Bridging the Implementation Gap:

Advancements in Model-Based Concurrent Program Verification

Robert Benjamin Rubbens

Bridging the Implementation Gap: Advances in Model-Based Concurrent Program Verification

Robert Benjamin Rubbens

Bridging the Implementation Gap: Advances in Model-Based Concurrent Program Verification

Dissertation

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus
prof. dr. ir. A. Veldkamp,
on account of the decision of the Doctorate Board,
to be publicly defended
on Wednesday the 15th of October 2025 at 12:45

by

Robert Benjamin Rubbens

born on the 23rd of December 1994
in Leiderdorp, the Netherlands

This dissertation has been approved by:

Promotor
prof. dr. M. Huisman

Co-promotor:
dr. P. van den Bos

IPA Dissertation Series No. 2025-14
The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Nederlandse Organisatie voor Wetenschappelijk Onderzoek
The work in this thesis was supported by the Mercedes (Maximal reliability of concurrent and distributed software) project, funded by the NWO VICI grant 639.023.710.

ISBN: 978-90-365-6909-5
DOI: 10.3990/1.9789036569101
Available online at doi.org/10.3990/1.9789036569101

Typeset with LaTeX
Printed by Ipskamp Printing
Cover design by Robert Benjamin Rubbens

written by a human
not by AI

Acknowledgements

Thank you, Marieke, for the thorough and qualitative supervision over the years. I've learnt many useful lessons from you, not just restricted to the importance of a good desk at the workplace, but also about writing, time management, PL research, and how to write unit tests in Java. I appreciate the trust you put in me while I explore a haphazard research agenda, and the nuance you employ in your speech when I push you to consider a controversial topic. I hope working with you and Andrea as a postdoc will be as productive as the PhD and master's thesis have been, and that we can do a few more projects together.

Petra, you joined the supervision a bit later down the road, but this did not stop you from thoroughly shaping my PhD. I think your insistence on being sceptical of my own perfectionism is finally starting to have an effect. I hope I can keep an eye on it myself, now that you're not checking in with me regularly anymore. I wish you best of luck with continuing your own research line. It seems to me that you have already made a very promising start at it.

Sophie, you did not supervise me for very long. However, I'm sure your keen eye and work-ethic has had its effect on my decision to pursue a PhD. I have not yet succeeded in imitating either, but there still time. Let's keep playing games and taking walks together.

VerCors team members, both past and present, thank you for the many nice conversations about a wide variety of topics, including work-, non-work, and plant related ones. I also enjoyed the occasional travel we got to do together. Pieter, you are missed in the VerCors team, but we are getting by somewhat, thanks to Alexander. I've learnt many things of you, not just from the random theoretical side-tangents you sent me on, but also on software engineering in general. Thank you for taking your time with me, and the endless deep discussions.

FMT colleagues, thank you for the daily chats about a variety of topics during lunch, e.g.: railway systems, geopolitics, star trek, books, movies and more. I also get a warm fuzzy feeling when I think of the many fun BOCOMS we've had. Y'all are just generally an awesome group. In particular, I like it when you indulge me when I want to talk about something very specific over lunch (which, in my humble opinion, I think I manage to keep to a sensible low frequency). Tom, thanks for being the insane computer science professor that lured me into academia.

My fellow DnD mates. Thank you for exploring Faerûn with me over the past few years. It's been even better than I expected. I know I haven't been the most reliable DM, but I hope that will not prevent us from continuing in some way or form in the future.

Edo, you are a good prover. Thanks for putting up with my twice-yearly I-want-to-be-a-mathematician itch and entertaining all my questions and tangents. Stefano, thanks for blowing my mind some more each time after telling you my mind was just blown by Kant, Descartes, Frege, etc. Lean on! Yanni, I really enjoyed your cooking and our regular chats about anything. I hope it all works out for you soon.

Philip, thank you for going with me on many boulder sessions. I appreciate that you don't complain when once again my agenda changes last minute. My daily world can be unpredictable at times. Our after-climb talks over drinks are always the icing on the cake. Thanks for never telling me to stop asking about DnD stuff. I don't think we'll ever try the bi-weekly boulder evening thing again, but it has been fun while it lasted :) Maybe the monthly boulder evening can work out in the near future.

Lars, Tom, Christopher, João, Wytse, Simon, Sung, Robbert, I enjoy our yearly-ish communications. I'm sure I'll hear more about your work in the coming years.

Ruben, Michiel, Willem, thank you for being such good friends. I appreciate your different perspectives on working in industry, keeping me grounded while I fly off into some academic fantasy land. Someday we'll start a company together. I'm not sure how we'll make money but this doctorate I'm getting should get us at least 30% of the way.

People of Alcatraz, the years of living with you on Calslaan 13-3 are among the best in my life. Thank you, for introducing me to amazing cinematographic wonders like Koyaanisqatsi and The Room, the many pub quizzes, and being awesome in general.

Eveline, I'm glad we stay in touch. I'm looking forward to part two!

Rick, whenever I need parenting advice I imagine what you would do. I'm looking forward to seeing Isaak play with Flynn and Frey again.

Sytze, Ron, you are my favourite idiots, the angel and demon on my left and right shoulder, respectively. I'm looking forward to our next chicken meal.

Menno, Beau, Wietse, Welmoed, Jaap, I'm still enjoying our get-togethers as much as I did in highschool. You are not family in the literal sense, but it's pretty close. Let's meet again soon.

Evita, Lisa, Thomas, Lucienne and Arnoul, and my extended family beyond. You are one of the reasons I made it this far, because I know you'll have my back if I ever need it.

Isaak, if you are anything like me, you will probably be somewhere in your twenties when you first read this. I love you. Please stop biting other kids at daycare. Look after your brother/sister/etc.

Hanna, I respect you so much, and I love you even more. The three of us are going to have a great time in the west. TNO has no idea about the powerhouse they just hired.

To all of those mentioned in this part of the thesis, be it specific or in aggregate: I hope we stay in touch.

Abstract

Software has become ubiquitous in both industry and elsewhere. Sadly, there are many examples of software bugs disrupting daily life at a large scale, such as the 2014 Heartbleed vulnerability, or the global Crowdstrike outage in 2024. Therefore, ensuring correctness of these software systems is crucial.

This is, however, hard. In a single-threaded setting, memory safety and functional correctness are difficult to verify. In a concurrent software environment, robust software design becomes even more challenging. Here, execution is interleaved between threads, causing some bugs to only be triggered in specific interleavings. This makes correct software much more difficult to write.

One way of ensuring correctness of concurrent software is through formal methods. Formal methods can check if a program follows a mathematical specification. Automated tools that implement these techniques create an opportunity for users to prove correctness of their software at unprecedented scale. This thesis focuses on the *auto-active* deductive verifier VerCors, which verifies concurrent programs with contracts for memory safety and functional correctness.

Even though there have been successful applications of formal methods to industrial systems, uptake is still limited. We found this is partly because of a gap between the mental models of engineers and the abstractions offered by formal methods. We improve the situation by combining, and extending capabilities of, formal methods with different workflows and levels of abstraction. These *hybrid* formal methods have the potential to narrow the gap between practical software development and verification with formal methods. We explore this theme in three parts.

The first part of this thesis investigates the possible reasons for the lack of adoption of one particular formal method: auto-active deductive verification. We apply the VerCors deductive verifier to an industrial code base of the Technolution company, finding two bugs. We conclude tool support still needs to be further improved, and that it is hard to connect mental models of developers to the abstraction level required for verification annotations.

In the second part of the thesis, we try to narrow the gap between mental models and formal methods by combining two verification tools: VerCors and the component-based software development framework JavaBIP. The key feature of JavaBIP is that it separates the *implementation* of components in Java from the *interaction* between components. We call this combination *Verified JavaBIP*, which verifies implementations of JavaBIP models for memory safety and functional correctness. We implement support for Verified JavaBIP in VerCors, and also implement run-time verification support in

JavaBIP. We illustrate Verified JavaBIP on the VerifyThis Long Term Verification Challenge.

In the third part of this thesis, we consider choreographies and deductive verification. Choreographies are a constrained language for describing protocols and distributed systems. In choreographies, messages are always well-typed, and endpoints never have to wait infinitely for messages. Choreographies also support code generation. To verify choreographies, VeyMont was introduced, allowing verification of all endpoints in one combined context. In this thesis, we make VeyMont more broadly applicable by extending it with shared memory and parameterization.

VeyMont initially did not support shared memory. This limited expressiveness of the choreographies, and made it impossible to use shared ghost state to prove correctness. We enable use of shared memory by adding *stratified permissions* to VeyMont, which is a new type of annotation that assigns memory annotations to endpoints. VeyMont also uses stratified permissions to preserve verification annotations during code generation, making generated code verifiable with VerCors, increasing robustness and maintainability. We verify a Tic-Tac-Toe choreography with three levels of optimization, demonstrating a trade-off between the volume of annotations required to verify the choreography and its run-time performance.

We also extend VeyMont choreographies with parameterization. Before, choreographies required the user to specify the number of endpoints upfront. This made it difficult to express distributed systems in VeyMont that naturally scale. We extend VeyMont to support choreographies with a parameterized number of endpoints. We impose modest restrictions on the syntax of choreographies to retain automation, and illustrate the extension by verifying a distributed summation choreography. This shows that despite the limitations imposed, the proposed support is sufficient to verify interesting choreographies.

To conclude, this thesis investigates the gap between formal methods and software engineering in industry. We do this by using a deductive verifier in an industrial setting, and determining what is missing in the state of the art. Moreover, we combine formal methods with different workflows and abstractions, as well as extend such hybrid formal methods, to further narrow the gap between formal methods and software design. This brings formal methods closer to industry, and hence will improve the reliability of software systems in the long term.

