines, being one order of magnitude faster for 2 machines.
- ProB OP outperforms JavaScript ST for 3 machines: Train, Sort, and N-Queens. JavaScript OP achieves faster runtimes than ProB OP for Train and Sort.
- JavaScript OP achieves faster runtimes than ProB OP for 9 machines, whereas 1 machine is processed slower, and 1 machine runs out of memory.
- When (de)activating operation reuse for both Java and JavaScript, Java outperforms JavaScript for 9 machines, whereas JavaScript achieves faster runtimes for 2 machines.

The generated C++ code achieves faster runtimes than ProB OP for most machines. C++ outperforms JavaScript for all machines when (de)activating operation reuse for both. Moreover, C++ achieves faster runtimes than Java for more machines. Note that C++'s clang compiler performs optimizations, which lead to significantly higher compile times. In more detail:

- C++ ST achieves faster runtimes than ProB OP for 8 machines (for 3 of them one magnitude faster, for 2 machines even two magnitudes faster), whereas ProB OP achieves faster runtimes for 3 machines, namely, Train, Sort and N-Queens.
- C++ OP achieves faster runtimes than ProB OP for 10 machines. N-Queens is the only machine where ProB OP outperforms C++ OP.
- When (de)activating operation reuse for C++ and Java, C++ outperforms Java for 7 out of 11 machines, whereas Java achieves faster runtimes for 3 machines. For the CBTC Controller, C++ ST performs worse than Java ST, whereas C++ OP outperforms Java OP.

Across all languages, N-Queens is processed more efficiently by ProB than B2Program. The reason is ProB's constraint-solving capabilities, as discussed in Chapter 7. As expected, operation reuse cannot achieve faster runtimes for N-Queens.

The generated Java and C++ code performs better than TLC for some machines, whereas it is the opposite for others. In more detail:

- Java ST achieves faster runtimes than TLC for 6 out of 11 machines and worse runtimes for 5 machines.
- C++ ST also achieves faster runtimes than TLC for 6 out of 11 machines, whereas for 5 machines, the runtimes are worse.
- Java OP performs better than TLC for 7 out of 11 machines and worse for 4 machines.
- C++ OP performs better than TLC for 7 out of 11 machines. For 2 machines, the runtimes of TLC are faster, whereas TLC performs similarly to C++ OP for the other 2 machines.

TLC achieves faster runtimes than JavaScript for most machines. In more detail:

- JavaScript ST achieves faster runtimes than TLC only for 2 machines, whereas TLC is faster for the other 9 machines.
- JavaScript OP achieves faster runtimes than TLC for 3 machines, whereas TLC outperforms JavaScript OP for the other 8 machines.

Multi-Threaded Model Checking. Figure 9.2 shows the speedups of multi-threaded model checking with eight threads in Java and C++ relative to TLC with eight threads. Table 9.5 in Section 9.5 provides more details on runtimes and memory consumption. Note that we run the benchmarks in this chapter on a different computer from those in Chapter 7.

TLC with eight threads achieves faster runtimes than Java and C++ with eight threads when operation reuse is deactivated. In more detail:

- Java ST achieves faster runtimes than TLC only for 1 machine (Sort), whereas the runtimes are similar for another machine (CBTC Controller). TLC outperforms the generated Java code for 9 machines.
- C++ ST outperforms TLC only for 2 machines, whereas TLC achieves faster runtimes for the other 9 machines.

Java OP outperforms TLC for some machines, whereas it is the opposite for others. Although operation reuse improves the performance of C++ with eight threads, TLC still achieves faster runtimes for more machines. In more detail:

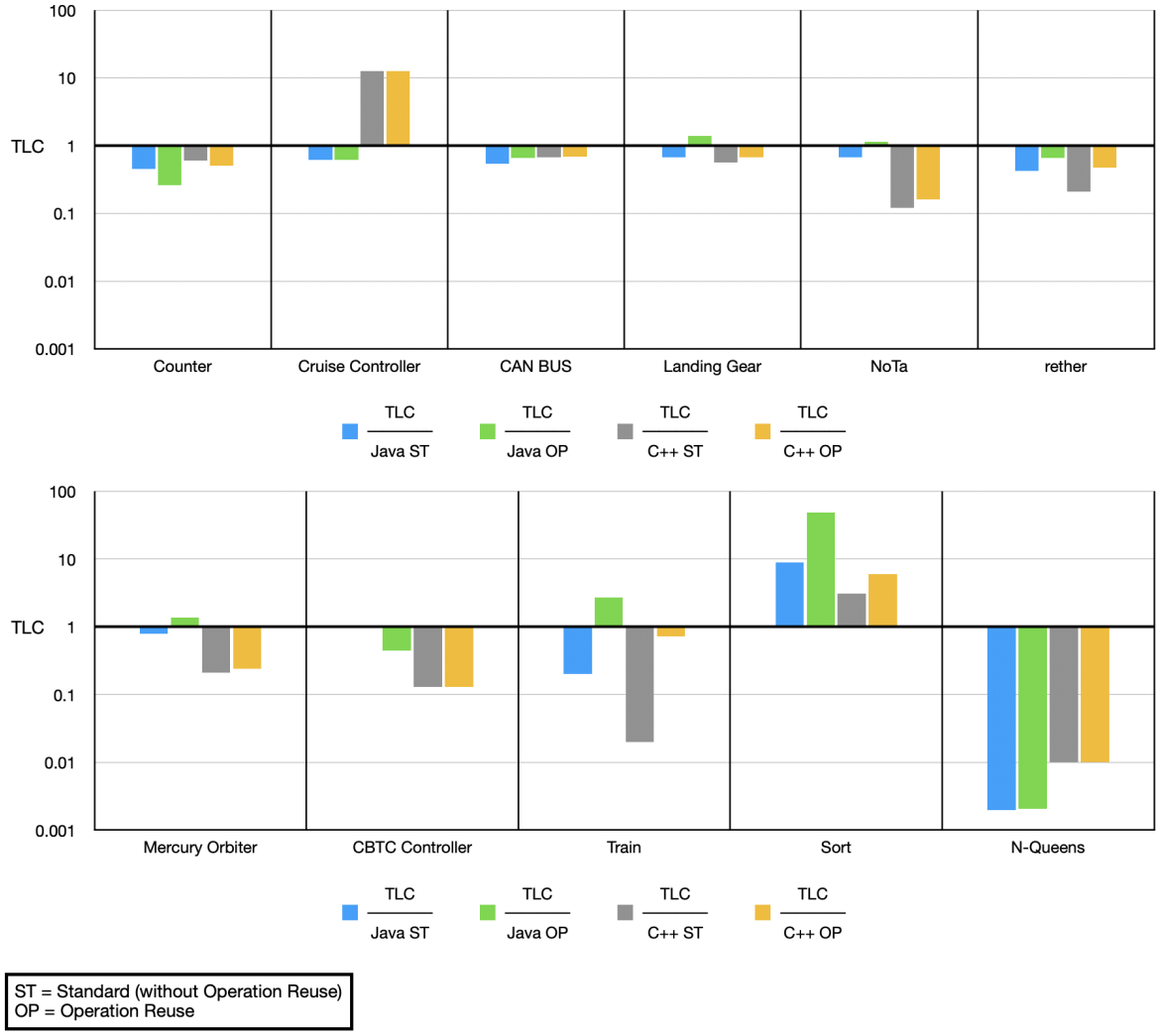


Figure 9.2.: Speedups of Multi-Threaded Model Checking for Java and C++ with 8 Threads Relative to TLC with 8 Threads as Bar Charts

- Java OP outperforms TLC for 5 machines, whereas TLC achieves faster runtimes for 6 machines.
- C++ OP outperforms TLC only for 2 machines (Cruise Controller and Sort), whereas TLC performs better for the other 9 machines.

Figure 9.3 shows the speedups of eight threads relative to one thread, i.e., we compare `X` with eight threads vs. `X` with one thread for all `X` $\in$ {TLC, Java ST, Java OP, C++ ST, C++ OP}, where ST means standard (without operation reuse) and OP means with operation reuse.

The results show that parallelization works more efficiently for TLC than for Java and C++ code generated by B2Program, i.e., TLC achieves higher speedups through parallelization than B2Program does.

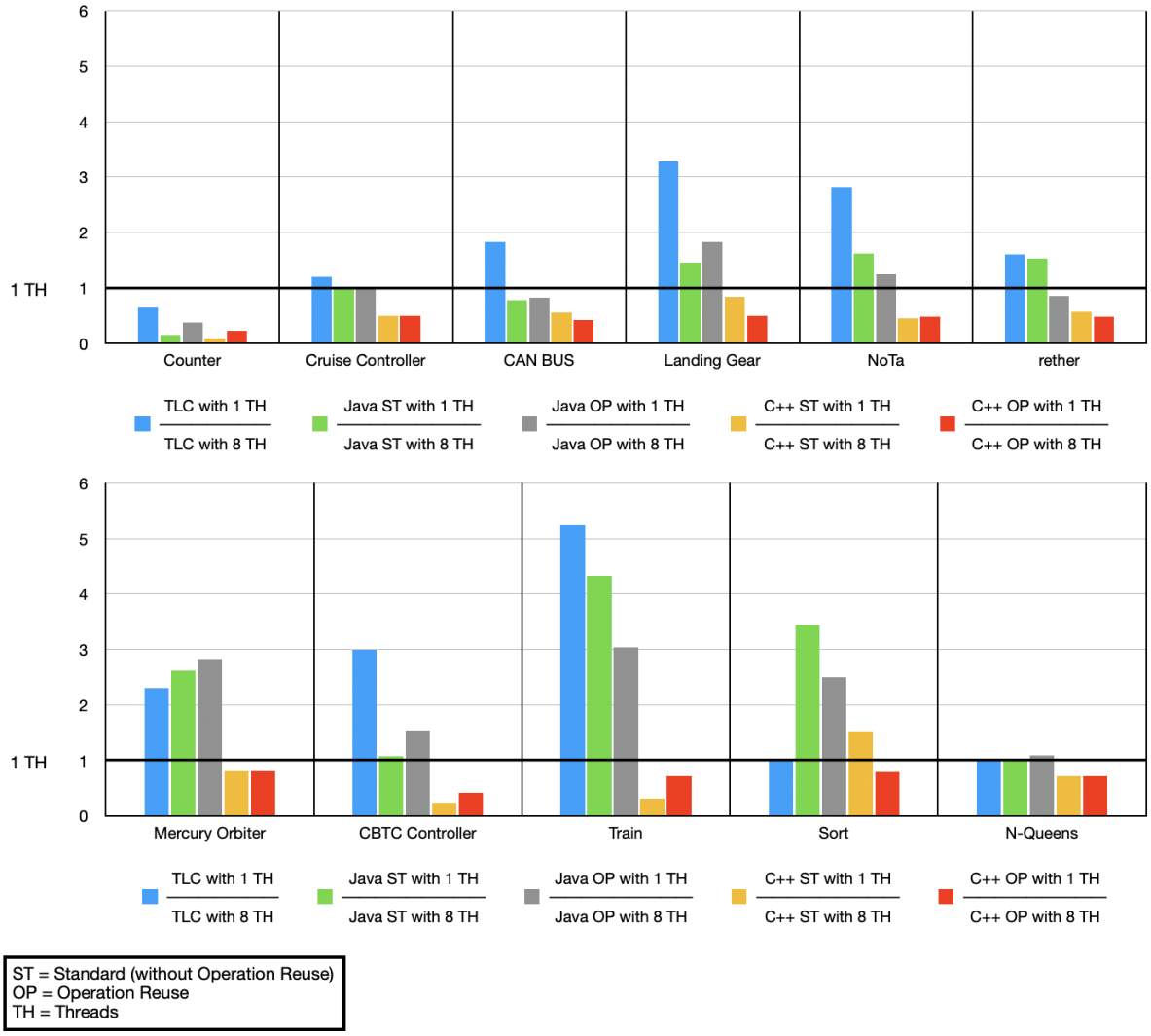


Figure 9.3.: Speedups of Multi-threaded Model Checking with 8 Threads Relative to Single-Threaded Model Checking as Bar Charts: `X` with 8 Threads vs. `X` with 1 Thread for all `X` ∈ {TLC, Java ST, Java OP, C++ ST, C++ OP}

While parallelization can improve the Java runtimes, the C++ runtimes are worse for all machines in both configurations (with and without operation reuse), except for Sort without operation reuse. These results differ from Chapter 7, where C++ with parallelization performed better. As mentioned earlier, we use a different computer to run the benchmarks in this chapter than in Chapter 7.

The detailed results are as follows:

- TLC with eight threads is faster than TLC with one thread by a factor of more than 2 for 5 machines. Parallelization achieves the highest speedup for Train with a factor greater than 5.

- Java ST is faster by a factor of more than 2 for 3 machines, comparing eight threads vs. one thread. Parallelization achieves the highest speedup for Train with a factor greater than 4.

- Java OP is also faster by a factor of more than 2 for 3 machines when comparing eight threads vs. one thread. Again, parallelization achieves the highest speedup for Train with a factor of around 3.

- Comparing TLC with Java ST, the respective speedups to 1 thread are higher for Java ST than for TLC for 2 machines, whereas TLC achieves higher speedups for 7 machines. Comparing TLC with Java OP, there are even 8 machines where Java OP achieves lower speedups relative to 1 thread. However, the speedups to 1 thread are higher for Java OP than for TLC for 2 machines.

- TLC achieves higher speedups through parallelization than C++ for all machines in both configurations, except Sort with C++ ST.

Operation Reuse. Figure 9.4 shows the speedups of operation reuse compared to the standard configuration (without operation reuse) for ProB and the generated code in Java, JavaScript, and C++. That means we compare `X` OP vs. `X` ST for all `X` $\in$ {ProB, Java 1 TH, Java 8 TH, JS (1 TH), C++ 1 TH, C++ 8 TH}, where ST means standard (without operation reuse), OP means operation reuse, and TH denotes the number of threads. A similar comparison for ProB, analyzing the efficiency of operation reuse was done by Leuschel [150]. Detailed results are presented in Table 9.3 and Table 9.5.

Figure 9.4 shows that operation reuse is significantly more efficient than the caching technique in Chapter 7. This statement applies to all evaluated languages in B2Program: Java, JavaScript, and C++. Section 9.3 describes the implementation of operation reuse [150] in B2Program and highlights the differences to caching in Chapter 7. Caching in Chapter 7 only reuses computations when reaching the current state through an operation that does not modify the variables read. In contrast, operation reuse [150] reuses the computed values from previously visited states that share the same projected state.

The results show that the speedup achieved with operation reuse for code generated by B2Program is less than that for ProB for most machines. Interpretation with ProB often leads to more overhead, which operation reuse can reduce.

Again, the generated code is already efficient for many machines where ProB ST struggles, which could explain why the speedup for the generated code is less than that for ProB for most machines.

In more detail:

- For two machines (Train and Sort), model checking with operation reuse achieves higher speedups for B2Program than for ProB. ProB OP outperforms the code generated by B2Program for both machines across all languages when operation reuse is deactivated for the generated code.

- Model checking with operation reuse achieves faster runtimes for B2Program for 7 out of 11 machines. While operation reuse leads to worse performance for 2 machines (Counter and CBTC Controller), the runtimes for the 2 other machines (Cruise Controller and N-Queens) remain unchanged.