Contents

Acknowledgements **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
- 1.1 Software Errors . . . 1
- 1.2 Concurrency . . . 2
- 1.3 Formal Methods . . . 4
  - 1.3.1 Automation in Formal Methods . . . 5
  - 1.3.2 Top-Down vs. Bottom-Up Verification . . . 6
- 1.4 Narrowing the Gap . . . 10
- 1.5 Formal Methods in Industry . . . 11
- 1.6 Combining Top-Down and Bottom-Up . . . 13
- 1.7 Choreographic Verification . . . 14
  - 1.7.1 Shared Memory . . . 17
  - 1.7.2 Parameterization . . . 18
- 1.8 Thesis Structure . . . 19

2 Background on Program Correctness **23**
- 2.1 Program Logics . . . 24
  - 2.1.1 Deductive Logic . . . 24
  - 2.1.2 Hoare Logic . . . 26
- 2.2 Auto-Active Deductive Program Verification with VerCors . . . 32
  - 2.2.1 Verification Workflow . . . 32
  - 2.2.2 Architecture . . . 33
  - 2.2.3 Java Verification with VerCors . . . 35
- 2.3 Permission-Based Separation Logic with PVL . . . 37
  - 2.3.1 Basic Syntax . . . 38
  - 2.3.2 Permissions . . . 41
  - 2.3.3 Concurrency in PVL . . . 43
- 2.4 VeyMont . . . 45
  - 2.4.1 Simple Choreographies . . . 45
  - 2.4.2 Verification . . . 48

3 Formal Methods in Industry **55**
- 3.1 Introduction . . . 56

3.2 Tunnel System Architecture 60
3.3 Verification of the Concurrent Data Manager Module 61
3.3.1 Event Loop Analysis 63
3.3.2 Discussion on the Discovered Bugs 66
3.4 Results Presentation 67
3.4.1 Design Process 67
3.4.2 Lessons Learned 70
3.5 Future Research 72
3.6 Conclusion 73

4 Towards Verified Concurrent Systems in Java **75**
4.1 Introduction 76
4.2 Related Work 78
4.3 JavaBIP and Verification Annotations 79
4.4 Architecture of Verified JavaBIP 80
4.5 Implementation of Verified JavaBIP 80
4.6 Casino Case Study 82
4.6.1 Solidity Implementation 82
4.6.2 JavaBIP Encoding 83
4.6.3 Deductive Verification 86
4.6.4 Run-Time Verification 87
4.6.5 Fixing the Problem 88
4.7 Conclusion and Future Work 89

5 Verified Shared Memory Choreographies **91**
5.1 Introduction 91
5.2 VeyMont Workflow and Choreography Language 95
5.2.1 VeyMont Workflow 96
5.2.2 Choreography Language 96
5.2.3 New Features of VeyMont 99
5.2.4 Motivating Example 100
5.3 Choreography Verification 101
5.3.1 Permission Stratification 101
5.3.2 Encoding of Choreographic Communication 103
5.4 Endpoint Projection 104
5.4.1 Statement and Expression Projection 104
5.4.2 Channel Generation 105
5.5 Case Studies 106
5.6 Related Work 110
5.7 Conclusion 112

6 Verified Parameterized Choreographies **115**
6.1 Introduction 115
6.2 Choreographies 119
6.2.1 Syntax 119
6.2.2 Deadlock Freedom 123
6.2.3 Shared Memory 124
6.2.4 Running Example: Distributed Summation 125
6.3 Choreographic Projection 126
6.3.1 Non-Parameterized Example 127
6.3.2 Non-Parameterized Projection Rules 128
6.3.3 Parameterized Example 129
6.3.4 Parameterized Projection Rules 129
6.3.5 Branch Unanimity 131
6.3.6 Choreographic Projection of Distributed Summation 132
6.4 Endpoint Projection 133
6.4.1 Non-Parameterized Example 133
6.4.2 Non-Parameterized Projection Rules 134
6.4.3 Parameterized Projection Rules 134
6.4.4 Example Endpoint Projection 136
6.5 Related Work 136
6.6 Conclusion 139

7 Conclusion **141**
7.1 Future Work 144

Bibliography **149**

Appendix **169**

A Publications by the Author **171**

B Research Data Management **173**

Samenvatting **175**

Introduction 1

Contents

1.1 Software Errors . . . 1
1.2 Concurrency 2
1.3 Formal Methods . . 4
1.4 Narrowing the Gap . 10
1.5 Industry 11
1.6 Combining Tools . . 13
1.7 Choreographies . . . 14
1.8 Thesis Structure . . 19

1.1 Software Errors

In our daily world, to say that software is ubiquitous is an understatement.

From the logic that flies air planes to smart jewellery, software is involved in some way. While this yields many benefits, it is in fact a double-edged sword: whenever the behaviour of software is different from what was intended, things can go wrong.

Looking back in history, it is not difficult to find examples of software behaving in unexpected ways. One particularly expensive case is that of the catastrophic test flight of the Ariane 5 rocket, costing hundreds of millions of dollars [59]. While the cause of this accident can be considered through many lenses, a salient observation is that the chain of accidents started with a seemingly simple conversion of a 64-bit floating point number into a 16-bit signed integer [118].

[59]: Dowson (1997), *The Ariane 5 software failure*

[118]: Nuseibeh (1997), *Soapbox: Ariane 5: Who Dunnit?*

It is now several decades later. Sadly, there is no evidence that we have solved any of the problems that have been plaguing computer scientists since the ’60s. On the 19th of July, 2024, Crowdstrike distributed an update of their Falcon tool to their customers. This update, which changed the configuration of the tool, was not fully propagated to every part of the tool [51]. Ultimately, this inconsistency resulted in an *out-of-bounds access*, meaning memory was

[51]: CrowdStrike (2024), *External Technical Root Cause Analysis — Channel File 291* (link)

accessed outside its intended range, crashing the host machine at the kernel level. Many industries were directly affected by this outage.

This is not the first time an out-of-bounds access bug made the news. In 2014, the bug that made the Heartbleed vulnerability possible was discovered [42]. The bug hinged upon an out-of-bounds access, which effectively allowed any client to force the server to dump a random portion of its memory.

[42]: Carvalho et al. (2014), *Heartbleed 101*

These incidents took place in different domains: aerospace, security, and software engineering. Yet, their fundamental structure is not so different. They all concern requirements that are simple when considered in isolation. For example, properly converting a decimal number to an integer, or reading an element from a list. These are well understood operations; they are concepts that software engineers deal with on a daily basis. Yet, it seems they are easily missed, and this is not unexpected. In all mentioned cases, there were likely thousands of lines of code, making bugs hard to discover by manual inspection.

One could argue that we need automatic tools to check simple requirements like the ones mentioned before in large volumes of code. Unfortunately, this would not suffice, as there is another source of complexity that must be considered: concurrency.

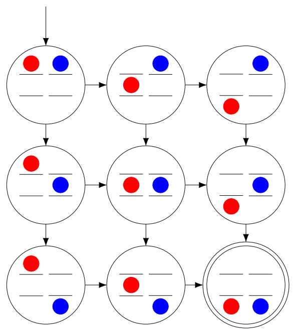

Figure 1.1: State space of a system of 2 components, red and blue. Each component executes 2 "move down" actions. Each path from the top left to the bottom right represents one possible interleaving.

1.2 Concurrency

In a concurrent system, execution steps of the system subcomponents are interleaved. This introduces the category of concurrency-related bugs, which only occur in some interleavings of the system, but not in others. Therefore, finding all bugs in a concurrent program requires considering all interleavings.

The number of interleavings one must consider grows exponentially, even for simple systems. For example, imagine a system of two components where each component needs to execute two actions, such as the state machine in Fig. 1.1. We draw the system states as black circles. Within the system states we draw the states of the two components, a red component and a blue component, side by side. Each component can take two actions in sequence: first, move its own coloured ball from the top to the middle level, and then move its ball from the middle to the bottom level. One component taking a step is visualized by an arrow connecting two system states. The initial state is the top-left circle, marked by an incoming arrow. The bottom right state is the final one, marked by a double circle. Notice how each path from the top-left state to the bottom-right state represents a possible evolution of the system, and hence a possible interleaving of the execution of the two components. As there are six possible paths in the figure, the system has six possible interleavings.

Now consider a similar system with 3 components. This already increases the number of interleavings to 90. A system with 5 components, where each component executes 4 actions, has more interleavings than the number of seconds in the average human life.

The exponential growth of interleavings can be mind bending, but the practical nature of concurrency bugs usually is not. For example, at the Pwn2Own conference in 2023, the Synacktiv team demoed a hack of the Tesla Model 3 that involved concurrency [63]. In this hack, Synactkiv exploited a time-of-check to time-of-use (TOCTOU) bug to gain control over the vehicle.

[63]: Gatlan (2023), *Windows 11, Tesla, Ubuntu, and macOS hacked at Pwn2Own 2023* (link)

A TOCTOU bug can occur whenever there is some time between the checking of a property and the subsequent use of the property, while there are other processes happening concurrently. In single-threaded systems, this is not a problem: as there is only one thread of execution, a property remains valid after it is checked, so long as the thread itself does not invalidate it. However, in a car with multiple simultaneously executing components, a TOCTOU bug can be problematic. Specifically, during the time delay between the check and the use of the property, another system component might change the shared state, invalidating the property in the process. Then, when the property is later used, it does not hold anymore, causing problems for the primary component.

In the demo by Synacktiv, they exploited the time delay between checking integrity of a firmware update, and actually applying the update. Within this time delay, Synacktiv managed to replace the firmware update with their own version. The integrity check is supposed to prevent exactly such an attack! This goes to show that, in a concurrent environment, even something as simple as checking an update before using it has to be carefully designed.

The problem of bugs seen in practice is usually not the requirements themselves, but the number of requirements and their pervasiveness. When multiplied with a modest amount of concurrency, the number of execution scenarios to consider grows far beyond what a human can comfortably reason about. Unfortunately, as long as software is written by humans, bugs will be part of the reality of software, because making mistakes is human. If we are to have any hope for producing software that is free of faults, this can only be achieved using a structured approach, leaving no requirement or interleaving to informal oversight. Such approaches are called *formal methods*.

1.3 Formal Methods

Formal methods are rigorous methodologies based on strict techniques that give guarantees, if not in an exhaustive fashion, then at least strictly delineated [64]. Formal methods differ from “informal” methods[1] in their description and application: a specific formal method prescribes a finite set of rules, which can be interpreted strictly and leave no ambiguity. In theory, this means anyone can apply a formal method correctly by “just” carefully following instructions. In practice, applying a formal method can still be challenging [64].

[64]: Gleirscher et al. (2023), *A manifesto for applicable formal methods*

1: Such as code review, pair programming, ad-hoc testing, and source code linters.

[64]: Gleirscher et al. (2023), *A manifesto for applicable formal methods*

In this thesis, we consider formal methods for the *verification* of systems. This means we consider methods that show:

> “the system does what you think it should do, and nothing else.” (Ter Beek et al., [20])

This is in contrast to e.g. writing a formal specification of a system only for documentation purposes, or using synthesis techniques to automatically generate complete finished implementations based on a specification.

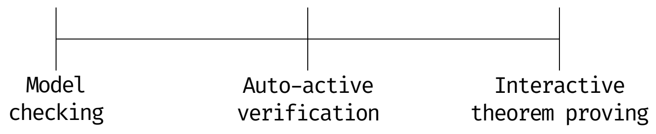

Figure 1.2: Spectrum of formal methods, from more to less automated

1.3.1 Automation in Formal Methods

Some formal methods are strict enough to be implemented as a computer program, allowing a computer to take care of most of the bookkeeping. E.g. some formal methods are based on exhaustive searches which are simply too large to complete by hand. Automating a formal method can not only lower the usage barrier for a formal method, but also opens the door to increased rigour and scalability. There are plenty of examples of formal methods that can be implemented. However, not all formal methods are equally automatic. The degree of automation is actually a spectrum, as depicted in Fig. 1.2, where some formal methods are more automatic than others. We discuss three points on this spectrum:

- a fully automatic formal method, *model checking*,
- a manual but computer-assisted formal method, *interactive theorem proving*, and
- a partially automated formal method, *auto-active verification.*

Model checking On the "automation" extreme end of the spectrum, a well-known example is model checking [20]. Given a model and a property, a model checker computes if the model satisfies the property. One possible way to do this is by explicitly computing the state space of the model, and inspecting in each state if the property indeed holds. If the state space can be computed automatically (which is usually the case), checking the model is a completely automatic process. Consider the example in Fig. 1.1. Given a model checker for "pictorial" state machines, we could ask it to check the property "the red ball is always ahead of the blue ball". Upon encountering one of the states where this is not the case, the model checker would terminate, and print the sequence of transitions that leads to this counter example state.