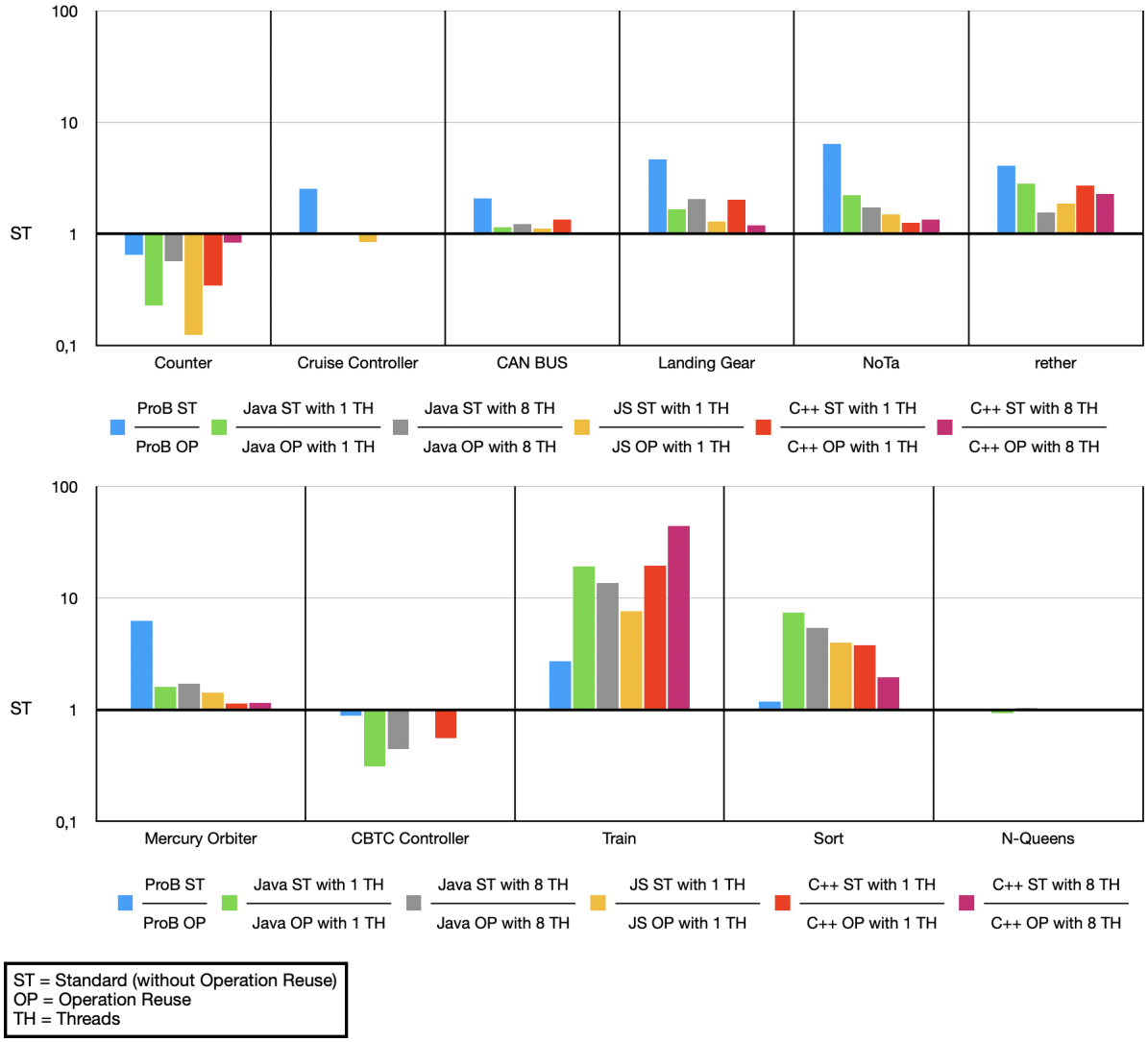


Figure 9.4.: Speedups of OP vs. ST for ProB and Generated Java, JavaScript (JS), and C++ Code as Bar Charts: `X` OP vs. `X` ST for all `X` ∈ {ProB, Java 1 TH, Java 8 TH, JS, C++ 1 TH, C++ 8 TH}

The results of memory consumption with operation reuse for the generated code are as follows: For some machines, memory consumption increases, whereas for others, it decreases. On the one hand, the caches require additional memory and therefore,

memory consumption increases. On the other hand, operation reuse enables reusing evaluated results for invariants, outgoing transitions, and the operations' effects rather than recalculating them. In B2Program, the implementations of B operations are immutable (i.e., they return new objects as results). Furthermore, B2Program's used libraries implement sets and relations with persistent data structures. As a result, the generated code can share data between multiple states and caches. Using cached results also avoids the memory consumption required for computation.

In more detail:

- Model checking of two machines (Counter and CBTC Controller) consumes more memory and leads to worse runtimes when activating operation reuse for all languages targeted by B2Program.
- Model checking with the generated code and operation reuse results in faster runtimes while consuming more memory for 2 machines (Train and Sort). The memory consumption for both machines also increases for ProB OP (compared to ProB ST).
- For 7 out of 11 machines, the memory consumption decreases for Java OP (compared to Java ST). For all these machines, the memory consumption remains the same for ProB.
- For JavaScript, the memory consumption decreases for 2 machines (NoTa and Mercury Orbiter) when operation reuse is activated. For both machines, the memory consumption remains the same for ProB.
- The memory consumption for C++ increases for four machines (Counter, CBTC Controller, Train, and Sort) when operation reuse is activated and stays the same for all other machines.

9.5. Detailed Results

This section shows detailed results for the benchmarks in this chapter.

Table 9.3 shows the single-threaded runtimes. The results are relevant to Figure 9.1, Figure 9.3, Figure 9.4 and Figure 9.6. Model checking CBTC Controller with JavaScript OP results in an out-of-memory error.

Table 9.3.: Model Checking Runtimes for ProB, TLC, and Generated Java, JavaScript (JS), and C++ Code in Seconds with Memory Consumption in KB, Speedups Relative to ProB OP, and Size of State Space (states and transitions). ST = Standard (without Operation Reuse), OP = Operation Reuse

Counter		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
(1 000 001 states,	Runtime	73.27	47.54	1.07	0.57	2.49	1.21	9.71	0.26	0.76
2 000 001 transitions)	Memory	1 112 772	1 112 770	593 584	353 536	2 186 304	444 832	2 385 344	209 424	482 528
	Speedup	1	1.54	68.48	128.54	29.43	60.55	7.55	281.81	96.41
Cruise Controller (Volvo,		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
1360 states,	Runtime	0.37	0.94	0.30	0.40	0.41	0.11	0.13	0.01	0.01
26 149 transitions)	Memory	171 802	171 247	174 624	120 864	107 616	86 000	91 456	3040	3280
	Speedup	1	0.39	1.23	0.93	0.90	3.36	2.85	37.00	37.00
CAN BUS (J.Colley,		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
132 599 states	Runtime	14.92	31.08	1.50	1.18	1.03	2.04	1.83	0.67	0.50
340 266 transitions)	Memory	351 768	350 597	419 328	418 784	366 800	192 080	181 472	163 792	163 520
	Speedup	1	0.48	9.95	12.64	14.49	7.31	8.15	22.27	29.84
Landing Gear [135]		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
(131 328 states,	Runtime	24.80	114.86	6.30	4.20	2.54	9.42	7.26	2.88	1.42
884 369 transitions)	Memory	475 445	468 602	503 536	897 808	455 328	203 952	210 880	148 432	151 152
	Speedup	1	0.22	3.94	5.90	9.76	2.63	3.42	8.61	17.46
NoTa [183]		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
(80 718 states,	Runtime	16.16	102.71	4.50	3.88	1.74	9.07	6.09	6.00	4.80
1 797 353 transitions)	Memory	925 126	924 446	591 072	977 216	443 840	325 632	166 720	150 960	147 984
	Speedup	1	0.16	3.59	4.16	9.29	1.78	2.65	2.69	3.37
rether [226]		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
(42 253 states,	Runtime	6.14	25.03	0.95	2.13	0.76	2.99	1.60	1.62	0.60
381 074 transitions)	Memory	304 383	295 550	338 368	724 832	288 208	154 080	151 568	39 936	41 680
	Speedup	1	0.25	6.46	2.88	8.08	2.05	3.84	3.79	10.23
Mercury Orbiter [115]		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
(245 026 states,	Runtime	54.84	342.39	6.04	8.65	5.43	21.87	15.29	10.12	8.90
2 188 892 transitions)	Memory	937 581	932 124	772 864	1 228 656	682 736	432 128	244 096	239 424	234 928
	Speedup	1	0.16	9.08	6.34	10.10	2.51	3.59	5.42	6.16
CBTC Controller [168]		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
(1 636 546 states,	Runtime	1095.26	967.60	20.33	7.23	23.26	30.52	-	11.74	21.01
7 134 235 transitions)	Memory	3 499 720	2 886 149	659 568	2 010 352	4 532 384	1 046 960	-	4 793 184	7 696 048
	Speedup	1	1.13	53.87	151.49	47.09	35.89	-	93.29	52.13
Train [151, 5] (ten routes,		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
672 174 states,	Runtime	390.92	1061.57	195.56	796.08	41.35	1127.59	148.38	701.26	36.07
2 244 486 transitions)	Memory	2 822 864	1 159 907	1 482 464	947 504	2 701 392	1 186 544	2 948 848	1 032 576	1 355 104
	Speedup	1	0.37	2.00	0.49	9.45	0.35	2.63	0.56	10.84
sort_1000 [200]		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
(500 501 states,	Runtime	187.65	221.12	1029.39	396.81	53.14	607.45	152.76	515.40	135.55
500 502 transitions)	Memory	916 152	600 590	363 584	795 824	1 796 976	268 384	1 940 224	325 744	620 176
	Speedup	1	0.85	0.18	0.47	3.53	0.31	1.23	0.36	1.38
N-Queens with N=4		ProB OP	ProB ST	TLC	Java ST	Java OP	JS ST	JS OP	C++ ST	C++ OP
(4 states,	Runtime	0.003	0.003	0.14	68.72	74.25	14.31	14.15	9.40	9.46
6 transitions)	Memory	162 709	162 708	174 128	582 864	492 000	296 672	301 312	45 600	45 264
	Speedup	1	1	0.02	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01

Figure 9.5 shows the speedups of multi-threaded model checking for Java and C++ with 6 threads relative to TLC with 6 threads (analogously to Figure 9.2 in Section 9.4).

Figure 9.6 shows the speedups of 6 threads relative to 1 thread (analogously to Figure 9.3 in Section 9.4).

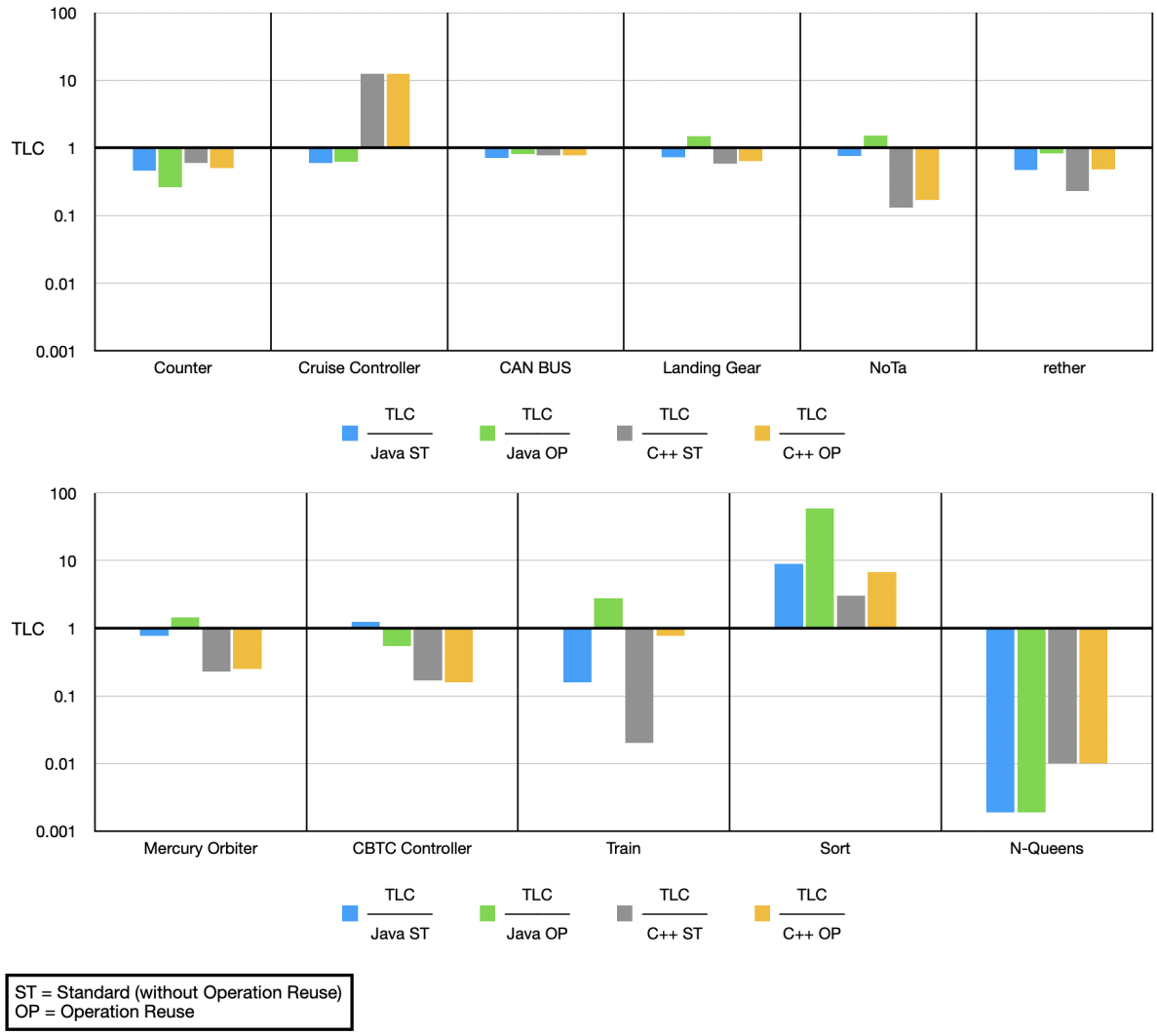


Figure 9.5.: Speedups of Multi-Threaded Model Checking for Java and C++ with 6 Threads Relative to TLC with 6 Threads as Bar Charts