[20]: Ter Beek et al. (2025), *Formal Methods in Industry*

Interactive theorem proving An example on the "manual" extreme end of the spectrum are interactive theorem provers (ITPs) [20]. ITPs can check every intermediate step of a proof automatically, given that the steps are provided by the user. For example, an ITP might have the following lemma in its standard library: "given a list X, if X is sorted and non-empty, then the first element of X is less than or equal to the last element of X". This lemma can be used to prove that, for any particular list Y, the first element of Y is less than or equal to the last element, assuming Y is sorted and non-empty. However, the user will need to instruct the ITP to apply *this* particular lemma to *this* particular list Y. As the user needs to provide such intermediate steps, interactive theorem provers are not automatic in the general case. However, research that seeks to automate parts of this process is currently ongoing and fruitful [54, 94, 125].

[20]: Ter Beek et al. (2025), *Formal Methods in Industry*

[54]: Czajka et al. (2018), *Hammer for Coq: Automation for Dependent Type Theory*
[94]: Limperg et al. (2023), *Aesop: White-Box Best-First Proof Search for Lean*
[125]: Paulson et al. (2010), *Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers*

Auto-active verification In this thesis, we consider a technique in the middle of the automation spectrum: *auto-active verification* [93]. Auto-active approaches require the user to provide some guidance in the form of program annotations. Once these annotations are written, the auto-active verifier takes over and requires no further intervention from the user. In a nutshell, auto-active verifiers are automatic, but also require some interaction upfront. While auto-active verification sounds similar to using an ITP, there is a subtle difference: in an ITP the user indicates when a proof should be finished using automation. In contrast, an auto-active verifier, which uses the program annotations to guide the proof, typically[2] does not allow the user to intervene during the actual proving process.

2: Of course, there are exceptions. For example, Verifast [75] allows the user to inspect intermediate states of verification. Viper [108] goes even further and allows the user to query and manipulate the intermediate verification state when verification fails [83]. As these facilities are intended for debugging, they are not typically used for large-scale verification.

1.3.2 Top-Down vs. Bottom-Up Verification

Another way to categorize formal methods is the *direction* in which they verify systems. On this spectrum, the bottom represents implementations, which can easily be executed, but are hard to reason about. The top represents abstract specifications, which might not be executable or lack details, but which are easier to analyse. The two extremes on this spectrum are *bottom-up* and *top-down* formal methods.

Bottom-up In bottom-up formal methods [24], the implementation is the starting point of the verification effort. The goal is to show correctness of an implementation w.r.t. a specification. By first writing the implementation and adding the specification on top of it, the specification becomes a second-class citizen. In other words, bugs found in the implementation will need to be fixed, and thus might require changing the implementation, but the basic approach is that the specification is written to describe the implementation *as-is*.

[24]: Bílý et al. (2023), *Refinement Proofs in Rust Using Ghost Locks*

The implementation being the central focus of the verification effort is particularly common in the context of verification of existing systems, e.g. as done by Hiep et al. in [69]. In this work, the intention was to verify the `LinkedList` implementation from the OpenJDK using the KeY verifier [4]. KeY is a good example of a bottom-up verifier: it is a program verification tool that requires the user to provide both a Java program and annotations in the form of *specification comments*. KeY can then check if the Java program respects the given annotations, possibly with interactive guidance from the user.

[69]: Hiep et al. (2020), *Verifying OpenJDK's LinkedList using KeY*

[4]: Ahrendt et al. (2016), *Deductive Software Verification - The KeY Book - From Theory to Practice*

The "bottom-up" categorization is not absolute. For example, model checkers also fit the description of a bottom-up formal method. Like any typical bottom-up formal method, model checkers verify if a given model M satisfies some property ϕ, both to be provided by the user [15]. However, the idea that the implementation is the central focus of the verification effort is less clear. For example, model checkers such as mCRL2 [39] or NuSMV [46] accept implementations in the form of an abstract state machine. While abstract state machines can be considered executable, in practice they are often abstractions of software, or maybe even of physical systems. In this way, as both the model and the property to be satisfied are partial descriptions of a system, it is not clear which one needs to change when a property ϕ does not hold for model M. This makes it less clear that model checkers are a bottom-up formal method. However, as model checkers require both the model and the property to be provided by the user, we categorize them as bottom-up.

[15]: Baier et al. (2008), *Principles of model checking*

[39]: Bunte et al. (2019), *The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability*

[46]: Cimatti et al. (2002), *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*

Bottom-up formal methods are also referred to as *a posteriori* [20] formal methods, or *post-hoc* verification [156].

[20]: Ter Beek et al. (2025), *Formal Methods in Industry*

[156]: Watson et al. (2016), *Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience?*

More broadly, bottom-up verification can also be viewed through the lens of the megamodel of program verification tools, as introduced by Lathouwers and Zaytsev [92]. This megamodel defines a hierarchy, with levels **PV0** to **PV6**, where each level contains tools that reach some degree of conceptual complexity. For example, **PV0** contains all modelling frameworks that do not have some kind of checker or verifier, **PV1** is the category of tools that can check models based on direct syntactic analysis (so-called *linters*), and so on. At the top of the hierarchy is **PV6**, which is the category of proof assistants.

[92]: Lathouwers et al. (2022), *Modelling program verification tools for software engineers*

In the megamodel, bottom-up tools typically reside in the **PV3** level. The distinction between bottom-up and **PV3** is that we consider bottom-up tools to be implementation-centric, whereas tools in the **PV3** level only need to 1. support annotating inputs with properties, and 2. be conceptually decomposable into a specification provider and a verifier. A counterexample of a bottom-up tool that is not in **PV3** is KeY. This is because KeY is typically used interactively, which causes it to be located in **PV6**.[3]

3: `https://slebok.github.io/proverb/key.html`

Top-down On the other end of the spectrum there are *top-down* formal methods, where the abstract specification is the ground truth. Through successive derivation steps a conforming implementation is *derived*. How many derivation steps take place, and what each step entails, depends on the specific instance of top-down method.

For example, in the CorC tool [141], the initial specification is a pre- and postcondition that the program should satisfy. Here, a precondition is a condition that may be assumed to hold before the program is executed. The postcondition is a condition that should hold when the program finished. Then, CorC provides support to iteratively construct an imperative program S. Each step refines S and maintains the relation that given the precondition, after executing S, the postcondition holds. For example, when the precondition implies the postcondition, CorC allows replacing S with the *skip* statement [141].

[141]: Runge et al. (2019), *Tool Support for Correctness-by-Construction*

Another example of a top-down formal method is Event-B [2], where the initial specification is an abstract machine with local state and transitions that modify local state. Here,

[2]: Abrial (2010), *Modeling in Event-B - System and Software Engineering*

a refinement step consists of defining a more concrete machine, and then showing that the concrete machine does not contradict the abstract machine. Refinement steps can be applied until the model is so concrete such that implementation is trivial.

Top-down formal methods are also referred to as *a priori* verification [20], *program derivation* [58, 81] or *correct-by-construction* [86]. In the megamodel of program verification tools, top-down formal methods that *only* generate an implementation reside in the **PV2** level, which is the level for synthesis tools. Top-down formal methods that also generate checkable specifications belong to a higher level.[4] This means the category of top-down formal methods is larger than just the **PV2** level.

[20]: Ter Beek et al. (2025), *Formal Methods in Industry*

[58]: Dijkstra (1975), *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*

[81]: Kaldewaij (1990), *Programming - the derivation of algorithms*

[86]: Kourie et al. (2012), *The Correctness-by-Construction Approach to Programming*

4: Lathouwers and Zaytsev also point out this nuance of the **PV2** level in general.

Implementation gap When a top-down formal method produces a final certified implementation, this does not necessarily mean that the job of the user is done. Some top-down formal methods create implementations that are not directly usable for execution, e.g. the implementation generated could be in a non-executable language. Another reason could be that the implementation still contains dummy implementations that the user is expected to fill in themselves. In both of these cases, manual *unchecked* implementation is necessary to get a fully functioning implementation.

In the context of model checking, where the models often informally represent a physical system, this is a well-known problem ([119], p. 298). In essence, it is challenging to check if a model represents the relevant behaviour of some other, possibly physical, system. O'Regan does not explicitly name the problem, but for the sake of clarity we refer to it as the *implementation gap*. Previous work calls this the *abstraction problem* (Oortwijn, [122], p. 12).

[119]: O'Regan (2023), *Mathematical Foundations of Software Engineering - A Practical Guide to Essentials*

[122]: Oortwijn (2019), *Deductive techniques for model-based concurrency verification*

Generally, top-down formal methods intended for architectural design suffer from this shortcoming. For example, initially the Event-B toolset only generated skeleton implementations that, in combination with a suitable runtime, would faithfully execute the model. However, these skeleton implementations would still have empty stubs in places of primitive actions, meaning they were not executable yet.

Later work by Rivera et al. resolved this by generating verifiable Java code, closing the implementation gap [129].

[129]: Rivera et al. (2017), *Code generation for Event-B*

Other top-down formal methods derive an implementation all the way to an executable level. For example, Lammich et al. implement refinement from a functional language to a subset of LLVM [88]. Because the final implementation in LLVM is executable, the implementation gap is avoided. Similarly, the CorC tool refines down to Java code that can be automatically verified with KeY, as CorC also provides the necessary proof steps [141].

[88]: Lammich (2019), *Generating Verified LLVM from Isabelle/HOL*

Bottom-up formal methods are less susceptible to an implementation gap, as they usually analyze an implementation directly. Consider the Java verifier KeY, which verifies Java programs. In this case, verification results apply directly to the given implementations. In contrast, model checkers are a counterexample. This is because model checkers often analyse abstract models, such as labelled state machines or finite-state machines, that are intended to describe some concrete physical system. Here, there is an implicit assumption that the physical system is accurately described by the abstract model. In those cases, model checkers risk falling into the implementation gap.

Interestingly, because some bottom-up formal methods do not have an implementation gap, this makes them one of the ways that top-down formal methods can use to ensure the final implementation is and remains correct. As mentioned earlier, this is what as is done for Event-B and CorC. Essentially, if top-down formal methods generate the right annotations in the final implementation, integrating bottom-up formal methods is possible.

1.4 Narrowing the Gap

While formal methods research and innovation is ongoing, their usage in practice is limited. This shows that there is a gap between industrial application and theoretical design of formal methods. Therefore, in this thesis, we investigate the following research question:

How to narrow the gap between mental models and formal methods?

One step towards narrowing the gap between mental models and software has already been made in the form of *software diagrams*. Examples of such diagram frameworks are UML and SysML [62, 140]. Developers use the diagrams from these frameworks, such as class diagrams, activity diagrams, and other kinds of augmented stated machines, to make the mental models of the software they are developing concrete. While these diagrams are widespread in industry, there is no standard formal method that takes these diagrams as a starting point.

[62]: Friedenthal et al. (2015), *A Practical Guide to SysML*
[140]: Rumpe (2016), *Modeling with UML*

Therefore, we present one possible answer to the research question, formulated by considering and combining top-down and bottom-up formal methods into new *hybrid* formal methods. We believe that hybrid formal methods have the power to contribute to the design phase, the verification phase and the maintenance phase of software.

In particular, we will consider the following formal methods:

- VerCors [13], a verifier for concurrent programs, introduced in Section 1.5,
- JavaBIP [30], a component-based software development framework, introduced in Section 1.6, and
- VeyMont [36], a VerCors-based verifier for choreographies, introduced in Section 1.7.

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*
[30]: Bliudze et al. (2017), *Exogenous coordination of concurrent software components with JavaBIP*
[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

In the next sections we will introduce the core technical chapters of this thesis in order of appearance, followed by an outline of the contributions of this thesis.

1.5 Formal Methods in Industry

It is clear that formal methods can contribute substantially to the reliability of software. Indeed, there are several success stories of formal methods in industry already.[5] The Infer static analysis tool is integrated in the software development pipeline of Facebook [40]. At Amazon, TLA$^+$ [89]

5: For a recent and comprehensive overview, see Ter Beek et al. [20].

[40]: Calcagno et al. (2015), *Moving Fast with Software Verification*
[89]: Lamport (2002), *Specifying Systems, The TLA+ Language and Tools for Hardware and Software*

is routinely used for verification projects [111]. The Ironclad project at Microsoft has also resulted in several novel tools, as well as verified non-trivial software [68, 97].

[68]: Hawblitzel et al. (2015), *IronFleet: proving practical distributed systems correct*
[97]: Lorch et al. (2020), *Armada: low-effort verification of high-performance concurrent programs*

[111]: Newc
Amazon Cho

However, despite successes, the prevailing opinion in both academia and industry is still that uptake of formal methods outside of academia is minimal [64, 152]. In an attempt to stimulate further adoption of formal methods in industry, as well as learn about the reasons why formal methods are not used more widely, in Chapter 3 we discuss a case study with the VerCors verification tool at the Technolution software development company.

[64]: Gleirscher et al. (2023), *A manifesto for applicable formal methods*
[152]: The White House (2024), *Back to the Building Blocks: A Path Toward Secure and Measurable Software*

VerCors VerCors [13] is an auto-active deductive program verifier for verification of concurrent programs. It allows users to annotate their programs using *pre- and postconditions*, and can automatically check if the programs respect the annotations. It uses *permission-based separation logic* [66] (PBSL) for ensuring memory safety and preventing concurrency errors such as data races. Using PBSL, it also supports various concurrency constructs, such as lock invariants for verifying programs with locks. Various languages are supported, such as Java, C, OpenCL, CUDA, or the custom prototyping language Prototypal Verification Language (PVL). For the case study of Chapter 3, we only used the Java frontend.

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

[66]: Haack et al. (2015), *Permission-Based Separation Logic for Multithreaded Java Programs*

Case study One of the problems uncovered in the case study of Chapter 3 concerns a concurrency bug. Specifically, an object field was accessed concurrently while not protected by a lock. We show that the bug could have been caught by VerCors. After completion of the case study, we communicated the results to the engineers at Technolution. We took special care to prepare a presentation for an audience of non-formal-methods experts, by chunking up information as much as possible and omitting details.

Among several lessons learned, one that stood out was that there is a barrier for engineers to write verification annotations in their code. This is a barrier of tooling, but also of culture. In addition, the engineers prefer tools that are close to the mental models they already have about the code. Finally, the tools being as automatic as possible is also a high priority.

1.6 Combining Top-Down and Bottom-Up

As stated previously, both top-down and bottom-up formal methods have their strengths and weaknesses. On the one hand, bottom-up verification is effective at the level of concrete software, which was an important aspect of the case study. On the other hand, top-down verification is effective at starting with a mental model, and refining towards a low-level implementation in steps. The case study with Technolution confirmed the intuition that a formal method combining both top-down and bottom-up approaches is necessary. This is further explored in Chapter 4: what hybrid formal method would have been appropriate to use in the case study of Chapter 3?

The case study being written in Java made VerCors the logical choice as the bottom-up part of this hybrid formal method. It has a capable Java frontend, and its design optimized for prototyping makes implementing an augmented Java frontend tractable. In addition, support for separation logic makes it possible to consider verification of concurrent software, instead of just sequential software.

JavaBIP For the purpose of a hybrid formal method for Java, the ideal counterpart to VerCors is JavaBIP. It is a top-down formal method, and a member of the BIP [17] and Reo [10] family of formal methods. BIP, JavaBIP and Reo are *exogenous* component-based software frameworks, where software is built by composing components. “Exogenous” implies that the interaction between these components is specified separately from the behaviour of the components. This is in contrast to defining software with interaction defined *endogenously*, that is, as part of the component implementation.

[17]: Basu et al. (2006), *Modeling Heterogeneous Real-time Components in BIP*

[10]: Arbab (2004), *Reo: A channel-based coordination model for component composition*

Like BIP, a JavaBIP model consists of a collection of components, and separately the possible interactions. A runtime system orchestrates these interactions, allowing the components to focus on implementation details. Unlike BIP, JavaBIP supports component implementation and interaction specification within Java. This lowers the barrier of entry and increases the understandability, as it makes the framework as a whole more concrete.

Tool combination Integrating an auto-active deductive program verifier like VerCors into JavaBIP gives model designers the capability of adding functional correctness properties to their JavaBIP models. This eliminates the implementation gap, as it enables enforcing correctness properties over component implementations. Specifically, we first extend JavaBIP models with syntax for specifying pre- and postconditions of transitions, as well as invariants for components and component states.

We then extend VerCors to verify such extended models against an implementation. This ensures that the implementation of the JavaBIP model does not only follow prescribed behaviour, but also respects the functional correctness properties.

While the approach was effective, there were still several shortcomings. First, there was no support for data sharing between components of the program model. This made the tool less generally applicable. Second, while JavaBIP supports models where the number of components depends on a run-time parameter, VerCors requires upfront specification of the number of components in the model. Verifying parameterized systems without concrete bounds is a difficult open research problem [109].

[109]: Neele (2020), *Reductions for parity games and model checking*

1.7 Choreographic Verification

Solving the issues of data sharing and parameterization in a hybrid formal methods requires a shift in foundations. In particular, for the last two chapters of this thesis we combine VerCors with a different top-down formal method: *choreographies*.

Choreographies Simple choreographies are defined as a set of participants, which we call “endpoints”, as well as a list of message exchanges between endpoints [105]. Simple choreographies are restrictive in the sense that they only model message exchanges between endpoints, and nothing else. However, this restriction gives simple choreographies two key properties. The first is *deadlock freedom*, meaning that endpoints that faithfully execute a choreography will not get stuck waiting for a message that will never be sent. The second is *message fidelity*, meaning that a participant only receives messages of the type it is expecting.

[105]: Montesi (2023), *Introduction to Choreographies*

An example of a simple choreography is in Fig. 1.3. It mentions three endpoints: Alex, Bob and Charlie. The choreography specifies two message exchanges, separated by a semicolon: first Alex sends a message to Bob, and then Bob sends a message to Charlie. Note that, by specifying every communication in terms of a sender *and* a receiver, it is impossible to write a deadlocking choreography.

$$\mathsf{Ex} \equiv$$
$$\mathsf{Alex} \to \mathsf{Bob}; \mathsf{Bob} \to \mathsf{Charlie}$$

Figure 1.3: An example of the simple choreography called Ex

Simple choreographies, e.g. as defined by Montesi, are abstract protocols, meaning that they do not constrain how the endpoints of the choreography process the messages [105]. A related aspect of choreographies is that they allow generating an implementation for each endpoint that participates in a choreography. We call this operation the *endpoint projection*, denoted with the operator $[\![\cdot]\!]_r$, which takes as an input a choreography and an endpoint r for which an implementation must be generated. Essentially, the endpoint projection takes only the part of the choreography that is relevant for the endpoint r. An example of this is in Fig. 1.4, where you see an implementation of the choreography in Fig. 1.3, generated in such a way that the implementation executes exactly the portion of the choreography Bob is responsible for. The central correctness property of the endpoint projection is *deadlock freedom of endpoint projections*: if the endpoint projections of all participants of a choreography are executed in parallel, no deadlock will occur. We define simple choreographies more precisely in Chapter 2.

$$[\![\mathsf{Ex}]\!]_{\mathsf{Bob}} \equiv \begin{array}{l}\texttt{receive Alex;}\\ \texttt{send Charlie;}\end{array}$$

Figure 1.4: An implementation for Bob’s part of the choreography from Fig. 1.3

Historically, choreographies originate from the field of Service-Oriented Architecture, which concerns the building of services by composing other services [95]. In this context, a

[95]: Liu et al. (2018), *Encyclopedia of Database Systems, Second Edition*

choreography is defined as a system where no single entity controls all other components of the system. Paraphrasing Liu and Özsu, choreography implies distributed control similar to how “dancers dance following a global scenario without a single point of control” [95]. In contrast to choreography, there is also orchestration, where a system is designed from the point of view of one service that controls the others.

VeyMont The generated implementation in Fig. 1.4 does not specify what must be done with the message received from Alex, nor what message should be sent to Charlie. This is another example of the implementation gap: an implementation for a bank module that sends a random number instead of the actual balance deducted from the account is clearly problematic.

Van den Bos and Jongmans have tackled this implementation gap by integrating choreographies with auto-active deductive verification [36, 80]. The result of this is the VeyMont tool, a deductive verifier and code generation tool for choreographies. VeyMont allows users to annotate choreographies with pre- and postconditions, which VeyMont can then automatically verify for correctness. That way, concerns such as a bank withdrawal can be specified in the choreography. VeyMont is built on top of VerCors, an auto-active deductive verifier for concurrent programs [13]. This allows VeyMont to naturally support checking of memory safety.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*
[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

VeyMont choreographies are more general than simple choreographies. For example, endpoints in VeyMont choreographies can do local actions on local state. VeyMont choreographies also have input parameters, and may contain while loops with loop conditions that involve multiple endpoints. Because VeyMont supports separation logic through VerCors, local actions can locally safely use concurrency, without affecting analysis of the choreography.

However, similar to Verified JavaBIP, the choreographies of VeyMont can still be further extended. For example, endpoints in a VeyMont choreography are not allowed to share data structures. VeyMont also requires choreographies to specify the number of endpoints upfront. Another limitation is that the endpoint projection does not include annotations in the projection. This means that, even though VeyMont verified the choreography, the code generated using the endpoint projection is not immediately verifiable with VerCors. Instead, the code must be manually annotated before it can be verified again. In chapters Chapters 5 and 6, we resolve these constraints, allowing verification of choreographies with shared memory and parameterization.

1.7.1 Shared Memory

In Chapter 5 we describe the stratified permissions approach to extend choreographies with shared memory. Essentially, stratified permissions is another layer of annotations that allows the users to specify which endpoint owns what memory. This has three benefits: first, it allows expressing and verifying choreographies which use intricate memory sharing strategies for optimization and efficiency. This means VeyMont becomes more broadly applicable to a broader class of choreographies.