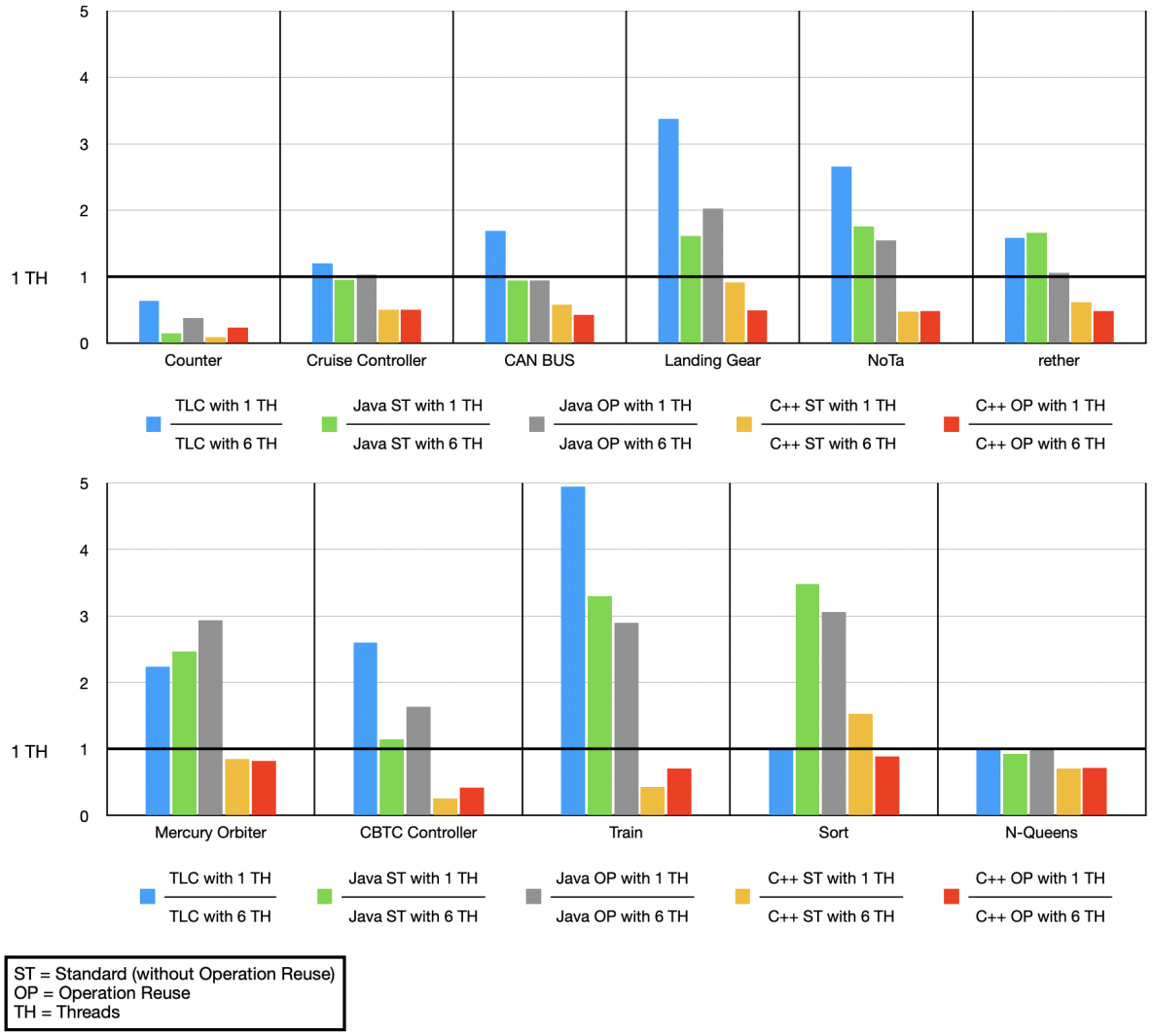


Figure 9.6.: Speedups of Multi-threaded Model Checking with 6 Threads Relative to Single-Threaded Model Checking as Bar Charts: `X` with 6 Threads vs. `X` with 1 Thread for all `X` ∈ {TLC, Java ST, Java OP, C++ ST, C++ OP}

For completeness, we also present the multi-threaded runtimes with 6 threads (see Table 9.4), as Chapter 7 contains benchmarks run with 6 threads, but on a different computer than in Section 9.4. The data is used to create Figure 9.5 and Figure 9.6, but not for any figures in Section 9.4.

Table 9.5 shows the multi-threaded runtimes with 8 threads. The results are relevant to Figure 9.2, Figure 9.3, and Figure 9.4.

Table 9.4.: Multi-threaded Model Checking Runtimes with 6 Threads for TLC, and Generated Java and C++ Code in Seconds with Memory Consumption in KB, Speedups Relative to TLC and respective configuration with 1 Thread, and Size of State Space (states and transitions). ST = Standard (without Operation Reuse), OP = Operation Reuse, TH = Threads

Counter		TLC	Java ST	Java OP	C++ ST	C++ OP
(1 000 001 states,	Runtime	1.68	3.68	6.47	2.79	3.33
2 000 001 transitions)	Memory	613 440	379 632	1 561 184	210 736	487 312
	Speedup to TLC	1	0.46	0.26	0.60	0.50
	Speedup to 1 TH	0.64	0.15	0.38	0.09	0.23
Cruise Controller (Volvo,		TLC	Java ST	Java OP	C++ ST	C++ OP
1360 states,	Runtime	0.25	0.42	0.40	0.02	0.02
26 149 transitions)	Memory	174 560	112 896	107 392	4352	4688
	Speedup to TLC	1	0.60	0.63	12.50	12.50
	Speedup to 1 TH	1.20	0.95	1.03	0.50	0.50
CAN BUS (J.Colley,		TLC	Java ST	Java OP	C++ ST	C++ OP
132 599 states	Runtime	0.89	1.25	1.10	1.15	1.15
340 266 transitions)	Memory	570 208	486 000	362 160	168 048	168 544
	Speedup to TLC	1	0.71	0.81	0.77	0.77
	Speedup to 1 TH	1.69	0.94	0.94	0.58	0.43
Landing Gear [135]		TLC	Java ST	Java OP	C++ ST	C++ OP
(131 328 states,	Runtime	1.87	2.61	1.26	3.18	2.92
884 369 transitions)	Memory	1 110 080	1 308 400	661 472	150 528	153 664
	Speedup to TLC	1	0.72	1.48	0.59	0.64
	Speedup to 1 TH	3.37	1.61	2.02	0.91	0.49
NoTa [183]		TLC	Java ST	Java OP	C++ ST	C++ OP
(80 718 states,	Runtime	1.69	2.21	1.12	12.74	9.96
1 797 353 transitions)	Memory	1 089 808	1 684 384	485 568	165 760	166 144
	Speedup to TLC	1	0.76	1.51	0.13	0.17
	Speedup to 1 TH	2.66	1.76	1.55	0.47	0.48
rether [226]		TLC	Java ST	Java OP	C++ ST	C++ OP
(42 253 states,	Runtime	0.60	1.28	0.72	2.60	1.24
381 074 transitions)	Memory	405 648	1 280 032	283 984	45 792	49 040
	Speedup to TLC	1	0.47	0.83	0.23	0.48
	Speedup to 1 TH	1.58	1.66	1.06	0.62	0.48
Mercury Orbiter [115]		TLC	Java ST	Java OP	C++ ST	C++ OP
(245 026 states,	Runtime	2.70	3.50	1.85	11.97	10.79
2 188 892 transitions)	Memory	1 152 320	2 273 904	1 369 872	286 912	267 744
	Speedup to TLC	1	0.77	1.46	0.23	0.25
	Speedup to 1 TH	2.24	2.47	2.94	0.85	0.82
CBTC Controller [168]		TLC	Java ST	Java OP	C++ ST	C++ OP
(1 636 546 states,	Runtime	7.81	6.31	14.31	45.31	50.13
7 134 235 transitions)	Memory	847 792	1 466 016	4 476 144	4 897 968	7 764 464
	Speedup to TLC	1	1.24	0.55	0.17	0.16
	Speedup to 1 TH	2.60	1.15	1.63	0.26	0.42
Train [151, 5] (ten routes,		TLC	Java ST	Java OP	C++ ST	C++ OP
672 174 states,	Runtime	39.53	241.39	14.26	1615.74	51.03
2 244 486 transitions)	Memory	1 601 984	1 319 216	3 641 888	1 054 944	1 421 776
	Speedup to TLC	1	0.16	2.77	0.02	0.77
	Speedup to 1 TH	4.95	3.30	2.90	0.43	0.71
sort_1000 [200]		TLC	Java ST	Java OP	C++ ST	C++ OP
(500 501 states,	Runtime	1031.70	114.03	17.36	337.70	152.49
500 502 transitions)	Memory	374 592	1 467 632	4 056 720	326 576	615 424
	Speedup to TLC	1	9.05	59.43	3.06	6.77
	Speedup to 1 TH	1.00	3.48	3.06	1.53	0.89
N-Queens with N=4		TLC	Java ST	Java OP	C++ ST	C++ OP
(4 states,	Runtime	0.14	73.86	74.22	13.20	13.14
6 transitions)	Memory	174 144	496 992	494 912	47 472	50 400
	Speedup to TLC	1	< 0.01	< 0.01	0.01	0.01
	Speedup to 1 TH	1.00	0.93	1.00	0.71	0.72

Table 9.5.: Multi-threaded Model Checking Runtimes with 8 Threads for TLC, and Generated Java and C++ Code in Seconds with Memory Consumption in KB, Speedups Relative to TLC and respective configuration with 1 Thread, and Size of State Space (states and transitions). ST = Standard (without Operation Reuse), OP = Operation Reuse, TH = Threads

Counter		TLC	Java ST	Java OP	C++ ST	C++ OP
(1 000 001 states,	Runtime	1.67	3.72	6.54	2.77	3.32
2 000 001 transitions)	Memory	646 080	374 704	1 554 112	210 960	494 848
	Speedup to TLC	1	0.45	0.26	0.60	0.50
	Speedup to 1 TH	0.64	0.15	0.38	0.09	0.23
Cruise Controller (Volvo,		TLC	Java ST	Java OP	C++ ST	C++ OP
1360 states,	Runtime	0.25	0.41	0.41	0.02	0.02
26 149 transitions)	Memory	174 560	112 688	110 592	4624	4976
	Speedup to TLC	1	0.61	0.61	12.50	12.50
	Speedup to 1 TH	1.20	0.98	1.00	0.50	0.50
CAN BUS (J.Colley,		TLC	Java ST	Java OP	C++ ST	C++ OP
132 599 states	Runtime	0.82	1.52	1.24	1.22	1.19
340 266 transitions)	Memory	543 072	512 608	337 392	167 488	168 816
	Speedup to TLC	1	0.54	0.66	0.67	0.69
	Speedup to 1 TH	1.83	0.78	0.83	0.55	0.42
Landing Gear [135]		TLC	Java ST	Java OP	C++ ST	C++ OP
(131 328 states,	Runtime	1.92	2.87	1.39	3.43	2.88
884 369 transitions)	Memory	1 137 536	1 153 264	659 600	150 768	153 920
	Speedup to TLC	1	0.67	1.38	0.56	0.67
	Speedup to 1 TH	3.28	1.46	1.83	0.84	0.49
NoTa [183]		TLC	Java ST	Java OP	C++ ST	C++ OP
(80 718 states,	Runtime	1.60	2.39	1.39	13.46	10.04
1 797 353 transitions)	Memory	1 180 752	1 604 608	472 128	164 896	166 288
	Speedup to TLC	1	0.67	1.15	0.12	0.16
	Speedup to 1 TH	2.81	1.62	1.25	0.45	0.48
rether [226]		TLC	Java ST	Java OP	C++ ST	C++ OP
(42 253 states,	Runtime	0.59	1.39	0.89	2.86	1.25
381 074 transitions)	Memory	405 296	1 133 040	284 576	43 904	48 768
	Speedup to TLC	1	0.42	0.66	0.21	0.47
	Speedup to 1 TH	1.61	1.53	0.85	0.57	0.48
Mercury Orbiter [115]		TLC	Java ST	Java OP	C++ ST	C++ OP
(245 026 states,	Runtime	2.62	3.30	1.92	12.50	10.93
2 188 892 transitions)	Memory	1 181 984	2 359 856	1 360 128	276 128	266 912
	Speedup to TLC	1	0.79	1.36	0.21	0.24
	Speedup to 1 TH	2.31	2.62	2.83	0.81	0.81
CBTC Controller [168]		TLC	Java ST	Java OP	C++ ST	C++ OP
(1 636 546 states,	Runtime	6.80	6.73	15.06	50.66	51.81
7 134 235 transitions)	Memory	967 424	1 413 296	4 498 384	4 875 280	7 776 752
	Speedup to TLC	1	1.01	0.45	0.13	0.13
	Speedup to 1 TH	2.99	1.07	1.54	0.23	0.41
Train [151, 5] (ten routes,		TLC	Java ST	Java OP	C++ ST	C++ OP
672 174 states,	Runtime	37.31	184.31	13.60	2268.59	50.98
2 244 486 transitions)	Memory	1 600 160	1 344 064	3 766 224	1 058 656	1 427 328
	Speedup to TLC	1	0.20	2.74	0.02	0.73
	Speedup to 1 TH	5.24	4.32	3.04	0.31	0.71
sort_1000 [200]		TLC	Java ST	Java OP	C++ ST	C++ OP
(500 501 states,	Runtime	1038.06	115.50	21.25	337.00	172.42
500 502 transitions)	Memory	370 864	1 455 296	3 851 776	326 752	625 168
	Speedup to TLC	1	8.99	48.85	3.08	6.02
	Speedup to 1 TH	0.99	3.44	2.50	1.53	0.79
N-Queens with N=4		TLC	Java ST	Java OP	C++ ST	C++ OP
(4 states,	Runtime	0.14	70.45	68.13	13.24	13.08
6 transitions)	Memory	174 080	498 768	590 080	46 416	49 232
	Speedup to TLC	1	< 0.01	< 0.01	0.01	0.01
	Speedup to 1 TH	1.00	0.98	1.09	0.71	0.72

Part IV.

Conclusions

10. Conclusions and Future Work

In this chapter, we recap the research questions asked in Chapter 1 and provide answers based on the research in this thesis.

10.1. Validation of Formal Models by Timed Probabilistic Simulation

Chapter 2 introduced a concept which allows formal models to be simulated with timing and probabilistic behavior. We implemented this concept in the *timed probabilistic simulation* technique in the SimB simulator. SimB also contains features for statistical validation. The research questions concerning Chapter 2 are:

- **Q1**: How can we annotate events in formal models with timing and probabilistic elements for simulation?
- **Q2**: When is it beneficial to use timed probabilistic simulation, and how does this technique help modelers validate formal models?

Answer Q1. The concept of *timed probabilistic simulation* relies on *activations* to annotate operations/events with timing and probabilistic behavior. SimB's activations are of two kinds: *direct activations* to execute an event after a specific time/delay and trigger other activations, and *probabilistic choices* to select between activations probabilistically. Direct activations might also probabilistically select values for non-deterministically assigned variables (and parameters) in the event. All activations together describe an activation diagram with time and probabilities, describing how events trigger one another in the simulation.

SimB's *activation diagrams* work independently of whether time is part of the formal model. If time is not part of the formal model, one can add timing behavior to the activation diagrams. Otherwise, SimB's activation diagrams can adapt to the timing behavior in the formal model, e.g., when time is modeled as described in [198, 46, 139]. In particular, the time in the *direct activations* can be static values or computed from the constants and variables in the formal model.

Answer Q2. *Timed probabilistic simulation* complements existing techniques such as animation and model checking and offers techniques to enhance the validation of formal models.

First, we can annotate events in an existing model with timing and probabilistic behavior to run an automatic simulation. With Real-Time simulation, one can simulate a single scenario in real time while observing the formulated behavior. Compared to animation, a simulation run is executed automatically, i.e., without user interaction, and in real time. Furthermore, one can replay the resulting trace with Real-Time simulation or validate a specific behavior through trace replay.

Second, timed probabilistic simulation supports Monte Carlo simulation along with statistical validation techniques. In particular, one can execute multiple simulation runs using Monte Carlo simulation and afterward apply hypothesis testing or estimate the likelihood of values. These techniques are helpful to validate probabilistic properties. Since SimB activation diagrams contain timing aspects, one can also formulate and check timing properties.

Third, the activation diagrams enable precise encoding of scenarios for simulation, targeting specific parts of the state space. One can then run Monte Carlo simulation to detect violations of desired properties. In particular, Monte Carlo simulation can find violations when model checking struggles due to state space explosion. Timed probabilistic simulation has been used for validating case studies where a complete model check is not feasible, such as the highway AI (see Chapter 5) and railML topologies [94]. Furthermore, Monte Carlo simulation is beneficial even when the state space is fully covered. For instance, Monte Carlo simulation can provide probability estimates for certain behaviors, which is not possible with explicit-state model checking alone. Note that the guarantees for Monte Carlo simulation are different from those for model checking. Applying Monte Carlo simulation, one can achieve a statistical guarantee with a confidence interval rather than a full guarantee, as with model checking.

Timed probabilistic simulation can be used for interactive systems and AI systems, as presented in Chapter 3 – Chapter 6 (see also **Q3** – **Q8** for the results).

Conclusion. Timed probabilistic simulation can be used to simulate formal models (1) as a Real-Time simulation with timing and probabilistic behavior or (2) as a Monte Carlo simulation where one can apply statistical techniques to validate resulting simulation runs. SimB performs Monte Carlo simulation in accelerated time, i.e., SimB does not wait for the delays to expire in real time but instead skips the delay to the next step during simulation.

Timed probabilistic simulation complements existing formal methods techniques such as animation and model checking. As SimB's concept shares similarities with CSP [107], Timed CSP [76, 67], Petri Nets [191], Timed Petri Nets [249], and Probabilistic Time Petri Nets [70], it could be interesting to compare these formalisms to SimB's concepts in the future. One could evaluate whether/how SimB adapts to these formalisms and vice versa. Furthermore, one could present a formal semantics, e.g., an operational semantics for SimB activation diagrams in the future.

The challenge remains to formulate an activation diagram that corresponds to realistic behavior, i.e., there may be a gap between the simulation and reality. This problem also affects AI applications (see Chapter 5 and Chapter 6, results discussed in Section 10.4).

There, we cater external simulations to SimB's activation diagram to simulate real AI in SimB.

10.2. Validation of Formal Models by Interactive Simulation

In Chapter 3, we implemented interactive elements in the SimB simulator to create more realistic prototypes. We refer to this technique as *interactive simulation*, which is an extension of timed probabilistic simulation in Chapter 2. The idea of *interactive simulation* is to make simulations respond to user interactions. As described in Section 1.8.2, a concrete example in the aviation domain is the landing gear system [135], where the landing gear extends/retracts in response to the pilot's input on the handle within a specific time. Another example is a vehicle's exterior light system [154], which reacts to the driver's actions on the pitman controller and the warning lights button. In particular, the corresponding vehicle's lights flash every 500 ms in real time.

The research questions concerning Chapter 3 are:

- **Q3**: How can we simulate system reactions in response to user interactions?
- **Q4**: When should we use *interactive simulation*, and how can we validate user interactions and system reactions?

Answer Q3. To integrate user interactions into SimB's activation concept, we introduce *listeners* to handle user interactions. A listener detects an event/operation performed by a user, along with a predicate that must be satisfied. Furthermore, a listener links to a SimB activation, which responds to the user interaction. That means that the user interaction detected by the listener triggers the SimB activation. With interactive simulation, one can thus manually perform a user interaction and observe the system's reactions in real time. On the tooling side, a user performs actions in an animator (e.g. ProB's animator) or a domain-specific visualization (e.g. VisB), while a simulator (e.g. SimB) performs the system's reactions.

Answer Q4. Interactive simulation combines animation (because a user manually performs interactions) and simulation (because the simulator automatically performs system reactions). Therefore, one can use interactive simulation to create real-time prototypes for systems where system reactions are triggered based on user interactions. Such systems include interactive systems such as human-machine interfaces. In particular, interactive simulation is recommended when a modeler, stakeholder, or domain expert wants to manually perform user interactions and observe the system's responses in real time.

Interactive simulation is applicable with other validation techniques. For instance, one can save scenarios created through interactive simulation as traces and later use them for testing. We also presented a technique to create a state space projection [136]

onto the appearances of visual components to reason about the interplay between user interactions and system reactions.

Conclusion. *Interactive simulation* combines animation with simulation to create real-time prototypes. Our tooling implements the concept using a simulator (SimB) and a visualization tool (VisB). This combination enables users to manually perform actions through a graphical user interface while a simulator automatically simulates system responses.

As outlined in Section 10.1, a possible future work is to present a formal semantics, e.g. an operational semantics, for SimB activation diagrams and compare them with other formalisms that incorporate timing and probabilistic behavior. Interactive simulation introduced interactive components into SimB activation diagrams, for which future work could present a formal semantics as well. Consequently, one could explore verifying user interactions and system reactions more rigorously in the future.

10.3. Development and Validation of a Formal Model and Prototype for an Air Traffic Control System

In Chapter 4, we presented a formal model of an air traffic control system, namely the Arrival Manager (AMAN). The formal model features events that an air traffic controller (ATCo) and AMAN's automatic components can perform. The research question concerning Chapter 4 is:

- **Q5**: How can we convert a formal model into a prototype for a real-time human-machine interface?

More Details on Q3. In the concept of *interactive simulation*, we linked user interactions to simulations of system responses (see discussion in Section 10.2). The AMAN case study shows another way a system can react to user interactions. In particular, user interactions modify the system's current state in the formal model, which AMAN's automatic component considers when performing its events. Technically, the simulation of the automatic component runs without being triggered explicitly by a user interaction.

Answer Q5. Human-machine interfaces are interfaces through which a *human* can interact with a *machine*. One can implement interactive parts of the prototype with a domain-specific visualization that serves as the user interface for the underlying formal model. This idea also aligns with prototyping approaches in PVSio-Web [241] and *formal* MVC [31]. One can implement automated components, representing the machine part, using a simulation that automatically executes events in real time.

A relevant aspect is how user interactions and automatic events influence each other. One can simulate automatic responses to user interactions in real time using interactive simulation (see Chapter 3). The AMAN case study in Chapter 4 highlights that the

design of a responsive system does not necessarily need a link between user interaction and system reaction. Here, automatic events occur every 10 seconds, assuming that the human is not performing an action at that moment. In this case study, the formal model coordinates interactive and automatic events in two ways: First, the formal model contains guards which block automatic events from being performed by the simulation (if a human is performing an action at that moment). Second, both automatic and user events perform actions which modify the system's current state by the formal model's variables. Thus, humans and machines respond to each other's actions through the system represented by the formal model.

Conclusion. Domain-specific visualization and real-time simulation are well-suited to represent the *human* and *machine* parts in prototypes of human-machine interfaces, respectively. As demonstrated in Chapter 4, a prototype can be beneficial to validate many requirements. Some requirements even relate specifically to the interface between humans and machines, for which a prototype is necessary for validation. In the future, one could evaluate more case studies with prototyping human-machine interfaces.

10.4. Validation of Reinforcement Learning Agents and Safety Shields with ProB

In Chapter 5, we trained reinforcement learning (RL) agents for a highway environment and validated them using SimB's statistical validation techniques. The challenge is to simulate real AI behavior with SimB. With the approach we implemented, we could also employ safety shields as runtime monitors for the highway AI. We achieved better results in Chapter 6 (additional chapter) compared to Chapter 5 by improving the training parameters and employing the Responsibility-Sensitive Safety (RSS) [209] technique. The research questions concerning Chapter 5 and Chapter 6 are:

- **Q6**: How suitable are SimB's simulation and validation capabilities to check the safety and evaluate the quality of AI systems?
- **Q7**: How can we simulate real AI with SimB?
- **Q8**: How can we use the formal model as a runtime monitor, i.e., a safety shield for AI systems?

Answer Q6. The challenge was to make SimB cater to real AI simulation (see **Q7**) so that SimB is feasible for validating real AI behavior.

Before Chapter 5, SimB's validation techniques include hypothesis testing and the estimation of the likelihood of fulfilling (or violating) properties. These techniques enable us to evaluate the safety of the AI. As outlined in **Q2**, one can achieve statistical guarantees with a confidence interval. Additionally, we extended SimB to estimate values over expressions.

As a result, SimB is suitable for simulating and validating AI systems, including safety aspects. Below, we recap the successful applications of SimB to AI systems.

In Chapter 5 and Chapter 6, we validated various RL agents and demonstrated the efficiency of safety shields. SimB helped us identify weaknesses in the RL agents' reward functions and safety shields, allowing us to improve them afterward. Chapter 6 demonstrated that an established technique like RSS improves safety for highway agents and even blocks (most) adversarial behavior.

SimB also works for simulating and validating AI systems beyond reinforcement learning. For instance, Gruteser et al. [95] use the same feature in SimB to simulate and validate an AI-based train system with image recognition and certified control. Gruteser et al. [95] detected weaknesses in the image recognition AI, certified control, and the safety shield.

Answer Q7. In Chapter 5 and Chapter 6, we simulate real AI via SimB by connecting the AI to SimB. Technically, the AI directly controls the activations via SimB. While Chapter 5 and Chapter 6 focus on RL agents, this technique is also suitable for other AI systems. For instance, Chapter 5 allows us to simulate real AI for a train system with image recognition and certified control [95]. However, the current tooling cannot create images based on the current state of the environment in the formal model. Instead, Gruteser et al. [95] sampled from previously taken images that resembled the current state.

Answer Q8. While simulating real AI with SimB, we can use the formal model as a runtime monitor, particularly as a safety shield, to enforce safety. The safety shield contains *safety constraints* in the guards of the actions, specifying which actions are considered safe given the current observation/state. Consequently, the AI can only perform one of those actions. In reinforcement learning applications (Chapter 5 and Chapter 6), the AI performs the action deemed safe with the highest reward.

According to Sha [208], the formal model acts as the simple system that enforces safety rules on a complex system, i.e., the AI. The approach described in Chapter 5 and Chapter 6 is comparable to *pre-shielding* [125].

Conclusion. In our approach, we simulate real AI with SimB while using a formal model as a runtime monitor, precisely, a safety shield. This approach enables the validation of simulation runs using techniques like statistical validation or trace replay. Those techniques help improve the safety shields and reward functions (see Chapter 5 and Chapter 6).

Safety shielding with a formal model ensures that the AI only performs actions deemed safe, whereas the AI alone would act unsafely. There are other approaches that aim to verify the neural network of an AI [121, 201]. Robustness checking [86, 92] verifies that a neural network is stable under perturbations. Certified control [118] is another runtime monitoring technique for image recognition. All these techniques aim to ensure the correctness of the AI at different levels and complement each other. Thus, a formal model

can be used as a safety shield in a multi-layered monitoring and verification approach along with other techniques, as illustrated in Figure 1 of [93] (also see results presented by Gruteser et al. [95]).

The idea of using a formal model for safety enforcement at runtime is also implemented for ASMs in the Asmeta toolset by Bonfanti et al. [37]. As future work, Bonfanti et al. consider applying their safety enforcement approach to AI systems. The results in this thesis could thus provide relevant insights for the Asmeta toolset towards application to AI systems.

The technique in Chapter 5 also works for AI systems beyond reinforcement learning. In particular, Gruteser et al. [95] use the features from Chapter 5 to simulate and validate an AI-based train system with image recognition and certified control, identifying weaknesses in the AI, certified control, and safety shield. Towards the future, one could explore safety shields and SimB's validation techniques for other AI applications or cyber-physical systems.

Instead of connecting real AI with SimB, one could extract a SimB activation diagram from the neural network for simulation. That approach would abstract away the AI and could allow other verification techniques, especially if future work presents a formal semantics for SimB activation diagrams. For instance, Păsăreanu et al. [189] extract Discrete Time Markov Chains [196] from the confusion matrices computed for the neural network to verify the AI using the probabilistic model checker PRISM [131].

10.5. Model Checking B Models via High-Level Code Generation

In Chapter 7, we extended B2Program to generate model checking code. The goal was to achieve efficient performance with code generation. This thesis focuses on the target languages Java, JavaScript/TypeScript, and C++. We compared the results with ProB [153] and TLC [246] (B machines translated via TLC4B [100] to TLA+). Some constructs in Classical B are challenging to support, resulting in some limitations. In Chapter 9, we resolved some limitations on Classical B supported by B2Program and further improved the performance. The research questions concerning Chapter 7 and Chapter 9 are:

- **Q9**: How can we generate model checking code with B2Program from a Classical B model, targeting imperative programming languages?
- **Q10**: What high-level constructs does B2Program support, and what are the limitations?
- **Q11**: How does the generated model checking code by B2Program perform compared to state-of-the-art tools such as ProB or TLC, and which techniques improve the performance?

Answer Q9. The generated model checking code implements an explicit-state model checking algorithm, which checks invariants and deadlock-freedom. First, the algorithm initializes the formal model, i.e., it starts in the initial state. The algorithm then traverses the complete state space by exploring the succeeding states of each visited state, starting from the initial state. In each visited state, the algorithm computes outgoing transitions and checks invariants and deadlock-freedom.

First, B2Program generates (1) a constructor to initialize the formal model and (2) functions to execute an operation with given parameter values [233].

Second, B2Program generates functions to compute all outgoing transitions for the current state. For operations without parameters, B2Program generates a boolean function from the guard, computing the enabledness of the operation. In this case, the algorithm executes the operation only if the guard evaluates to true. B2Program treats parameters $p_1, \ldots, p_n$ in an operation's guard P similar to the *set comprehension* $\{p_1, \ldots, p_n \mid P\}$. The generated function computes parameter values for which the guard is true, i.e., the operation is enabled. The algorithm then executes the operation for all these parameter values. Generated functions to compute outgoing transitions are also used to check for deadlock-freedom. In particular, a deadlock occurs if no operation has a guard that evaluates to true.

Third, B2Program generates *boolean functions* for each invariant conjunct to check the invariant in the current state.

Answer Q10. B2Program supports various high-level constructs, including set operations, relation operations, set comprehensions, lambdas, quantified predicates with universal and existential quantifiers, and, in some cases, even infinite sets and non-deterministic substitutions.

Quantified constructs must contain sub-predicates (conjuncts) to constrain the free variables. Those sub-predicates are $x = E$ for assigning a free variable x to the value of an expression E, or $x \in E$, $x \subset E$, $x \subseteq E$ for iterating over a finite set computed from an expression E to constrain x. Consequently, B2Program translates those constraining predicates ($x \in E$, $x \subset E$, $x \subseteq E$) to *for-loops* in imperative programming languages. This feature is relevant for set comprehensions and, therefore, also for computing outgoing transitions.

B2Program tries to rewrite expressions on the right-hand side of $\in$, $\notin$, $\subseteq$, and $\not\subseteq$ to avoid explicit computation of infinite (or large) sets. Therefore, infinite sets are only allowed on the right-hand side of those predicates if rewritten to such a form. For instance, B2Program does not allow storing an infinite set in a variable.

B2Program supports non-deterministic substitutions for simulation by selecting one possible execution branch. For model checking, B2Program only supports top-level `SELECT`, `PRE` and `ANY` as non-deterministic substitutions in an operation. Computing all outgoing transitions is more challenging for other non-deterministic substitutions, which could be future work.