Second, it allows so-called *ghost code* to use shared memory as well. Ghost code is code that a verifier needs to complete the proof, but which does not influence run-time execution of the program. While it does not matter for expressive power of the tool, it does make some proof steps easier to state. When ghost code uses shared memory, this memory is called *ghost state*. This is state that is used for program verification, but which also does not influence run-time execution. In other words, ghost state can only be mutated using ghost code.

Third and finally, through stratified permissions, the endpoint projection has enough information to determine which endpoint an expression is related to. This allows the endpoint projection to include annotations in the endpoint projection. In most cases, if the choreography verifies with VeyMont, the endpoint projection of a choreography can be verified with VerCors, too. This is beneficial for maintenance purposes, because the endpoint projection of a choreography can be re-verified, separately from VeyMont, after manually modifying the choreography. In addition, by separately verifying the output of VeyMont with VerCors, detection of bugs and hence robustness is increased.

VanDenBos2023

To illustrate the new support for shared memory, we apply VeyMont to three choreographic versions of Tic-Tac-Toe. This example originates from earlier work [**VanDenBos2023**], yet is still relevant because the correctness proof given in earlier work does not apply in the context of shared memory choreographies. In addition, each version shows a different trade-off between code performance and the amount of ghost code required. Essentially, the more efficient version of Tic-Tac-Toe can also be verified, in exchange for having to write more annotations with ghost code.

1.7.2 Parameterization

Originally, VeyMont choreographies were already partially parameterized. For example, a VeyMont choreography can have a parameter `X` which endpoint `A` will send to endpoint `B`. However, the number of endpoints participating in a VeyMont choreography always had to be specified upfront.

In Chapter 6, we extend choreographies with parameterization, allowing verification of scalable choreographies. We achieve this by extending VeyMont choreographies with two parameterized primitives. The first is *endpoint families*, which declare a family of endpoints, the size of which may depend on parameters of the choreography. The second is *parameterized communication*, which is a statement that specifies communication between two ranges of endpoint families. This extension only supports one-to-one communication; this requirement is checked automatically.

We then extend the choreographic verification algorithm of VeyMont to support these primitives, e.g. by leveraging primitives for structured parallelism from VerCors to encode the parameterized communication statement. We also adapt the endpoint projection to support parameterized choreographies. In essence, the projection must be overapproximating so that the resulting implementation can execute the choreography for any member of an endpoint family.

To show the effectiveness of the extension, we manually apply the approach to a distributed summation choreography formulated for a ring network. We show that each endpoint of the choreography computes the same (correct) sum.

1.8 Thesis Structure

The main parts of this thesis are summarised below.

I. Formal Methods in Industry
In Chapter 3, we investigate the applicability of the VerCors deductive verifier on an industrial code base. We find two bugs in the process. We document our efforts on communicating these technical results to a non-formal-methods audience, by chunking up information and omitting details as much as possible. We also report on the response of the audience: there is a significant technical and cultural gap to start writing annotations, as well as a high preference for automated methods.

This chapter is based on the following publication:

- Raúl E. Monti, Robert Rubbens, and Marieke Huisman. "On Deductive Verification of an Industrial Concurrent Software Component with VerCors". See [106].

II. Towards Verified Concurrent Systems in Java
In Chapter 4 we combine the JavaBIP framework with VerCors, yielding the Verified JavaBIP toolset. This allows specifying components *and* invariants at the design level with JavaBIP, while simultaneously verifying if the system implementation complies with the system specification with VerCors. For illustration we apply Verified JavaBIP to the VerifyThis Casino challenge [6], showing how it can catch a hypothetical bug.

[6]: Ahrendt et al. (2023), *The VerifyThis Collaborative Long-Term Challenge Series*

This chapter is based on the following publication and artifact:

- Simon Bliudze, Petra Van den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. "JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java". See [27].
- Simon Bliudze, Petra Van den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. *Artefact of: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java*. See [26].

III. Verified Shared Memory Choreographies
In Chapter 5 we extend VeyMont with support for choreographies with shared memory. This allows expressing a richer class of choreographies. Besides allowing verification of choreographies with efficient implementations, this also enables correctness proofs that require ghost state. We apply VeyMont to three variations of the earlier-introduced Tic-Tac-Toe case study, showing that there is a trade-off between performance and the amount of annotations required for verifying correctness.

This chapter is based on the following publication and artifact:

- Robert Rubbens, Petra Van den Bos, and Marieke Huisman. "VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory". See [133].
- Robert Rubbens, Petra Van den Bos, and Marieke Huisman. *Artifact of: VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*. See [131].

IV. Verified Parameterized Choreographies
In Chapter 6, we further extend VeyMont to support choreographies with a parameterized number of participants. This feature is crucial for specifying choreographies that scale naturally in a parameter N. Examples of such choreographies are protocols for communicating in a ring architecture, or distributed databases where increasing the number of nodes also increases redundancy and hence robustness of the network. We illustrate the proposed technique by discussing a manual encoding of the distributed summation protocol for ring networks.

This chapter is based on the following publication, technical report and artifact:

- Robert Rubbens, Petra Van den Bos, and Marieke Huisman. "Verified Parameterized Choreographies". See [135].
- Robert Rubbens, Petra Van den Bos, and Marieke Huisman. *Verified Parameterized Choreographies Technical Report*. See [136].
- Robert Rubbens, Petra Van den Bos, and Marieke Huisman. *Artefact of: Verified Parameterized Choreographies*. See [134].

Background on Program Correctness 2

Contents

2.1 Program Logics . . **24**
2.1.1 Deductive Logic . . 24
2.1.2 Hoare Logic 26
2.2 VerCors **32**
2.2.1 Workflow 32
2.2.2 Architecture 33
2.2.3 Java 35
2.3 PVL **37**
2.3.1 Basic Syntax 38
2.3.2 Permissions 41
2.3.3 Concurrency 43
2.4 VeyMont **45**
2.4.1 Choreographies . . 45
2.4.2 Verification 48

In this thesis, we focus on a number of techniques for ensuring correctness of programs: auto-active deductive program verification and choreographies. In particular, we focus on two languages in the context of program verification:

- Java, because of its prevalence in industry, and
- Prototypical Verification Language (PVL), the internal prototyping and verification language of VerCors.

We first introduce basic concepts in program correctness: *deductive logic*, *inference rules* and *Hoare logic* in Section 2.1. We show how these foundations are defined, and go through several examples to illustrate how they can be used in practice.

We then introduce *auto-active deductive program verification* in Section 2.2, showing how these foundations are realized in practical techniques to verify correctness of object-oriented programs. In particular, this introduction is done in the context of VerCors [13]. We give an example of using VerCors in the context of Java, and also discuss the interface of VerCors.

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

The speciality of VerCors is analysis of programs with concurrency and shared memory. For this, it uses *permission-based separation logic* (PBSL). We introduce PBSL and PVL in Section 2.3.

This thesis also focuses on *choreographies* [105], a correctness by construction (CbC) technique. Following previous

[105]: Montesi (2023), *Introduction to Choreographies*

work, choreographies have been extended with local state and verification annotations [36, 80]. We introduce simple choreographies, as well as the essence of the work by Van den Bos and Jongmans in Section 2.4.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*
[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

2.1 Program Logics

Program logics are logics that allow reasoning about the correctness of programs, for some definition of correctness and some definition of programs. This section gives a condensed summary by considering two particular building blocks of program logics: *deductive logic*, and in particular, *Hoare logic*.

First, we introduce inference rules as a notation for deductive logic. Then we introduce Hoare logic, which is designed for reasoning about program correctness in terms of pre- and postconditions.

As the introduction done in this section is intentionally minimalistic, we recommend the following works for more thorough discussions of the material. The textbook by Ben-Ari introduces many aspects of logic, including program correctness [22]. The book by Nipkow and Klein gives a more practical introduction, oriented around the Isabelle theorem prover [117]. Lastly, the recent book by Montesi gives a good general introduction to inference rules, before introducing choreographies using that framework [105].

[22]: Ben-Ari (2012), *Mathematical Logic for Computer Science, 3rd Edition*
[117]: Nipkow et al. (2002), *Isabelle/HOL - A Proof Assistant for Higher-Order Logic* (link)
[105]: Montesi (2023), *Introduction to Choreographies*

2.1.1 Deductive Logic

A deductive logic defines the rules for what facts may be assumed, and how to construct new true facts from those assumed facts. Deductive logics are a theoretical framework more than a practical one; while it is possible to use deductive logic manually on paper to prove correctness of, e.g., programs, this is tedious and time consuming. In practice, deductive logics are used to show *meta-theorems*, that is, to prove that the core principle of a verification technique does what is intended. In this thesis, we use the a syntactical approach strongly inspired by the tree notation of Gentzen [22].

Inference rules We define a deductive logic as a set of inference rules. An inference rule shows what assumptions are necessary to reach a particular conclusion. We use notation as shown in Fig. 2.1, where the assumptions $h_0, \dots, h_n$, called hypotheses, are separated using a horizontal line from the *conclusion*. The intended meaning of an inference rule is that the conclusion *conclusion* holds if you can also prove that its assumptions $h_0, \dots, h_n$ hold. In practice, this is achieved by recursively applying inference rules until there are no hypotheses left to prove.

Inference rules with zero hypotheses are called *axioms*. These are inference rules that can always be applied, and form the logical foundation of the deductive system.

$$\textsc{RuleName}\quad \frac{h_0, h_1, \dots, h_n}{\textit{conclusion}}$$

Figure 2.1: Inference rule notation

Summarizing, given these definitions of inference rules and axioms, proving a fact boils down to providing a finite sequence of rules to be applied to the conclusion. If there are no leftover hypotheses after going through the sequence, the fact can be considered "true".

Example We now give an example by introducing a tiny deductive system to prove the following fact:

My bike will become wet at **12:00**.

In this deductive system, there are two axioms we can use. These are AxPark and AxWeather, defined as follows:

$$\textsc{AxPark}\quad \frac{}{\textbf{My bike}\text{ is outside at time }\textbf{12:00}.} \qquad \textsc{AxWeather}\quad \frac{}{\text{Rain at time }\textbf{12:00}.}$$

The axiom AxPark indicates you may assume something about the location of the bike at a certain time. The axiom AxWeather indicates you may assume something about the time at which it will rain.

From an intuitive point of view, it is already clear that the goal we set out to prove must be true. However, to formally show it in a deductive system, the following inference rule is necessary to connect the goal to the aforementioned axioms:

$$\textsc{InTheRain}\quad \frac{X\text{ is outside at time }t. \qquad \text{Rain at time }t.}{X\text{ will become wet at time }t.}$$

The rule $\textsc{InTheRain}$ is schematic: you may choose to plug in any object X and any time t into the rule. This yields a new rule, *specialized* for the values provided. Essentially, the rule $\textsc{InTheRain}$ encodes a *transitive* property: because the rain and the object X share a time (in particular, t) and a location (in particular, outside), the property of being wet is transferred from the rain to the object X.