Furthermore, B2Program uses external libraries that allocate memory dynamically to support high-level constructs. Thus, B2Program must not target embedded systems.

Answer Q11. Compared to ProB, the model checking code generated by B2Program performs better for most benchmarks. For some models, ProB performs better than B2Program due to its constraint-solving capabilities. TLC and the code generated by B2Program both perform similarly well. There are models where B2Program performs better than TLC and vice versa.

Operation reuse [150] improves the model checking runtimes for many models translated by B2Program to the respective target languages. Similar to operation reuse in ProB [150], the technique caches the operations' effects, the computation of outgoing transitions, and the evaluation of invariants. The speedup achieved with operation reuse for code generated by B2Program is slightly less than for ProB for most models. For code generated with B2Program, the memory consumption of some models increases with operation reuse, whereas the opposite is true for others.

Parallelization also improves the runtimes for many models translated to Java by B2Program. However, for C++, the runtimes are improved by parallelization on the x86 architecture for many models, but not on the ARM architecture. Comparing B2Program with TLC, it seems that TLC's parallelization is more efficient.

Conclusion. In this thesis, we extend B2Program to generate model checking code from high-level B models, achieving efficient performance for many high-level models. The SPIN [109] model checker also follows the approach to generate model checking code, resulting in a strong performance. The main difference from B2Program is that SPIN translates low-level Promela models to C.

The performance of model checking is also relevant for animation (see answer to **Q13**). Animators also evaluate invariants, compute outgoing transitions, and execute operations.

To validate the correctness of the generated code, B2Program follows an approach that tests the generated code against existing tools such as ProB. By supporting multiple languages, one could safeguard each backend against another [233]; however, the backends share the same frontend and are thus not implemented independently. Furthermore, we used tests (initially created to validate ProB) to check laws for arithmetic, logical, comparison, set, and relation operations. An alternative approach to ensure correctness is to verify the code generator. The verified compiler CompCert [146] follows such an approach, which works on a small subset of C, targeting embedded systems. Complete verification, as achieved with CompCert, is more challenging – if not impossible – to accomplish with B2Program since the generated code relies on external libraries for high-level constructs.

10.6. Generating Interactive Documents for Domain-Specific Validation of Formal Models

In Chapter 8, we extend B2Program to generate JavaScript code and an HTML document that supports domain-specific visualization, animation, trace replay, and simulation. The goal is to involve domain experts in the validation process. Chapter 8 shares the same

limitations as Chapter 7. Furthermore, the performance of B2Program is also relevant for animation and trace replay. As mentioned earlier, we resolved some limitations and improved the performance for B2Program in Chapter 9. The research questions concerning Chapter 8 (results also influenced by Chapter 9) are:

- **Q12**: How can we generate code for validation, and how is it beneficial for validating formal models?
- **Q13**: How does the code generated by B2Program perform for "classical simulation" and animation?

More details on Q10. The limitations portrayed in Section 10.5 also apply to Chapter 8. As model checking does not (yet) have precise control over some non-deterministic constructs, we cannot animate and trace replay operations with these constructs precisely.

Answer Q12. Code generation for validation should be applied early in the development process, ideally with support for domain experts to provide feedback.

According to the V model of the software development process [78], our approach aims to apply validation activities early in the development phase, rather than only after implementation, where errors often require more effort to fix. Therefore, the code generator must be able to handle high-level, i.e., more abstract formal models, to achieve this goal. High-level code generation is the approach followed by B2Program.

Furthermore, the code generator shall also support techniques that are easier for domain experts to understand. To this end, domain experts should be able to animate and simulate scenarios, inspect them in a domain-specific visualization, and provide feedback in natural language. Modelers and domain experts can then work together, sharing animated (or simulated) traces.

This approach differs from state-of-the-art approaches in code generation for safety-critical software, e.g., as followed by AtelierB code generators [51]. In particular, state-of-the-art code generators are applied to the final refinement at the end of the development process.

Answer Q13. Model checking explores the complete state space by computing all transitions in each state, while animation computes all outgoing transitions for a visited state. Since an animation step behaves similarly to one model checking step, the performance of model checking also reflects the performance of animation. Comparing the code generated by B2Program with ProB, the speedups are higher for "classical simulation" than for model checking (animation).

Although JavaScript is an interpreted language, the runtimes of the generated JavaScript code from B2Program are in the same order of magnitude as the runtimes of the generated Java and C++ code for most models.

Conclusion. In general, high-level code generation helps to involve domain experts in the development and validation process at an early stage. In this work, domain experts can animate and simulate scenarios, inspect them in a domain-specific visualization, and provide feedback in natural language. Consequently, one can use feedback from domain experts early in the development phase. Another approach involving domain experts in the validation process is achieved with domain-specific scenarios (tests) in a behavior-driven development fashion, e.g., as done for Event-B with Gherkin using ProB [212, 74], or for ASMs with AsmetaV [47] and the AValLa language. The latter is also supported by the code generator Asm2C++ [36], which generates C++ tests from AValLa scenarios.

We demonstrated that high-level code generation leads to efficient performance for animation, model checking, and trace replay for many B models.

Yet, B2Program generates TypeScript/JavaScript code used in HTML documents. Depending on the domain experts' needs, one could support another graphical user interface, e.g., JavaFX, or 3D support via WebGL, OpenGL, etc., in the future.

Information About Included Manuscripts

In the following, I provide information about the manuscripts included in this thesis. Although I can declare that major parts of the manuscripts were written, implemented, and evaluated as my contribution, some parts have been co-authored by colleagues and students (in alphabetical order): Dominik Brandt, Jannik Dunkelau, David Geleßus, Christopher Happe, Michael Leuschel, Atif Mashkoor, and Sebastian Stock.

This thesis introduces the SimB simulator I implement in ProB2-UI [25]. SimB utilizes the ProB animator [153] for simulation. More details in ProB and ProB2-UI are available at: `https://prob.hhu.de/`.

Below, I declare which parts of all manuscripts are my contributions and which are contributions by my colleagues and students. Additionally, I declare where the chapters are published or submitted. All published manuscripts are peer-reviewed. Parts of the submission have also been peer-reviewed, while the newer parts are submitted to a journal with a peer-review process.

Validation of Formal Models by Timed Probabilistic Simulation

The chapter “Validation of Formal Models by Timed Probabilistic Simulation" (Chapter 2) is published as a full paper in the Proceedings of the “International Conference on Rigorous State-Based Methods" 2021 (ABZ 2021) [237]. The paper is co-authored by Fabian Vu, Michael Leuschel, and Atif Mashkoor. The initial idea of this paper originates from Michael Leuschel’s suggestion for a feature to control automatic simulation of multiple events in a formal model.

Fabian Vu’s contributions to the manuscript are:

- Writing the initial draft
- Discussion on which event/activations to choose when multiple activations schedule the same event
- Presentation of principles for probabilistic behavior
- Details on the concept of activation diagrams
- Presentation of the scheduling algorithm for activation diagrams

- Implementation of the SimB simulator applying the scheduling algorithm on activation diagrams in ProB2-UI
- Implementation of validation techniques for SimB
- Evaluation of simulation and validation of case studies
- Related work

Michael Leuschel's and Atif Mashkoor's contributions to the manuscript are:

- Improvements in presentation
- Presentation of principles for timing behavior
- Initial hints on activation diagrams
- Suggestions of some more case studies: Dueling Cowboys, Tourists, Leader Election
- Additions on related work

Full bibliographic reference: Fabian Vu, Michael Leuschel, and Atif Mashkoor. Validation of Formal Models by Timed Probabilistic Simulation. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 12709 in *Lecture Notes of Computer Science*, pages 81–96, Springer, 2021. doi: 10.1007/978-3-030-77543-8_6

Validation of Formal Models by Interactive Simulation

The chapter "Validation of Formal Models by Interactive Simulation" (Chapter 3) is published as a short paper in the Proceedings of the "International Conference on Rigorous State-Based Methods" 2023 (ABZ 2023) [236]. The paper is co-authored by Fabian Vu and Michael Leuschel.

Fabian Vu's contributions to the manuscript are:

- Writing the initial draft
- Idea to combine domain-specific visualization and simulation
- Conceptualization and implementation of *interactive simulation* in SimB
- Implementation of projection diagrams on VisB elements
- Presentation and demonstration on case studies
- Related work

Michael Leuschel's contributions to the manuscript are:

- Improvements in presentation
- Additions on related work

Full bibliographic reference: Fabian Vu and Michael Leuschel. Validation of Formal Models by Interactive Simulation. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 14010 of *Lecture Notes of Computer Science*, pages 59–69, Springer, 2023. doi: 10.1007/978-3-031-33163-3_5

Development and Validation of a Formal Model and Prototype for an Air Traffic Control System

The chapter “Development and Validation of a Formal Model and Prototype for an Air Traffic Control System" (Chapter 4) is submitted with the same title as a journal article in the *Formal Aspects of Computing (FAC Journal)*. The submitted journal article is an extended version of the full paper “Modeling and Analysis of a Safety-Critical Interactive System Through Validation Obligations", which is published in the Proceedings of the “International Conference on Rigorous State-Based Methods" 2023 (ABZ 2023) [88]. Both the submitted journal article and the conference paper [88] are co-authored by David Geleßus, Sebastian Stock, Fabian Vu, Michael Leuschel, and Atif Mashkoor. A prior version of the submitted journal article is part of Sebastian Stock’s doctoral thesis [215].

Fabian Vu’s contributions to the manuscript are:

- Writing the initial draft (equally with David Geleßus and Sebastian Stock)
- Background (equally with David Geleßus and Sebastian Stock)
- Introduction of AMAN System (equally with David Geleßus)
- Requirements extraction from the requirements document (equally with David Geleßus and Sebastian Stock)
- Initial versions of Arrival Manager in Event-B (equally with Sebastian Stock)
- Initial validation considerations (equally with Sebastian Stock)
- Development and presentation of a formal model for Arrival Manager in Event-B until `M9` (equally with David Geleßus and Sebastian Stock)
- Verifying and validating the formal model until `M9` (equally with David Geleßus and Sebastian Stock)
- Development and presentation of Arrival Manager prototype with VisB and SimB:
  - Development and presentation of domain-specific visualization in VisB for Arrival Manager at `M6` and `M9`

  - Development and presentation of SimB simulation for Arrival Manager
- Discussing lessons learned (equally with David Geleßus and Sebastian Stock)
- Related work (equally with David Geleßus and Sebastian Stock)

David Geleßus's, Sebastian Stock's, Michael Leuschel's and Atif Mashkoors's contributions (in addition to those mentioned above) to the manuscript are:

- Development, presentation, and proving of the formal model for Arrival Manager in Event-B on `M10`
- VisB visualization for `M10`
- Improvements on VisB visualization for Arrival Manager
- Discussions on Abstractions
- Improvements in presentation

Full bibliographic reference: David Geleßus, Sebastian Stock, Fabian Vu, Michael Leuschel, and Atif Mashkoor. Modeling and Analysis of a Safety-Critical Interactive System Through Validation Obligations. In *Proceedings ABZ (International Conference on Rigorous State-Based Methods)*, volume 14010 of *Lecture Notes of Computer Science*, pages 284–302, Springer, 2023. doi: 10.1007/978-3-031-33163-3_22

Validation of Reinforcement Learning Agents and Safety Shields with ProB

The chapter "Validation of Reinforcement Learning Agents and Safety Shields with ProB" (Chapter 5) is published as a full paper in the Proceedings of the "International Symposium on NASA Formal Methods" 2024 (NFM 2024) [232]. The paper is co-authored by Fabian Vu, Jannik Dunkelau, and Michael Leuschel. Initially, Jannik Dunkelau had the idea to evaluate RL agents for the highway environment, while Fabian Vu had the idea to generate traces from AI systems and validate them with SimB. Combining both ideas, Davin Holten initially experimented with using SimB to validate traces from the highway environment in his bachelor's thesis [108]. Davin Holten's bachelor's thesis was supervised by Jannik Dunkelau and Fabian Vu. In Davin Holten's bachelor's thesis, the RL agent was simulated outside of formal method tools, and the resulting traces were then validated with SimB. Chapter 5 simulates real AI behavior at runtime via SimB, which also enables runtime monitoring.

Fabian Vu's contributions to the manuscript are:

- Writing the initial draft
- Idea to use SimB for validating the performance of AI systems

- Implementation and presentation of runtime simulation (from an external source) and safety shielding with ProB and SimB
- Discussion on Validatability and Verifiability
- Creation and presentation of a formal model for the highway environment in B
- Creation of a domain-specific VisB visualization for the highway environment
- Validation of highway AI in ProB2-UI, particularly statistical validation with SimB and trace replay + presentation of results
- Improvements/Extensions of SimB's statistical validation
- Related work (equally with Jannik Dunkelau)

Jannik Dunkelau's and Michael Leuschel's contributions to the manuscript are:

- Idea to evaluate RL agents, and to use the highway environment as a case study
- Background on reinforcement learning, safety shielding, and highway environment
- Improvements in the presentation of runtime simulation and shielding
- Configuration and training of RL agents
- Improvements on the formal model for the highway environment
- Improvements on VisB visualization for the highway environment
- Suggestions on which properties and metrics to validate
- Discussions on explainable AI
- Additions on related work

Full bibliographic reference: Fabian Vu, Jannik Dunkelau, and Michael Leuschel. Validation of Reinforcement Learning Agents and Safety Shields with ProB. In *Proceedings NFM (International Symposium on NASA Formal Methods)*, volume 14627 of *Lecture Notes of Computer Science*, pages 279-297, Springer, 2024. doi: 10.1007/978-3-031-60698-4_16

Chapter 6 is an additional chapter of Chapter 5, presenting improvements and further results. Chapter 6 was written by Fabian Vu. Chapter 6 uses the Responsibility-Sensitive Safety (RSS) [209] technique. It was Michael Leuschel's idea to improve the safety of the highway AI by RSS. The formal model for RSS used in Chapter 6 was developed by Michael Leuschel. Section 6.2 presents the safety considerations implemented by Michael Leuschel in the formal RSS model. New reinforcement learning agents in Section 6.3 are now trained and presented by Fabian Vu. The evaluations are done on the same highway environment as in Chapter 5 by Fabian Vu. Threat to validity is discussed by Fabian Vu. Steps 1,2,3, and 5 in Figure 6.1 were done by Fabian Vu, while step 4 was done by Michael Leuschel. Jannik Dunkelau was not involved in Chapter 6.

Model Checking B Models via High-Level Code Generation

The chapter “Model Checking B Models via High-Level Code Generation" (Chapter 7) is published as a full paper in the Proceedings of the “International Conference on Formal Engineering Methods" 2022 (ICFEM 2022) [231]. The paper is co-authored by Fabian Vu, Dominik Brandt, and Michael Leuschel. The idea of researching code generation for model checking B models was proposed by Michael Leuschel. Florian Mager and Klaus Sausen were involved in early discussions on a student's project before writing the paper [231].

The paper is an extended work of the code generator B2Program, which existed before this thesis. In particular, B2Program was introduced in a previous paper [233], which resulted from my student's work during my master's studies and my bachelor's thesis [229]. Dominik Hansen was the supervisor of my bachelor's thesis.

Fabian Vu's contributions to the manuscript are:

- Writing the initial draft
- Design and implementation of model checking functionalities in B2Program, including
  - Model checking algorithm
  - Computation of outgoing transitions (initial discussions with Michael Leuschel how to treat operations with parameters)
  - Evaluation of invariants
- Design and implementation of parallelization and caching in B2Program
- Discussions on the limitations of B2Program (equally with Michael Leuschel)
- Evaluation of benchmarks for multiple setups
- Related work

Dominik Brandt's and Michael Leuschel's contributions to the manuscript are:

- Additions to the selection of benchmarks
- Assistance on executing benchmarks + uncovering performance lacks
- Improvements and refactoring in implementation
- Support in testing B2Program + uncovering bugs
- Additions on related work
- Improvements in presentation

Furthermore, Lucas Döring was involved in improving the performance of generated C++ code from B2Program.

Full bibliographic reference: 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)*, volume 13478 of *Lecture Notes of Computer Science*, pages 334–351, Springer, 2022. doi: 10.1007/978-3-031-17244-1_20

Chapter 9 is an additional chapter of Chapter 7 written by Fabian Vu. The improvements for B2Program include lifting restrictions on quantified constructs, rewriting predicates with set membership and subset of, and implementing operation reuse (technique presented by Leuschel in [150] for ProB). The improvements in Chapter 9 were all implemented by Fabian Vu. The rewrites shown in Table 9.1 are inspired by the fact that ProB [98] rewrites predicates with $\in$, $\subseteq$. These rewrites are specifically designed for B2Program and not taken from ProB. More optimizations are implemented in ProB [98], which are not covered in B2Program. Fabian Vu also evaluated the performance in Chapter 9.

Generating Interactive Documents for Domain-Specific Validation of Formal Models

The chapter “Generating Interactive Documents for Domain-Specific Validation of Formal Models" (Chapter 8) is published as a journal article in the “International Journal on Software Tools for Technology Transfer (STTT Journal)" 2024 [235]. The journal article is an extended version of the full paper “Generating Domain-Specific Interactive Validation Documents", which is published in the Proceedings of the “International Conference on Formal Methods for Industrial Critical Systems" 2022 (FMICS 2022) [234]. The journal article is part of the special issue for FMICS 2022, containing extended versions of papers invited and selected from the conference. Both the journal article [235] and the conference paper [234] are co-authored by Fabian Vu, Christopher Happe, and Michael Leuschel.

Both papers [234, 235] are extensions of the code generator B2Program, which existed before this thesis. In particular, B2Program was introduced from a previous paper [233], which resulted from my project's work during my master's studies and my bachelor's thesis [229]. Dominik Hansen was the supervisor of my bachelor's thesis.

The initial idea for the dynamic export was proposed by Fabian Vu, which was again inspired by the static export implemented by Michael Leuschel. Christopher Happe was neither involved in writing the conference paper [234] nor the journal article [235] but implemented B2Program's TypeScript/JavaScript code generation in an HTML document with support for VisB visualization, animation, and trace replay in his master's thesis [103]. Christopher Happe also experimented with the Landing Gear System and the Vehicle's Exterior Light System in the generated HTML document in his master's

thesis [103], which provides the proof-of-concept for the validation work in Chapter 8. Christopher Happe's master's thesis [103] was supervised by Fabian Vu. Fabian Vu made further improvements and extensions to the dynamic export.

Fabian Vu's contributions to the manuscript are:

- Writing the initial draft
- Idea to generate TypeScript/JavaScript and HTML code from domain-specific VisB visualization and classical B models for dynamic export
- Presentation of the validation workflow and the validation steps of dynamic export
- Improvements and presentation of code generation from B and VisB visualizations to TypeScript/JavaScript code and an HTML document
- Extensive presentation of the GUI
- Implementation and presentation of additional features for dynamic export: SimB features, writing natural language text, and trace export
- Implementation of generating model checking code for TypeScript/JavaScript
- Demonstration of validating case studies, including aspects of communication between modelers and domain experts
- Discussions on limitations
- Performance evaluation of model checking benchmarks + details on simulation benchmarks
- Related work

Christopher Happe's and Michael Leuschel's contributions to the manuscript are:

- Implementation and presentation of static export
- Implementation of dynamic export
  - Extension of B2Program to support TypeScript/JavaScript
  - Implementation of code generation from VisB visualizations to an HTML document
  - Implementation of animator and trace replay in an HTML document, including trace import
- Base for presentation of GUI
- Initial performance evaluation of simulation benchmarks
- Initial experiments with case studies

- Support in testing B2Program + uncovering bugs
- Additions on related work
- Improvements in presentation

Full bibliographic references:

- Fabian Vu, Christopher Happe, and Michael Leuschel. Generating Domain-Specific Interactive Validation Documents. In *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*, volume 13487 of *Lecture Notes of Computer Science*, pages 32–49, Springer, 2022. doi: 10.1007/978-3-031-15008-1_4
- Fabian Vu, Christopher Happe, and Michael Leuschel. Generating interactive documents for domain-specific validation of formal models. In *STTT Journal (International Journal on Software Tools for Technology Transfer)*, 26(2):147–168, 2024. doi: 10.1007/s10009-024-00739-0

Chapter 9 is also an additional chapter of Chapter 8. Information on Chapter 9 was provided before.

Bibliography

[1] Mohamed Abdel-Aty and Shengxuan Ding. A matched case-control analysis of autonomous vs human-driven vehicle accidents. *Nature Communications*, 15(1):4931, 2024.

[2] Tesnim Abdellatif and Kei-Léo Brousmiche. Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models. In *Proceedings NTMS*, pages 1–5, 2018.

[3] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-Based Implementation of Real-Time Applications. In *Proceedings EMSOFT*, ACM, pages 229–238, 2010.

[4] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[5] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[6] Jean-Raymond Abrial. From Z to B and then Event-B: assigning proofs to meaningful programs. In *Proceedings iFM*, LNCS 7940, pages 1–15. Springer, 2013.

[7] 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. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, November 2010.

[8] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An Open Extensible Tool Environment for Event-B. In *Proceedings ICFEM*, LNCS 4260, pages 588–605. Springer, 2006.

[9] Jean-Raymond Abrial and Dominique Cansell. Click'n Prove: Interactive Proofs within Set Theory. In *Proceedings TPHOL*, LNCS 2758, pages 1–24. Springer, 2003.

[10] Sten Agerholm. Translating specifications in VDM-SL to PVS. In *Proceedings TPHOLs*, LNCS 5674, pages 1–16. Springer, 1996.

[11] Bernhard K. Aichernig and Martin Tappler. Probabilistic black-box reachability checking (extended version). *Formal Methods in System Design*, 54(3):416–448, 2019.

[12] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings AAAI*, pages 2669–2678. AAAI Press, 2018.

[13] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[14] Mohamed Ben Amor. Hamsters - A New Task Model for Interactive Systems. Master's thesis, Universitaire Notre Dame de la Paix, 2009.

[15] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[16] Yamine Aït-Ameur, Idir Aït-Sadoune, Mickael Baron, and Jean-Marc Mota. Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes multi-modaux. *Journal d'Interaction Personne-Système*, 1, 10 2014.

[17] Phil Bagwell. Ideal Hash Trees. *Es Grands Champs*, 1195, 2001.

[18] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[19] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings iFM*, LNCS 2999, pages 1–20. Springer, 2004.

[20] David Basin, Ralf Sasse, and Jorge Toro-Pozo. Card Brand Mixup Attack: Bypassing the PIN in non-Visa Cards by Using Them for Visa Transactions. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 179–194. USENIX Association, Aug 2021.

[21] David Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV standard: Break, Fix, Verify. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1766–1781. IEEE, May 2021.

[22] David Basin, Patrick Schaller, and Jorge Toro-Pozo. Inducing authentication failures to bypass credit card PINs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3065–3079. USENIX Association, August 2023.

[23] Rémi Bastide, David Navarre, and Philippe Palanque. A tool-supported design framework for safety critical interactive systems. *Interacting with Computers*, 15(3):309–328, 2003. Computer-Aided Design of User Interface.

[24] Jens Bendisposto. *Directed and Distributed Model Checking of B-Specifications*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2014.

[25] Jens Bendisposto, David Geleßus, Yumiko Jansing, Michael Leuschel, Antonia Pütz, Fabian Vu, and Michelle Werth. ProB2-UI: A Java-Based User Interface for ProB. In *Proceedings FMICS*, LNCS 12863, pages 193–201. Springer, 2021.

[26] 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. *ArXiv*, abs/1603.04401, 2016.

[27] Jens Bendisposto, Sebastian Krings, and Michael Leuschel. Who watches the watchers: Validating the ProB Validation Tool. In *Proceedings F-IDE*, EPTCS 149. Open Publishing Association, 2014.

[28] Jens Bendisposto and Michael Leuschel. Proof Assisted Model Checking for B. In *Proceedings ICFEM*, LNCS 5885, pages 504–520. Springer, 2009.

[29] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, LNCS 1066, pages 232–243. Springer, 1996.

[30] Dines Bjørner. The Vienna Development Method (VDM). In *Mathematical Studies of Information Processing*, LNCS 75, pages 326–359. Springer Berlin Heidelberg, 1979.

[31] Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. formal MVC: A Pattern for the Integration of ASM Specifications in UI Development. In *Proceedings ABZ*, LNCS 14010, pages 340–357. Springer, 2023.

[32] Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. From Concept to Code: Unveiling a Tool for Translating Abstract State Machines into Java Code. In *Proceedings ABZ*, pages 160–178. Springer, 2024.

[33] Silvia Bonfanti, Marco Carissoni, Angelo Gargantini, and Atif Mashkoor. Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino. In *Proceedings NFM*, LNCS 10227, pages 295–301. Springer, 2017.

[34] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. AsmetaA: Animator for Abstract State Machines. In *Proceedings ABZ*, LNCS 10817, pages 369–373. Springer, 2018.

[35] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. Validation of Transformation from Abstract State Machine Models to C++ Code. In *Proceedings ICTSS*, LNCS 11146, pages 17–32. Springer, 2018.

[36] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. Design and validation of a C++ code generator from Abstract State Machines specifications. *Journal of Software: Evolution and Process*, 32(2), 2020.

[37] Silvia Bonfanti, Elvinia Riccobene, and Patrizia Scandurra. A Runtime Safety Enforcement Approach by Monitoring and Adaptation. In *Proceedings ECSA*, LNCS 12857, pages 20–36. Springer, 2021.

[38] Richard Bonichon, David Déharbe, Thierry Lecomte, and Valério Medeiros Jr. LLVM-based code generation for B. In *Proceedings SBMF*, LNCS 8941, pages 1–16. Springer, 2014.

[39] Frédéric Boniol and Virginie Wiels. The landing gear system case study. In *ABZ 2014: The Landing Gear Case Study*, CCIS 433, pages 1–18. Springer, 2014.

[40] Egon Börger. The abstract state machines method for high-level system design and analysis. In *Formal Methods: State of the Art and New Directions*, pages 79–116. Springer, 2010.

[41] Howard Bowman and Rodolfo Gomez. Process Calculi: LOTOS. In *Concurrency Theory: Calculi and Automata for Modelling Untimed and Timed Concurrent Systems*, pages 19–54. Springer, 2006.

[42] Marco Bozzano, Harold Bruintjes, Alessandro Cimatti, Joost-Pieter Katoen, Thomas Noll, and Stefano Tonetta. Formal Methods for Aerospace Systems: Achievements and Challenges. In *Cyber-Physical System Design From an Architecture Analysis Viewpoint: Communications of NII Shonan Meetings*, pages 133–159. Springer, 2017.

[43] Michael Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. In *Proceedings FM*, LNCS 3582, pages 221–236. Springer, 2005.

[44] José Creissac Campos, Camille Fayollas, Michael D. Harrison, Célia Martinie, Paolo Masci, and Philippe Palanque. Supporting the analysis of safety critical user interfaces: An exploration of three formal tools. *ACM Trans. Comput. Hum. Interact.*, 27(5):35:1–35:48, 2020.

[45] José Creissac Campos and Michael D. Harrison. Systematic analysis of control panel interfaces using formal tools. In *Interactive Systems. Design, Specification, and Verification*, LNCS 5136, pages 72–85. Springer, 2008.

[46] Dominique Cansell, Dominique Méry, and Joris Rehm. Time Constraint Patterns for Event B Development. In *Formal Specification and Development in B*, LNCS 4355, pages 140–154. Springer, 2007.

[47] Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Scenario-Based Validation Language for ASMs. In *Proceedings ABZ*, LNCS 5238, pages 71–84. Springer, 2008.

[48] Mats Carlsson, Johan Widen, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science Kista, 1988.

[49] Néstor Cataño and Victor Rívera. EventB2Java: A Code Generator for Event-B. In *Proceedings NFM*, LNCS 9690, pages 166–171. Springer, 2016.

[50] Kai Lai Chung. Markov chains. *Springer-Verlag, New York*, 1967.

[51] ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2016. Available at `http://www.atelierb.eu/`.

[52] Ernie Cohen. Validating the Microsoft Hypervisor. In *Proceedings FM*, LNCS 4085, pages 81–81. Springer, 2006.

[53] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *Proceedings CAV*, LNCS 10981, pages 183–190. Springer, 2018.

[54] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Topological sort*, pages 573–576. MIT press, 2022.

[55] Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl, and Peter W. V. Tran-Jørgensen. Towards Enabling Overture as a Platform for Formal Notation IDEs. *arXiv preprint arXiv:1508.03893*, 2015.

[56] Alcino Cunha, Nuno Macedo, and Eunsuk Kang. Task Model Design and Analysis with Alloy. In *Proceedings ABZ*, LNCS 14010, pages 303–320. Springer, 2023.

[57] Bruno d'Ausbourg. Using Model Checking for the Automatic Validation of User Interfaces Systems. In *Design, Specification and Verification of Interactive Systems '98*, pages 242–260. Springer Vienna, 1998.

[58] Marc de Jonge and Theo C. Ruys. The SpinJa Model Checker. In *Proceedings SPIN Workshop*, LNCS 6349, pages 124–128. Springer, 2010.

[59] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings TACAS*, LNCS 4963, pages 337–340. Springer, 2008.

[60] Paulus A.J.M. de Wit and Roberto Moraes Cruz. Learning from AF447: Human-machine interaction. *Safety Science*, 112:48–56, 2019.

[61] Arnaud Dieumegard, Ning Ge, and Eric Jenn. Event-B at Work: Some Lessons Learnt from an Application to a Robot Anti-collision Function. In *Proceedings NFM*, LNCS 10227, pages 327–341. Springer, 2017.

[62] Ben L. DiVito and Larry W. Roberts. Using Formal Methods to Assist in the Requirements Analysis of the Space Shuttle GPS Change Request. Technical report, NASA Langley Research Center, 1996.

[63] Alan J. Dix. Formal methods. In *Perspectives on HCI: Diverse Approaches*, pages 9–43, London, 1995. Academic Press.

[64] Ivaylo Dobrikov and Michael Leuschel. Enabling Analysis for Event-B. In *Proceedings ABZ*, LNCS 9675, pages 102–118. Springer, 2016.

[65] Daniel Dollé, Didier Essamé, and Jérôme Falampin. B dans le transport ferroviaire. L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques*, 22(1):11–32, 2003.

[66] Lucas Döring. Feasibility and Uses of a Superset of B0 for embedded code-generation. Master's thesis, Heinrich-Heine-Universität Düsseldorf, 2023.

[67] Marc Dragon, Andy Gimblett, and Markus Roggenbach. A Simulator for Timed CSP. In *Proceedings AVoCS*, volume 46 of *Electronic Communications of the EASST*, 2011.