We now have all the parts necessary to complete the proof, yielding the following proof tree, which should be read from bottom to top:

$$\dfrac{\textsc{AxPark}\ \dfrac{}{\textbf{My bike}\text{ is outside at time }\textbf{12:00}.} \qquad \dfrac{}{\text{Rain at time }\textbf{12:00}.}\ \textsc{AxWeather}}{\textbf{My bike}\text{ will become wet at }\textbf{12:00}.}\ \textsc{InTheRain}$$

Essentially, we fill in the blanks in rule $\textsc{InTheRain}$, with X = **my bike** and t = **12:00**, and apply it to the goal. Going bottom-up, this yields two hypotheses: "**My bike** is outside at time **12:00**", and "Rain at time **12:00**". These are proved directly using axioms $\textsc{AxPark}$ and $\textsc{AxWeather}$, finishing the proof.

2.1.2 Hoare Logic

Hoare logic is a deductive logic for reasoning about the correctness of programs. It was introduced by Hoare in "An Axiomatic Basis for Computer Programming" [71], and has since been a foundation of many program logics. The essence of Hoare logic is that it, for example, allows proving statements such as "The program `x = x + 1; x = x + 1;` increments `x` by `2`." More importantly, Hoare logic is designed in such a way that, if a program does *not* increment `x` by `2`, it is also *not* provable in Hoare logic. When a logic has this property, we call it "sound", meaning, whenever you can prove a fact in the logic, the fact is also indeed true from a semantic point of view.

Syntax A key part of Hoare logic is the *Hoare triple*, which is a ternary predicate typically written using the following notation[1]:

1: Hoare originally used the syntax

$$P\,\{\,c\,\}\,Q,$$

with brackets around the program. To the best of our knowledge, the style with braces around the assertions is more

$$\{ P \} \, c \, \{ Q \}$$

Here, P is called the *precondition*, and Q the *postcondition*. They are expressions over the program state of type boolean. For assertions P and Q we assume a syntax with logical atoms and operators such as $\wedge$, $\vee$, $\neg$, and *true* and *false*.

For programs, referred to symbolically with c and d, we use the following syntax[2]:

2: We omit a looping statement and an assignment statement from the syntax for ease of presentation. They are, however, typically included; see [116] for a practical example.

$$c, d \ ::= \mathbf{skip} \mid c; d \mid \text{if } b \text{ then } c \text{ else } d$$

Programs are referred to symbolically with letters c and d. The **skip** statement does nothing. The sequential composition statement first executes c, then d. The if statement first evaluates the boolean-typed expression b. If this results in *true*, c is executed. Otherwise, d is executed. For boolean expressions b, we assume a standard syntax with integer and boolean literals, and appropriate unary and binary operators.

The meaning of a Hoare triple is as follows:

$$\{ P \} \, c \, \{ Q \} \iff \begin{cases} \text{If the program } c \text{ is started in a state } s \text{ s.t.} \\ s \text{ satisfies } P\text{, then the state } s' \text{ that results} \\ \text{from executing } c \text{ will satisfy } Q. \end{cases}$$

In other words, $\{ P \} \, c \, \{ Q \}$ holds if, when P holds initially, Q holds after running c. This is called *partial correctness*, which is the type of correctness that is relevant for this thesis. Alternatively, there is also *total correctness*, which is written using a Hoare triple with square brackets:

$$[P] \, c \, [Q] \iff \begin{cases} \text{If the program } c \text{ is started in a state } s \\ \text{s.t. } s \text{ satisfies } P\text{, then the program } c \textit{ will} \\ \text{terminate in a } \textit{finite number of steps}\text{, and} \\ \text{the resulting state } s' \text{ will satisfy } Q. \end{cases}$$

This type of correctness is not discussed further in this thesis. Note that we do not define what it means for a program c to be executed in a state s and to result in a state s'. This can be done by defining a program semantics, i.e. by formally defining how programs compute output states given input states. This is out of scope for this condensed summary; we refer the reader to [116] for a practical introduction.

[116]: Nipkow et al. (2014), *Concrete Semantics - With Isabelle/HOL*

Inference rules To manipulate and prove correctness of Hoare triples, we define an axiom or inference rule for each type of statement:

$$\textsc{HSkip}\quad \{\,P\,\}\ \mathbf{skip}\ \{\,P\,\} \qquad\qquad \textsc{HSeq}\quad \frac{\{\,P\,\}\ c\ \{\,Q\,\} \qquad \{\,Q\,\}\ d\ \{\,R\,\}}{\{\,P\,\}\ c; d\ \{\,R\,\}}$$

$$\textsc{HIf}\quad \frac{\{\,P \land b\,\}\ c\ \{\,Q\,\} \qquad \{\,P \land \neg b\,\}\ d\ \{\,Q\,\}}{\{\,P\,\}\ \text{if}\ b\ \text{then}\ c\ \text{else}\ d\ \{\,Q\,\}}$$

The **skip** statement does nothing, and hence a precondition P that holds before the statement also holds after executing it. The sequential composition statement satisfies a Hoare triple if an intermediate "connecting" condition Q can be found that connects the two statements. In other words, a condition Q has to exist, which is both a suitable postcondition for c as well as a suitable precondition for d. The inference rule for if requires two scenarios to be considered: one where b evaluates to *true*, and one where b evaluates to *false*. The Hoare triples for each branch can use this extra knowledge (meaning, whether b evaluated to *true* or *false*) in proving their own respective Hoare triples.

Finally, we define an inference rule that is not related to a particular statement:

$$\textsc{HConseq}\quad \frac{P \to P' \qquad \{\,P'\,\}\ c\ \{\,Q'\,\} \qquad Q' \to Q}{\{\,P\,\}\ c\ \{\,Q\,\}}$$

This rule is useful when the pre- and postconditions of two Hoare triples do not line up precisely. It effectively allows weakening a the precondition of a Hoare triple, and strengthening a postcondition. The only caveat is that the two hypotheses that check the weakening and strengthening of the pre- and postcondition will need to be checked outside the deductive system, as our definition of Hoare logic does not include inference rules for propositional logic. When the implications are trivial, we omit them.

Example We will now illustrate Hoare logic with a small example. Consider a company that builds automatically opening and closing garage doors, and which has recently decided to start developing the control software as well. To develop the control software, they have decided to use the language defined earlier.

The language first has to be extended to make it suitable for implementing garage door control software. In particular, the company adds two statements to **open** and **close** garage doors. In addition, boolean conditions are extended to also contain a predicate to check if the garage door is open or not:

$$c ::= \cdots \mid \textbf{open} \mid \textbf{close} \qquad\qquad b ::= \cdots \mid \textbf{isOpen}$$

The garage doors that this company builds have a particular design: if the **open** statement is executed when the garage door is already open, *the garage door engine will burn out.* To prevent this from happening in programs verified with Hoare logic, the company restricts the inference rules for **open** and **close** as follows:

$$\frac{\textsc{HOpen}}{\{\, \neg\textbf{isOpen} \,\}\ \textbf{open}\ \{\, \textbf{isOpen} \,\}}$$

$$\frac{\textsc{HClose}}{\{\, \textbf{isOpen} \,\}\ \textbf{close}\ \{\, \neg\textbf{isOpen} \,\}}$$

By defining the inference rules this way, a program can only be proven correct if it never executes **open** when the garage door is already open. However, note that this does not prevent you from writing programs that burn out the engine. The safety property "engine does not burn out" is conditional on proving correctness of a program *before* you execute it. In other words, defining inference rules in a certain restricted way does not stop an engineer from writing the program "**open**; **open**", uploading that to a garage door controller, and then burning out the garage door engine. However, if the engineer *verifies* their program first with

Hoare logic, and only then uploads the program to the controller, engine burn-outs will be prevented.

The challenge we now face is the following: can we write a program that will never make the garage door engine burn out?

We will tackle this challenge by first constructing a program, called SafeOpen, that we believe should work.[3]. Then we will specify the behaviour of SafeOpen using a Hoare triple, and use Hoare logic to prove the program correct.

3: Note that this is an instance of the *bottom-up verification* approach introduced in Chapter 1. We could also use a *top-down* approach, where we first write down the Hoare triple we need, and fill in the missing program c in steps. We will leave this as an exercise to the reader.

We write the program by following basic intuition: if we always check if the door is open before executing the **open** instruction, the engine should never burn out. We encode this approach by guarding **open** with an if:

$$\text{SafeOpen} \equiv \text{if } \mathbf{isOpen} \text{ then } \mathbf{skip} \text{ else } \mathbf{open}$$

The Hoare triple for SafeOpen should respect the following constraints:

- It should always be executable. Hence, its precondition should be *true*.
- It should always terminate in a state where the garage door open. Hence, its postcondition should be **isOpen**.

This yields the following Hoare triple:

$$\{\, \mathit{true} \,\}\ \text{SafeOpen}\ \{\, \mathbf{isOpen} \,\}$$

For the second step of proving SafeOpen correct, we will start by applying the rule for if, HIf, resulting in the following partial proof tree:

$$\frac{\{\, \mathit{true} \wedge \mathbf{isOpen} \,\}\ \mathbf{skip}\ \{\, \mathbf{isOpen} \,\} \qquad \{\, \mathit{true} \wedge \neg\mathbf{isOpen} \,\}\ \mathbf{open}\ \{\, \mathbf{isOpen} \,\}}{\{\, \mathit{true} \,\}\ \text{if } \mathbf{isOpen} \text{ then } \mathbf{skip} \text{ else } \mathbf{open}\ \{\, \mathbf{isOpen} \,\}}\ \text{HIf}$$

We would like to finish the proof tree by applying the axioms for **skip** and **open**, namely HSkip and HOpen. However, this is not possible because the "*true* ∧" part of the precondition does not match the preconditions of the inference rules. We can drop this part of the precondition in the partial proof tree by using the rule HConseq. In the next two partial proof trees we have omitted the part for the *false* branch of the proof for space reasons, as indicated by the triple dots symbol, but it is symmetric to the *true* branch.

$$\dfrac{\textsc{HConseq}\ \dfrac{\mathit{true} \land \textbf{isOpen} \to \textbf{isOpen} \qquad \{\,\textbf{isOpen}\,\}\ \textbf{skip}\ \{\,\textbf{isOpen}\,\}}{\{\,\mathit{true} \land \textbf{isOpen}\,\}\ \textbf{skip}\ \{\,\textbf{isOpen}\,\}} \qquad \cdots}{\{\,\mathit{true}\,\}\ \text{if } \textbf{isOpen} \text{ then } \textbf{skip} \text{ else } \textbf{open}\ \{\,\textbf{isOpen}\,\}}\ \textsc{HIf}$$

Figure 2.2: Partial proof tree after applying rule HConseq.

Finally, now that the precondition is in the proper form, we can finish the proof tree using the axiom for **skip**, HSkip.

$$\dfrac{\textsc{HConseq}\ \dfrac{\mathit{true} \land \textbf{isOpen} \to \textbf{isOpen} \qquad \dfrac{}{\{\,\textbf{isOpen}\,\}\ \textbf{skip}\ \{\,\textbf{isOpen}\,\}}\ \textsc{HSkip}}{\{\,\mathit{true} \land \textbf{isOpen}\,\}\ \textbf{skip}\ \{\,\textbf{isOpen}\,\}} \qquad \cdots}{\{\,\mathit{true}\,\}\ \text{if } \textbf{isOpen} \text{ then } \textbf{skip} \text{ else } \textbf{open}\ \{\,\textbf{isOpen}\,\}}\ \textsc{HIf}$$

Figure 2.3: Finished proof tree after applying rule HSkip.