[68] Marlon Dumas and Arthur H. M. Ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proceedings UML*, pages 76–90. Springer, 2001.

[69] Andrew Edmunds. Templates for Event-B Code Generation. In *Proceedings ABZ*, LNCS 8477, pages 284–289. Springer, 2014.

[70] Yrvann Emzivat, Benoît Delahaye, Didier Lime, and Olivier H. Roux. Probabilistic Time Petri Nets. In *Application and Theory of Petri Nets and Concurrency*, LNCS 9698, pages 261–280. Springer, 2016.

[71] Didier Essamé and Daniel Dollé. B in Large-Scale Projects: The Canarsie Line CBTC Experience. In *B 2007: Formal Specification and Development in B*, LNCS 4355, pages 252–254. Springer, 2006.

[72] Camille Fayollas, Célia Martinie, Philippe Palanque, Eric Barboni, Racim Fahssi, and Arnaud Hamon. Exploiting Action Theory as a Framework for Analysis and Design of Formal Methods Approaches: Application to the CIRCUS Integrated Development Environment. In *The Handbook of Formal Methods in Human-Computer Interaction*, HCIS, pages 465–504. Springer, 2017.

[73] Aaron W. Fifarek, Lucas G. Wagner, Jonathan A. Hoffman, Benjamin D. Rodes, M. Anthony Aiello, and Jennifer A. Davis. SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements. In *Proceedings NFM*, LNCS 10227, pages 420–426. Springer, 2017.

[74] Tomas Fischer and Dana Dghyam. Formal Model Validation Through Acceptance Tests. In *Proceedings RSSRail*, LNCS 11495, pages 159–169. Springer, 2019.

[75] Ronald A. Fisher. Theory of Statistical Estimation. *Mathematical Proceedings of the Cambridge Philosophical Society*, 22(5):700–725, 1925.

[76] Marc Fontaine, Andy Gimblett, Faron Moller, Hoang Nga Nguyen, and Markus Roggenbach. Timed CSP Simulator. In *Proceedings of the Posters & Tool demos Session, iFM & ABZ*, 2012.

[77] Peter Forbrig, Célia Martinie, Philippe Palanque, Marco Winckler, and Racim Fahssi. Rapid Task-Models Development Using Sub-models, Sub-routines and Generic Components. In *Human-Centered Software Engineering*, LNCS 8742, pages 144–163. Springer, 2014.

[78] Kevin Forsberg and Harold Mooz. The Relationship of System Engineering to the Project Cycle. *INCOSE International Symposium*, 1(1):57–65, 1991.

[79] Nathan Fulton and André Platzer. Safe Reinforcement Learning via Formal Methods: Toward Safe Control through Proof and Learning. *Proceedings AAAI*, 32(1), 2018.

[80] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code Generation for Event-B. In *Proceedings iFM*, LNCS 8739, pages 323–338. Springer, 2014.

[81] Hubert Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In *Proceedings TACAS*, LNCS 1384, pages 68–84. Springer, 1998.

[82] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Metamodel-based Simulator for ASMs. In *Proceedings ASM Workshop*, 2007.

[83] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *JUCS - Journal of Universal Computer Science*, 14(12):1949–1983, 2008.

[84] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Model-driven language engineering: The ASMETA case study. In *Proceedings ICSEA*, pages 373–378. IEEE, 2008.

[85] Ning Ge, Arnaud Dieumegard, Eric Jenn, Bruno d'Ausbourg, and Yamine Aït-Ameur. Formal development process of safety-critical embedded human machine interface systems. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–8. IEEE, 2017.

[86] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2018.

[87] David Geleßus and Michael Leuschel. ProB and Jupyter for Logic, Set Theory, Theoretical Computer Science and Formal Methods. In *Proceedings ABZ*, LNCS 12071, pages 248–254. Springer, 2020.

[88] David Geleßus, Sebastian Stock, Fabian Vu, Michael Leuschel, and Atif Mashkoor. Modeling and Analysis of a Safety-Critical Interactive System Through Validation Obligations. In *Proceedings ABZ*, LNCS 14010, pages 284–302. Springer, 2023.

[89] Romain Geniet and Neeraj Kumar Singh. Refinement Based Formal Development of Human-Machine Interface. In *Proceedings STAF*, LNCS 11176, pages 240–256. Springer International Publishing, 2018.

[90] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johan Schumann. Generation of Formal Requirements from Structured Natural Language. In *Proceedings REFSQ*, LNCS 12045, pages 19–35. Springer, 2020.

[91] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. Automated formalization of structured natural language requirements. *Information and Software Technology*, 137:106590, 09 2021.

[92] Divya Gopinath, Guy Katz, Corina S. Păsăreanu, and Clark Barrett. DeepSafe: A Data-Driven Approach for Assessing Robustness of Neural Networks. In *Proceedings ATVA*, LNCS 11138, pages 3–19. Springer, 2018.

[93] Jan Gruteser, David Geleßus, Michael Leuschel, Jan Roßbach, and Fabian Vu. A Formal Model of Train Control with AI-Based Obstacle Detection. In *Proceedings RSSRail*, LNCS 14198, pages 128–145. Springer, 2023.

[94] Jan Gruteser and Michael Leuschel. Validation of RailML Using ProB. In *Proceedings ICECCS 2024*, LNCS 14784, pages 245–256. Springer, 2025.

[95] Jan Gruteser, Jan Roßbach, Fabian Vu, and Michael Leuschel. Using Formal Models, Safety Shields and Certified Control to Validate AI-Based Train Systems. In *Proceedings FMAS*, EPTCS 411, pages 151–159. Open Publishing Association, 2024.

[96] Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. Understanding Inconsistency in Azure Cosmos DB with TLA+. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 1–12. IEEE, 2023.

[97] Stefan Hallerstede and Thai Son Hoang. Qualitative Probabilistic Modelling in Event-B. In *Proceedings iFM*, LNCS 4591, pages 293–312. Springer, 2007.

[98] Stefan Hallerstede and Michael Leuschel. Constraint-based deadlock checking of high-level specifications. *Theory and Practice of Logic Programming*, 11(4–5):767–782, 2011.

[99] Dominik Hansen and Michael Leuschel. Translating TLA + to B for Validation with ProB. In *Proceedings iFM*, volume LNCS 7321, pages 24–38. Springer, 2012.

[100] Dominik Hansen and Michael Leuschel. Translating B to TLA+ for Validation with TLC. In *Proceedings ABZ*, LNCS 8477, pages 40–55. Springer, 2014.

[101] 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. *International Journal on Software Tools for Technology Transfer*, 22(3):315–332, 2020.

[102] Hans Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535, 1995.

[103] Christopher Happe. Validierung von B Modellen mit Hilfe von Codegenerierung nach HTML und JavaScript. Master's thesis, Heinrich-Heine-Universität Düsseldorf, 2022.

[104] Ian James Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4:330–339, 1989.

[105] Joseph Herkert, Jason Borenstein, and Keith Miller. The Boeing 737 MAX: Lessons for engineering ethics. *Science and engineering ethics*, 26:2957–2974, 2020.

[106] Thai Son Hoang. Reasoning about almost-certain convergence properties using Event-B. *Science of Computer Programming*, 81:108–121, 2014.

[107] Tony Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[108] Davin Holten. Validierung von Reinforcement-Learning-Agenten mittels Trace-Analyse. Bachelor's thesis, Heinrich-Heine-Universität Düsseldorf, 2023.

[109] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[110] Frank Houdek and Alexander Raschke. Adaptive Exterior Light and Speed Control System. Available at `https://abz2020.uni-ulm.de/case-study`, 2019.

[111] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety Verification of Deep Neural Networks. In *Proceedings CAV*, LNCS 10426, pages 3–29. Springer, 2017.

[112] Xiaowei Huang, Wenjie Ruan, Qiyi Tang, and Xingyu Zhao. Bridging Formal Methods And Machine Learning With Global Optimisation. In *Proceedings ICFEM*, LNCS 13478, pages 1–19. Springer, 2022.

[113] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant A Tutorial. *Rapport Technique*, 178, 1997.

[114] Wilson Ifill, Steve Schneider, and Helen Treharne. Augmenting B with Control Annotations. In *Proceedings B'07*, LNCS 4355, pages 34–48. Springer, 2007.

[115] Dubravka Ilić, Linas Laibinis, Timo Latvala, Elena Troubitsyna, and Kimmo Varpaaniemi. *Deployment in the Space Sector*, pages 45–62. Springer, 2013.

[116] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. IEEE, 1991.

[117] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[118] Daniel Jackson, Valerie Richmond, Mike Wang, Jeff Chow, Uriel Guajardo, Soonho Kong, Sergio Campos, Geoffrey Litt, and Nikos Aréchiga. Certified control: An architecture for verifiable safety of autonomous vehicles. *CoRR*, abs/2104.06178, 2021.

[119] Peter W. V. Jørgensen, Morten Larsen, and Luís D. Couto. A Code Generation Platform for VDM. In *Proceedings Overture Workshop*. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446, 2015.

[120] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *Proceedings TACAS*, LNCS 9035, pages 692–707. Springer, 2015.

[121] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proceedings CAV*, LNCS 10426, pages 97–117. Springer, 2017.

[122] Maurice G. Kendall, Alan Stuart, and J. Keith Ord. Kendall's Advanced Theory of Statistics. In *Oxford University Press*, 1987.

[123] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[124] Tsutomu Kobayashi, Martin Bondu, and Fuyuki Ishikawa. Formal Modelling of Safety Architecture for Responsibility-Aware Autonomous Vehicle via Event-B Refinement. In *Proceedings FM*, LNCS 14000, pages 533–549. Springer, 2023.

[125] Bettina Könighofer, Florian Lorber, Nils Jansen, and Roderick Bloem. Shield synthesis for reinforcement learning. In *Proceedings ISoLA*, LNCS 12476, pages 290–306. Springer, 2020.

[126] 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*, 58(1):160–187, 2021.

[127] 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*, LNCS 11023, pages 275–295. Springer, 2018.

[128] Sebastian Krings. *Towards Infinite-State Symbolic Model Checking for B and Event-B*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, August 2017.

[129] Sebastian Krings, Joshua Schmidt, Carola Brings, Marc Frappier, and Michael Leuschel. A Translation from Alloy to B. In *Proceedings ABZ*, LNCS 10817, pages 71–86. Springer, 2018.

[130] Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In *Proceedings TACAS*, LNCS 8413, pages 389–391. Springer, 2014.

[131] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pages 200–204. Springer, 2002.

[132] Marta Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. Symbolic model checking for probabilistic timed automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, LNCS 3253, pages 293–308. Springer, 2004.

[133] Lukas Ladenberger. *Rapid Creation of Interactive Formal Prototypes for Validating Safety-Critical Systems*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2016.

[134] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising Event-B Models with B-Motion Studio. In *Proceedings FMICS*, LNCS 5825, pages 202–204. Springer, 2009.

[135] Lukas Ladenberger, Dominik Hansen, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. Validation of the ABZ Landing Gear System Using ProB. *International Journal on Software Tools for Technology Transfer*, 19(2):187–203, 2017.

[136] Lukas Ladenberger and Michael Leuschel. Mastering the Visualization of Larger State Spaces with Projection Diagrams. In *Proceedings ICFEM*, LNCS 9407, pages 153–169. Springer, 2015.

[137] Lukas Ladenberger and Michael Leuschel. BMotionWeb: A Tool for Rapid Creation of Formal Prototypes. In *Proceedings SEFM*, LNCS 9763, pages 403–417. Springer, 2016.

[138] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[139] Leslie Lamport. Real-Time Model Checking Is Really Simple. In *Proceedings CHARME*, LNCS 3725, pages 162–175. Springer, 2005.

[140] Matthew Landers and Afsaneh Doryab. Deep Reinforcement Learning Verification: A Survey. *ACM Computing Surveys*, 55(14s):1–31, 2023.

[141] Peter Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The overture initiative integrating tools for VDM. *ACM SIGSOFT Software Engineering Notes*, 35:1–6, 2010.

[142] Thierry Lecomte. Safe and Secure Architecture Using Diverse Formal Methods. In *Proceedings ISoLA*, LNCS 13704, pages 321–333. Springer, 2022.

[143] Thierry Lecomte, David Deharbe, Denis Sabatier, Etienne Prun, Patrick Péronne, Emmanuel Chailloux, Steven Varoumas, Adilla Susungi, and Sylvain Conchon. Low Cost High Integrity Platform. In *Proceedings 10th European Congress Embdedded Real Time Software and System*, 2020.

[144] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical Model Checking: An Overview. In *Proceedings Runtime Verification*, LNCS 6418, pages 122–135. Springer, 2010.

[145] Axel Legay, Anna Lukina, Louis Marie Traonouez, Junxing Yang, Scott A. Smolka, and Radu Grosu. Statistical model checking. In *Computing and Software Science: State of the Art and Perspectives*, LNCS 10000, pages 478–504. Springer, 2019.

[146] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[147] Edouard Leurent. An environment for autonomous driving decision-making. `https://github.com/eleurent/highway-env`, 2018.

[148] Michael Leuschel. Formal Model-Based Constraint Solving and Document Generation. In *Proceedings SBMF*, LNCS 10090, pages 3–20. Springer, 2016.

[149] Michael Leuschel. Spot the Difference: A Detailed Comparison Between B and Event-B. In *Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday*, LNCS 12750, pages 147–172. Springer, 2021.

[150] Michael Leuschel. Operation Caching and State Compression for Model Checking of High-Level Models - How to Have Your Cake and Eat It. In *Proceedings iFM*, LNCS 13274, pages 129–145. Springer, 2022.

[151] Michael Leuschel, Jens Bendisposto, and Dominik Hansen. Unlocking the Mysteries of a Formal Model of an Interlocking System. In *Proceedings Rodin Workshop*, 2014.

[152] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *Proceedings FME*, LNCS 2805, pages 855–874. Springer, 2003.

[153] Michael Leuschel and Michael Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[154] Michael Leuschel, Mareike Mutz, and Michelle Werth. Modelling and Validating an Automotive System in Classical B and Event-B. In *Proceedings ABZ*, LNCS 12071, pages 335–350. Springer, 2020.

[155] Michael Leuschel, Mireille Samia, Jens Bendisposto, and Li Luo. Easy Graphical Animation and Formula Visualisation for Teaching B. *The B Method: from Research to Teaching*, pages 17–32, 2008.

[156] Michael Leuschel and Edd Turner. Visualising Larger State Spaces in ProB. In *ZB 2005: Formal Specification and Development in Z and B*, LNCS 3455, pages 6–23. Springer, 2005.

[157] Michael Leuschel, Fabian Vu, and Kristin Rutenkolk. Case Study: Safety Controller for Autonomous Driving on Highways. In *Proceedings ABZ*, LNCS 15728, pages 203–211. Springer, 2025.

[158] Sarah M. Loos, André Platzer, and Ligia Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *Proceedings FM*, LNCS 6664, pages 42–56. Springer, 2011.

[159] Amel Mammar and Michael Leuschel. Modeling and Verifying an Arrival Manager Using Event-B. In *Proceedings ABZ*, LNCS 14010, pages 321–339. Springer, 2023.

[160] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning Markov Decision Processes for Model Checking. In *Proceedings QFM*, EPTCS 103, pages 49–63. Open Publishing Association, 2012.

[161] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105(2):255–299, 2016.

[162] Célia Martinie, Philippe Palanque, and Marco Winckler. Structuring and Composition Mechanisms to Address Scalability Issues in Task Models. In *Proceedings INTERACT*, LNCS 6948, pages 589–609. Springer, 2011.

[163] Paolo Masci and César A. Muñoz. A Graphical Toolkit for the Validation of Requirements for Detect and Avoid Systems. In *Proceedings TAP*, LNCS 12165, pages 155–166. Springer, 2020.

[164] Atif Mashkoor and Jean-Pierre Jacquot. Utilizing Event-B for domain engineering: a critical analysis. *Requirements Engineering*, 16(3):191–207, 2011.

[165] Atif Mashkoor and Jean-Pierre Jacquot. Validation of formal specifications through transformation and animation. *Requirements Engineering*, 22(4):433–451, 2017.

[166] Atif Mashkoor, Michael Leuschel, and Alexander Egyed. Validation Obligations: A Novel Approach to Check Compliance between Requirements and their Formal Specification. In *ICSE'21 NIER*, pages 1–5. IEEE, 2021.

[167] Atif Mashkoor, Faqing Yang, and Jean-Pierre Jacquot. Refinement-based Validation of Event-B Specifications. *Software and Systems Modeling*, 16(3):789–808, 2017.

[168] Franco Mazzanti and Alessio Ferrari. Ten Diverse Formal Models for a CBTC Automatic Train Supervision System. In *Proceedings MARS/VPT*, EPTCS 268, page 104–149. Open Publishing Association, 2018.

[169] Peter Mehlitz, Neha Rungta, and Willem Visser. A hands-on Java Pathfinder tutorial. In *Proceedings ICSE*, pages 1493–1495. IEEE, 2013.

[170] Ismail Mendil, Neeraj Kumar Singh, Yamine Aït-Ameur, Dominique Méry, and Philippe Palanque. An Integrated Framework for the Formal Analysis of Critical Interactive Systems. In *Proceedings APSEC*, pages 139–148. IEEE, 2020.

[171] Dominique Méry and Neeraj Kumar Singh. Real-Time Animation for Formal Specification. In *Proceedings Complex Systems Design & Management*, pages 49–60. Springer, 2010.

[172] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-B models. In *Proceedings SoICT*, SoICT '11, page 179–188. Association for Computing Machinery, 2011.

[173] Robin Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer, 1980.

[174] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge University Press, 1999.

[175] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen

King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[176] Christopher Z. Mooney. *Monte Carlo Simulation*. Quantitative Applications in the Social Sciences 116. Sage, 1997.

[177] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3):50–57, 2013.

[178] David Navarre, Philippe Palanque, Fabio Paternò, Carmen Santoro, and Rémi Bastide. A Tool Suite for Integrating Task and System Models through Scenarios. In *Interactive Systems: Design, Specification, and Verification*, LNCS 2220, pages 88–113. Springer, 2001.

[179] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, 2015.

[180] Carlos A. L. Nunes and Ana C. R. Paiva. Automatic Generation of Graphical User Interfaces From VDM++ Specifications. In *Proceedings ICSEA*, pages 399–404, 2011.

[181] Tomohiro Oda, Keijiro Araki, Yasuhiro Yamamoto, Kumiyo Nakakoji, Han-Myung Chang, and Peter Larsen. Specifying Abstract User Interface in VDM-SL. In *Proceedings Overture Workshop*, pages 5–19. Overture, December 2020.

[182] Tomohiro Oda, Yasuhiro Yamamoto, Kumiyo Nakakoji, Keijiro Araki, and Peter Gorm Larsen. VDM Animation for a Wider Range of Stakeholders. In *Proceedings Overture Workshop*, pages 18–32, 2015.

[183] Ian Oliver. Experiences in Using B and UML in Industrial Development. In *B 2007: Proceedings Formal Specification and Development in B*, LNCS 4355, pages 248–251. Springer, 2007.

[184] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction – CADE-11*, LNCS 607, pages 748–752. Springer, 1992.

[185] Philippe Palanque and José Creissac Campos. AMAN Case Study. `https://drive.google.com/file/d/1IqftxQIvrWpX1lcRts3WJzrBH7a3dMln/view`.

[186] Philippe Palanque and José Creissac Campos. AMAN Case Study. In *Proceedings ABZ*, LNCS 14010, pages 265–283. Springer, 2023.

[187] Maurizio Palmieri, Cinzia Bernardeschi, and Paolo Masci. A framework for FMI-based co-simulation of human–machine interfaces. *Software and Systems Modeling*, 19(3):601–623, 2020.

[188] Terence Parr. StringTemplate Website. `http://www.stringtemplate.org/`, 2013. Accessed: 2021-09-23.

[189] Corina S. Păsăreanu, Ravi Mangal, Divya Gopinath, Sinem Getir Yaman, Calum Imrie, Radu Calinescu, and Huafeng Yu. Closed-Loop Analysis of Vision-Based Autonomous Systems: A Case Study. In *Proceedings CAV*, LNCS 13964, pages 289–303. Springer, 2023.

[190] Evertjan Peer, Vlado Menkovski, Yingqian Zhang, and Wan-Jui Lee. Shunting Trains with Deep Reinforcement Learning. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3063–3068. IEEE, 2018.

[191] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[192] Hoang Pham. *Software Reliability*. Springer, 2000.

[193] Dung T. Phan, Radu Grosu, Nils Jansen, Nicola Paoletti, Scott A. Smolka, and Scott D. Stoller. Neural Simplex Architecture. In *Proceedings NFM*, LNCS 12229, pages 97–114. Springer, 2020.

[194] Daniel Plagge and Michael Leuschel. Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *International Journal on Software Tools for Technology Transfer*, 12(1):9–21, 2010.

[195] André Platzer and Edmund M. Clarke. Formal Verification of Curved Flight Collision Avoidance Maneuvers: A Case Study. In *Proceedings FM*, LNCS 5850, pages 547–562. Springer, 2009.

[196] Nicolas Privault. *Discrete-Time Markov Chains*, pages 89–113. SUMS. Springer, 2013.

[197] Pouria Razzaghi, Amin Tabrizian, Wei Guo, Shulu Chen, Abenezer Taye, Ellis Thompson, Alexis Bregeon, Ali Baheri, and Peng Wei. A Survey on Reinforcement Learning in Aviation Applications. *Engineering Applications of Artificial Intelligence*, 136:108911, 2024.

[198] 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*, pages 179–190, 2007.

[199] Jesse Reimann, Nico Mansion, James Haydon, Benjamin Bray, Agnishom Chattopadhyay, Sota Sato, Masaki Waga, Étienne André, Ichiro Hasuo, Naoki Ueda, and Yosuke Yokoyama. Temporal Logic Formalisation of ISO 34502 Critical Scenarios: Modular Construction with the RSS Safety Distance. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 186–195. Association for Computing Machinery, 2024.

[200] Víctor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. Code generation for Event-B. *International Journal on Software Tools for Technology Transfer*, 19(1):31–52, 2017.

[201] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proceedings IJCAI*, pages 2651–2659, 2018.

[202] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, 1(5):206–215, 2019.

[203] Ahmad El Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep Reinforcement Learning framework for Autonomous Driving. *Electronic Imaging*, 29(19):70–76, January 2017.

[204] Guy Scher, Sadra Sadraddini, and Hadas Kress-Gazit. Probabilistic Rare-Event Verification for Temporal Logic Robot Tasks. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 12409–12415. IEEE, 2023.

[205] Maike Schwammberger, Christopher Harper, Gleifer Vaz Alves, Greg Chance, Tony Pipe, and Kerstin Eder. Integrating Formal Verification and Simulation-based Assertion Checking in a Corroborative V&V Process. *CoRR*, abs/2208.05273, 2022.

[206] Thierry Servat. BRAMA: A New Graphic Animation Tool for B Models. In *B 2007: Formal Specification and Development in B*, LNCS 4355, pages 274–276. Springer, 2006.

[207] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. Toward Verified Artificial Intelligence. *Communications of the ACM*, 65(7):46–55, 2022.

[208] Lui Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[209] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. On a Formal Model of Safe and Scalable Self-driving Cars. *CoRR*, abs/1708.06374, 2017.

[210] Neeraj Kumar Singh, Yamine Aït-Ameur, Romain Geniet, Dominique Méry, and Philippe Palanque. On the Benefits of Using MVC Pattern for Structuring Event-B Models of WIMP Interactive Applications. *Interacting with Computers*, 33(1):92–114, 2021.

[211] Neeraj Kumar Singh, Yamine Aït-Ameur, Ismail Mendil, Dominique Mery, David Navarre, Philippe Palanque, and Marc Pantel. F3FLUID: A formal framework for developing safety-critical interactive systems in FLUID. *Journal of Software: Evolution and Process*, 35(7):e2439, 2022.

[212] Colin Snook, Thai Son Hoang, Dana Dghaym, Asieh Salehi Fathabadi, and Michael Butler. Domain-specific scenarios for refinement-based methods. *Journal of Systems Architecture*, 112:101833, 2021.

[213] Colin Snook, Thai Son Hoang, Asieh Salehi Fathabadi, Dana Dghaym, and Michael Butler. Scenario Checker: An Event-B tool for validating abstract models. In *Proceedings of the 9th Rodin User and Developer Workshop*, pages 12–14, 2021.

[214] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall Hemel Hempstead, 1992.

[215] Sebastian Stock. *Early and Systematic Validation of Formal Models*. PhD thesis, Johannes Kepler University Linz, June 2024.

[216] Sebastian Stock, Atif Mashkoor, and Alexander Egyed. Validation-Driven Development. In *Proceedings ICFEM*, LNCS 14308, pages 191–207. Springer, 2023.

[217] Sebastian Stock, Fabian Vu, David Geleßus, Michael Leuschel, Atif Mashkoor, and Alexander Egyed. Validation by Abstraction and Refinement. In *Proceedings ABZ*, LNCS 14010, pages 160–178. Springer, 2023.

[218] Sebastian Stock, Fabian Vu, Atif Mashkoor, Michael Leuschel, and Alexander Egyed. IVOIRE deliverable 1.1: Classification of existing VOs & tools and formalization of VOs semantics. *CoRR*, abs/2205.06138, 2022.

[219] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[220] Martin Tappler and Bernhard K. Aichernig. Differential Safety Testing of Deep RL Agents Enabled by Automata Learning. In *Proceedings AISoLA*, LNCS 14380, pages 138–159. Springer, 2024.

[221] Martin Tappler, Filip Cano Cordoba, Bernhard K. Aichernig, and Bettina Könighofer. Search-Based Testing of Reinforcement Learning. In *Proceedings IJCAI*, pages 503–510, 2022.

[222] Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. Formal Methods in Industry. *Formal Aspects of Computing*, 2024.

[223] 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.

[224] Hoang-Dung Tran, Feiyang Cai, Manzanas Lopez Diego, Patrick Musau, Taylor T. Johnson, and Xenofon Koutsoukos. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–22, 2019.

[225] Freark van der Berg and Alfons Laarman. SpinS: Extending LTSmin with Promela through SpinJa. *Electronic Notes in Theoretical Computer Science*, 296:95–105, 2013.

[226] Chitra Venkatramani and Tzi-cker Chiueh. Design, implementation, and evaluation of a software-based real-time Ethernet protocol. *ACM SIGCOMM Computer Communication Review*, 25(4):27–37, 1995.

[227] Jean-Christophe Voisinet, Bruno Tatibouët, and Ahmed Hammad. JBTools: An Experimental Platform for the Formal B Method. In *Proceedings PPPJ/IRE*, pages 137–139. National University of Ireland, 2002.

[228] George A. Vouros. Explainable Deep Reinforcement Learning: State of the Art and Challenges. *ACM Computing Surveys*, 55(5):1–39, 2022.

[229] Fabian Vu. A high-level code generator for safety-critical B models. Bachelor's thesis, Heinrich-Heine-Universität Düsseldorf, 2018.

[230] Fabian Vu. Simulation and Verification of Reactive Systems in Lustre with ProB. Master's thesis, Heinrich-Heine-Universität Düsseldorf, 2020.

[231] Fabian Vu, Dominik Brandt, and Michael Leuschel. Model Checking B Models via High-Level Code Generation. In *Proceedings ICFEM*, LNCS 13478, pages 334–351. Springer, 2022.

[232] Fabian Vu, Jannik Dunkelau, and Michael Leuschel. Validation of Reinforcement Learning Agents and Safety Shields with ProB. In *Proceedings NFM*, LNCS 14627, pages 279–297. Springer, 2024.

[233] Fabian Vu, Dominik Hansen, Philipp Körner, and Michael Leuschel. A Multi-target Code Generator for High-Level B. In *Proceedings iFM*, LNCS 11918, pages 456–473. Springer, 2019.

[234] Fabian Vu, Christopher Happe, and Michael Leuschel. Generating Domain-Specific Interactive Validation Documents. In *Proceedings FMICS*, LNCS 13487, pages 32–49. Springer, 2022.

[235] Fabian Vu, Christopher Happe, and Michael Leuschel. Generating interactive documents for domain-specific validation of formal models. *International Journal on Software Tools for Technology Transfer*, 26(2):147–168, 2024.

[236] Fabian Vu and Michael Leuschel. Validation of Formal Models by Interactive Simulation. In *Proceedings ABZ*, LNCS 14010, pages 59–69. Springer, 2023.

[237] Fabian Vu, Michael Leuschel, and Atif Mashkoor. Validation of Formal Models by Timed Probabilistic Simulation. In *Proceedings ABZ*, LNCS 12709, pages 81–96. Springer, 2021.

[238] Xiao Wang, Saasha Nair, and Matthias Althoff. Falsification-Based Robust Adversarial Reinforcement Learning. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 205–212. IEEE, 2020.

[239] J. Stanley Warford. *The MVC Design Pattern*, pages 175–199. Vieweg+Teubner Verlag, Wiesbaden, 2002.

[240] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[241] Nathaniel Watson, Steve Reeves, and Paolo Masci. Integrating User Design and Formal Models within PVSio-Web. In *Proceedings F-IDE*, EPTCS 284, pages 95–104. Open Publishing Association, 2018.

[242] Matt Webster, Michael Fisher, Neil Cameron, and Mike Jump. Formal Methods for the Certification of Autonomous Unmanned Aircraft Systems. In *Proceedings SAFECOMP 2011*, LNCS 6894, pages 228–242. Springer, 2011.

[243] Michelle Werth and Michael Leuschel. VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics. In *Proceedings ABZ*, LNCS 12071, pages 260–265. Springer, 2020.

[244] John Witulski. *A Python B Implementation - PyB A Second Tool-Chain*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2018.

[245] Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières. JeB: Safe Simulation of Event-B Models in JavaScript. In *Proceedings APSEC*, volume 1, pages 571–576. IEEE, 2013.

[246] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In *Proceedings CHARME*, LNCS 1703, pages 54–66. Springer, 1999.

[247] Tong Zhao, Ekim Yurtsever, Joel A. Paulson, and Giorgio Rizzoni. Formal Certification Methods for Automated Vehicle Safety Assessment. *IEEE Transactions on Intelligent Vehicles*, 8(1):232–249, 2022.

[248] Zhi-Hua Zhou. *Machine learning*. Springer Nature, 2021.

[249] W.M. Zuberek. Timed Petri nets definitions, properties, and applications. *Microelectronics Reliability*, 31(4):627–644, 1991.