Discussion The garage door example shows it is possible to reason precisely about programs with domain-specific requirements through Hoare logic. However, it also shows that even for a basic setting and small programs, such as opening and closing garage doors, the amount of work required can quickly gets out of hand if done manually.[4]

4: We cannot even show the partial proof tree within the margins of the page after applying only two rules!

This leads us to believe that it is crucial to design and implemented these techniques in verification tools, such that we can bring the benefits of program verification to industry, while minimizing effort required of the user. Vice versa, it is important that the core of the algorithms implemented in these verification tools is rooted in formalisms like Hoare logic. This way, we can explain the guarantees they give, and reason about and compare the strengths and weaknesses of verification tools.

2.2 Auto-Active Deductive Program Verification with VerCors

One way to guarantee correctness of programs is by using auto-active deductive program verification tools. These are program verification tools that check if a *program* complies with a *specification*. Often, this specification is provided by the user, but this does not have to be the case. For example, the specification might be implicit, such as with the integer division operator a/b, which requires that b is non-zero. Depending on the complexity of the program, it might also possible to generate annotations [91].

[91]: Lathouwers (2023), *Exploring annotations for deductive verification*

Theoretically speaking, if a deductive program verification tool terminates successfully, this means there exists a Hoare triple, as defined in Section 2.1, with pre- and postconditions matching the specification of the program. The program verification tool guarantees that there exists a proof of triple in an appropriate program logic, e.g. Hoare logic.

The word “auto-active” implies that the tool requires input from the user in the form of annotations. Once these annotations are provided, the tool can analyse the program and its specification without intervention from the user [110].

[110]: Nelson et al. (2019), *Scaling symbolic evaluation for automated verification of systems code with Serval*

The work done in this thesis takes place in the context of VerCors, a program verification tool with a focus on concurrent programs with shared memory. For simplicity, this section we will assume a sequential setting. The concurrent capabilities of VerCors will be introduced in Section 2.4. We will first discuss the general verification workflow in Section 2.2.1, followed by a description of VerCors' architecture in Section 2.2.2. Finally, we will give an introduction to correctness annotations in the context of the Java frontend of VerCors in Section 2.2.3.

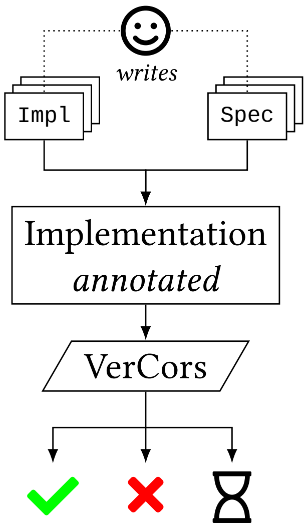

Figure 2.4: VerCors workflow

2.2.1 Verification Workflow

Verifying a program with VerCors works as visualized in Fig. 2.4. The users starts with annotating a *program* with a *specification*. We will show examples of correctness annotations in Sections 2.2.3 and 2.3. The program and its annotations are given to VerCors, which then analyzes the program and verifies that the program respects the annotations. Verification is done by translating the input program and specification to a low-level representation, and then invoking an existing verifier. This is further discussed in Section 2.2.2. Verification with VerCors yields one of the following three results:

1. If all annotations are respected, VerCors prints "verification successful" and terminates. In this case, where VerCors can show that a program follows the specification exactly, we call the program "correct" with respect to its specification (as described by the annotations). Relating this result to Section 2.1, the program and its specification is also a valid Hoare triple.
2. VerCors might report an error on one of the annotations, indicating that the program does not respect this annotation. This means that there is either a bug in the program, or a problem with the annotations.
3. If the correctness proof is too difficult for VerCors, VerCors might time out with an indication of which annotation it could not verify. This might happen when e.g. non-linear integer arithmetic is involved, which is undecidable [160]. VerCors might also not terminate, which can happen when quantifier instantiation heuristics cause the underlying solver to diverge [35].

[160]: Zhang et al. (2024), *Deep Combination of CDCL(T) and Local Search for Satisfiability Modulo Non-Linear Integer Arithmetic Theory*

[35]: Bordis et al. (2024), *Free Facts: An Alternative to Inefficient Axioms in Dafny*

In cases 2 and 3, the user will have to reconsider both the program and specification. VerCors being unable to prove correctness is not a judgement about the quality of the program or specification. There can be a problem (e.g. bug or typo), in either, or even both. After making appropriate changes, the user can consider going through the workflow again.

2.2.2 Architecture

VerCors supports several languages, such as Java, C, OpenCL, Cuda and PVL. To keep the complexity of a multi-language verifier tractable, VerCors is centered around its internal intermediate representation (IR), called Common Object Language (COL). When analyzing a program, the program is first parsed and then translated into COL. In this case, there

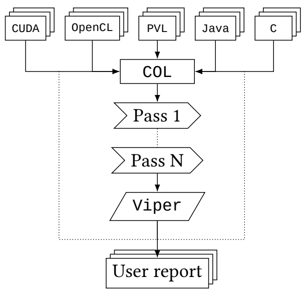

Figure 2.5: VerCors architecture

might still be programming-language specific AST nodes in the IR. Then, VerCors applies rewrite steps that encode high-level constructs from the input programming language into low-level constructs supported by Viper [108], the back-end of VerCors. Viper then in turn translates the low-level program into first-order logic, and submits this to an SMT solver. SMT solvers can automatically prove or disprove a fragment of first-order logic propositions. In practice, we use solvers such as Z3 or CVC5 [16, 107] to check its correctness. When Viper terminates with a verification result VerCors translates the result back to the level of the input sources. This way, the user can reason about the verification errors in the context of the input program, and adjust the specification or implementation if required.

[108]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

[16]: Barbosa et al. (2022), *cvc5: A Versatile and Industrial-Strength SMT Solver*
[107]: De Moura et al. (2008), *Z3: An Efficient SMT Solver*

For example, in the Java frontend, abrupt control flow primitives like `break` and `continue` are handled specially. When these primitives are used together with `finally`, VerCors first transforms all instances of these into exceptional control flow [137]. After this, VerCors applies a transformation that transforms all exceptional control flow into `goto`.

[137]: Rubbens et al. (2021), *Modular Transformation of Java Exceptions Modulo Errors*

```
while(c()) {
  m1();
  if (p()) {
    break;
  }
  m2();
}
```

(a) Before translation

```
try {
  while(c()) {
    m1();
    if (p()) {
      throw new Break();
    }
    m2();
  }
} catch (Break e) { }
```

(b) After translation

Figure 2.6: Example of translating `break` into `throw`

An example of transforming `break` into `throw` is shown in Fig. 2.6. In the top listing there is the input program, consisting of a `while` loop that contains a `break` statement. VerCors translates the `break` statement into a `throw`, and wraps the `while` loop into a `try-catch` block that catches the exception thrown by the `throw` statement, as shown in the bottom listing. This transformation preserves the control flow of the input program, and ensures that all control flow is either sequential or exceptional. This makes control flow in the program simpler overall, as there are less primitives to consider.

This is one of the core ideas of the VerCors architecture: decomposition of larger transformation steps into smaller ones makes the overall architecture more maintainable. In the case of the earlier example, the larger transformation step is handling `break` and `continue` in the presence of `finally`. The smaller transformations are e.g. as `break` to `throw` and exceptions to `goto`.

Most of the work done in this thesis, except for Chapter 3, extends this transformation-based architecture. Essentially,

```java
class C {
  //@ ensures \result == (a ⊕ b) / 2;
  int avg(int a, int b) {
    return (a + b) / 2;
  }
}
```

(a) An implementation of integer averaging that avoids integer overflow

```java
class C {
 //@ requires a >= 0 && b >= 0;
 //@ ensures \result == (a ⊕ b) / 2;
 int smartAvg(int a, int b) {
  int r;
  if (a % 2 == 1 && b % 2 == 1)
    r = 1;
  else
    r = 0;

  return a / 2 + b / 2 + r;
 }
}
```

(b) An implementation of integer averaging that avoids integer overflow

Figure 2.7: Two implementations of `avg` in Java. Paraphrased from Chen [45].

the goal is to solve the problem of the chapter by defining one or more transformations that 1. simplify high-level constructs away, and 2. can be implemented in VerCors. This allows reusing the existing verification infrastructure of VerCors when implementing tool support.

2.2.3 Java Verification with VerCors

As an example of verification in Java, consider the `avg` method in Fig. 2.7a. This method computes the average of `a` and `b` by adding them up and then dividing by two.

In the Java frontend of VerCors, contract annotations are written as *specification comments*. These are standard Java comments where the first character after the comment marker is an "`@`" symbol. With specification comments, contract annotations can be written for Java programs without touching the actual implementation. In addition, specifications comments are ignored by the Java compiler, and hence do not influence the runtime behaviour of the program.

Within specification comments, users can write method contracts. These consist of a precondition and a postcondition. A precondition has the syntax "`requires` *expr*`;`", where *expr* is an expression of boolean type. In particular, *expr* may read but not modify object fields. VerCors checks this automatically. A precondition indicates that *expr* has to hold before the method can be called. Conversely, "`ensures` *expr*;" indicates *expr* should hold when the method terminates. Pre- and postconditions in specification comments correspond to pre- and postconditions in Hoare triples.

An example of a postcondition is on line 2 of Fig. 2.7a, which contains the specification of the `avg` function. In this case, it states that the return value of the method, represented by the primitive `\result`, should be equal to `a` $\oplus$ `b`. Here, $\oplus$ refers to the *mathematical addition* operator. In contrast to plain integer addition, $\oplus$ does not overflow and returns a number in $\mathbb{Z}$.[5] This uncovers a problem with the definition of `avg`: for large values of `a` and `b`, addition will overflow, causing a wrong result to be computed!

5: At the time of writing, VerCors does not support bounded number arithmetic. One way to support this is to integrate predicate subtyping, also referred to as refinement types, into VerCors, as described by Dubbeling [60].

To detect the overflow problem, we can automatically verify Fig. 2.7a with VerCors. The analysis takes a few seconds, and the output will be something like Fig. 2.8.

```
=========================
At avg.java
-------------------------
int avg(int a, int b) {
         [-----
  return (a + b) / 2;
          -----]
}
-------------------------
Addition might overflow.
=========================
```

Figure 2.8: Output of verification of Fig. 2.7a

Fixing the problem To avoid overflow, we can take special care in the implementation of `avg` and distribute division over `a` and `b`. To compute the proper average, we need to carry over a remainder of 1 separately in the case that both `a` and `b` are odd. This is approach is implemented in the `smartAvg` method in Fig. 2.7b. This fix imposes a modest precondition: `a` and `b` must both non-negative[6].

6: We leave generalization of this method for both positive and negative arguments, and its correctness, as an exercise for the reader.

When Fig. 2.7b is analysed, VerCors finds no errors and prints "`Verification completed successfully`". This means that, given that the precondition on line 2 is satisfied, `smartAvg` computes the average of `a` and `b` without causing overflow. Formally, it means that there exists a proof for the following Hoare triple:

$$\{\, \texttt{a} \geq 0 \wedge \texttt{b} \geq 0 \,\}\, \texttt{smartAvg(a, b)} \,\{\, \texttt{result} = \frac{\texttt{a} \oplus \texttt{b}}{2} \,\}$$

Assertions Assertions can be added to a method body as specification comments using the following syntax: `//@ assert` *expr*`;`. VerCors ensures that an assertion holds at that point in the program. If this is not the case, VerCors will fail to verify the input. For example, we can add the following assert to line 10 to locally ensure that we made no mistake when computing `r`:

```
        //@ assert r == 0 || r == 1;
```

To instruct VerCors to assume that an expression holds without checking it first, the "`//@ assume` *expr*" statement can be used. A complete description of the VerCors specification language can be found in the VerCors tutorial [154].

[154]: VerCors team (2025), *VerCors tutorial* (link)

2.3 Permission-Based Separation Logic with PVL

The Prototypical Verification Language (PVL) is an object-oriented programming (OOP) language with contracts and assertions. It is used to prototype verification features in VerCors. It can also be used to verify programs that use programming language features that VerCors does not support yet. In that case, a manual encoding step is required.

Besides standard Object Oriented Programming (OOP) constructs such as classes, fields and methods, it has primitives for verification, such as `assume` and `assert` statements. The syntax of PVL for OOP primitives is a subset of Java. For verification annotations, the syntax is the same as introduced in the previous section, except that annotation comment markers are not necessary.

VerCors verifies memory safety and data-race freedom using permission-based separation logic (PBSL) [66]. A data race occurs when two or more threads access shared memory, and one of the accesses is a write. In other words, shared memory can either be read from by multiple threads, or written to by one thread, but never at the same time. This is the key invariant that PBSL enforces.

[66]: Haack et al. (2015), *Permission-Based Separation Logic for Multithreaded Java Programs*

PBSL achieves this in two parts. First, it allows users to write permission annotations in contracts and assertions. These permission annotations indicate which locations are readable and writeable. VerCors then checks if the program respects these permission annotations. Effectively, a program annotated with permission annotations can only read or write to locations specified by the annotations. Second,

$$
\begin{aligned}
x, y, z &::= \text{field}, \quad v, u, w ::= \text{variable}, \quad m ::= \text{method}, \quad f ::= \text{function}\\
C &::= \text{class}, \quad P ::= \text{predicate}\\
T &::= \texttt{int} \mid \texttt{boolean} \mid \texttt{seq<}T\texttt{>} \mid C \mid \cdots\\
\textbf{prog} &::= \overline{\textbf{decl}} \qquad \textbf{decl} ::= \textbf{cls} \mid \textbf{pred} \mid \textbf{func} \mid \textbf{proc}\\
\textbf{cls} &::= \texttt{lock\_invariant}\ R\texttt{;}\ \texttt{class}\ C\ \texttt{\{}\ \overline{T\ v}\texttt{;}\ \overline{\textbf{pred}}\ \overline{\textbf{func}}\ \overline{\textbf{meth}}\ \texttt{\}}\\
\textbf{pred} &::= \texttt{resource}\ P\texttt{(}\overline{T\ v}\texttt{)}\ \texttt{=}\ R\texttt{;}\\
\textbf{func} &::= K\ \texttt{pure}\ T\ f\texttt{(}\overline{T\ v}\texttt{)}\ \texttt{=}\ H\texttt{;}\\
\textbf{meth}, \textbf{proc} &::= K\ T\ m\texttt{(}\overline{T\ v}\texttt{)}\ \texttt{\{}\ \overline{S}\ \texttt{\}}\\
S &::= H\ \texttt{=}\ H\texttt{;} \mid \texttt{if (}H\texttt{)}\ S\ S \mid \cdots\\
&\quad \mid K\ \texttt{par (}T\ v\ \texttt{=}\ H\ \texttt{..}\ H\texttt{)}\ S\\
E &::= v \mid E\ \texttt{+}\ E \mid \cdots\\
H &::= \cdots \mid H\texttt{.}x \mid \texttt{this}\\
R &::= H \mid \texttt{Perm(}H\texttt{.}x\texttt{,}\ H\texttt{)} \mid R\ \texttt{**}\ R \mid H\ \texttt{==>}\ R\\
K &::= \texttt{requires}\ R\texttt{;}\ \texttt{ensures}\ R\texttt{;}
\end{aligned}
$$

Figure 2.9: PVL syntax

PBSL allows permissions to be split, merged, but never duplicated. This ensures there is only ever a single write permission, or multiple read permissions, but never a write and a read permission simultaneously.

We will first introduce the syntax of PVL in Section 2.3.1. Then, we will introduce permissions and resource predicates in Section 2.3.2. Finally, in Section 2.3.3 we will introduce the concurrency features of PVL that are relevant for this work: *lock invariants* and `par` *blocks*. For a more extensive documentation of PVL, see [13, 154].

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*
[154]: VerCors team (2025), *VerCors tutorial* (link)

2.3.1 Basic Syntax

The syntax of PVL is shown in Fig. 2.9. We first define names for several types of declarations, including some typical elements for each declaration type. E.g. the names x, y and z are symbolic names for class fields. PVL supports several built-in types, such as integers, booleans, and sequences. Each class C is also a distinct type. A PVL program consists of zero or more top level definitions: classes, predicates, functions and procedures. Predicates, functions and procedures also occur as members of a class. When this is the case, they can refer to fields of the class instance using `this`. When a procedure is defined within a class, we refer to it as *method*. The difference between functions and methods/procedures is that functions may only read heap locations, and not modify it.

The syntax of classes and standard statements (e.g. assignment, `if`, `while`, etc.) is similar to Java. Lock invariants, with syntax `lock_invariant` R, are discussed in Section 2.3.3.

For `while` loops, a dedicated verification annotation is supported: the loop invariant, with the following syntax:

```
loop_invariant I;
while (E) { S }
```

The loop invariant allows VerCors to verify the loop without having to unfold every iteration of the `while` loop. It does this by requiring the invariant to hold before the `while` loop, and at the end of every loop iteration. This allows making an inductive argument for correctness of the `while` loop: if the invariant holds before iteration 0, and assuming the invariant holds at the start of iteration n, the invariant holds at the end of iteration n, the invariant may be assumed after the loop terminates, disregarding the number of iterations actually executed.

The fact that loop invariants are required to make the proof finite makes them a bit different from other specification annotations. Whereas other types of annotations, e.g. pre- and postconditions, document and restrict the desired program behaviour, the purpose of loop invariants is to serve as proof hints that allow the verifier to do a finite proof. In contrast, without the inductive argument, VerCors would have to unroll the loop an unknown number of times.

The inference rule for `while` can be defined as follows:

$$\frac{\{\, I \wedge E \,\}\ \mathrm{S}\ \{\, I \,\}}{\{\, I \,\}\ \texttt{while (}E\texttt{)}\ S\ \{\, \neg E \wedge I \,\}}\ \textsc{HWhile}$$

Note that during one of the loop iterations, the invariant may be temporarily violated. What is key for the inductive

```
requires a >= 0 && b >= 0;
ensures \result == (a ⊕ b) / 2;
int avg(int a, int b) {
  int r = (a % 2 == 1 && b % 2 == 1 ? 1 : 0);
  return a / 2 + b / 2 + r;
}
```

Figure 2.10: Implementation of `avg` from Fig. 2.7b reimplemented in PVL

argument is that the invariant is re-established before the iteration finishes.

[55]: Darabi et al. (2017), *A Verification Technique for Deterministic Parallel Programs*

Another special PVL feature is the `par` block statement. It is an important concurrency primitive in PVL [55]. The first part of the `par` block is the contract, which specifies behaviour from the perspective of one thread. Then follows a binder that will contain the index of each thread, followed by a range that determines the number of threads. The body of a `par` block consists of imperative statements that each thread started by the `par` block will execute, such as `if`, `while`, `par`, etc. The semantics of `par` blocks is further discussed in Section 2.3.3.

Figure 2.9 defines several kinds of expressions. Pure expressions E only use local variables, pure operators like + and immutable value constructors (e.g. sequences). A heap-dependent expression H extends E with field dereferencing. Resource expressions R can contain permission annotations, which we discuss further in Section 2.3.2.

Different kinds of expressions are separated at the syntax level to make it easy to distinguish expression capabilities. Intuitively, VerCors uses annotations in the form of R to check memory safety of expressions H. Expressions E are essentially constraints/functions over local state and do not require special care to evaluate.

Contracts K are written using the `requires` and `ensures` keywords, and can be added to methods, procedures, functions. Because functions may only read heap locations, the postcondition of a function must always be an expression H. This is checked separately by VerCors.

Example Figure 2.10 shows the example from the previous section, written in PVL as a top-level procedure. Note how the specification comment markers are absent, and the rest of the implementation is identical.

2.3.2 Permissions

Permissions specify which fields are writable or readable using the following syntax: “`Perm(`$o.x$`, `e`)`”. Here, o is an H expression and e an expression of type fraction s.t. $0 < f \leq 1$. Fraction $f = 1$ specifies read and write access. Fraction $0 < f < 1$ specifies only read access. Fields that are not specified with `Perm`, or with $f = 0$, are inaccessible.

Permissions can be combined using the separating conjunction `**` operator, such that the sum of fractions never exceeds 1 for a field. Permissions can be split and combined, e.g.

`Perm(v.f, 1)` ≡ `Perm(v.f, 1\2) ** Perm(v.f, 1\2)`.

Note the use of “`\`” to indicate fractional division, as opposed to integer division using “`/`”. For boolean expressions, `**` behaves as `&&`. We define the *footprint* of an expression or statement as the permissions required to evaluate or execute it. E.g. a possible footprint of `o.x` would be `Perm(o.x, 1\2)`. Note that for heap locations that are only read, any positive fraction of permission suffices. An acceptable footprint for `o.x = o.y` is `Perm(o.x, 1) ** Perm(o.y, 1\3)`.

Self-framing In PBSL, expressions are “self-framing”, when they include the permissions for all heap locations occurring in the expression *before* said heap location is read. Effectively, an expression being self-framing means that the expression evaluation result will not change, even in the presence of multiple threads executing concurrently.

Syntactically, self-framing implies that if a heap location is read on the right-hand side of a separating conjunction, permission for this heap location must necessarily be present in the left-hand side of this separating conjunction. For example, the expression

```
Perm(x.f, 1) ** x.f == 0
```

is self-framing:

1. `x.f` is the only heap location read by this expression, and

2. permission for `x.f` is specified on the left-hand side of the separating conjunction.

In this example, `x` is some local variable, such as a method argument or perhaps a quantifier binder.

Self-framing of expressions is a well-formedness condition that is automatically checked by VerCors. Several contract annotations are required to be self-framing, such as pre- and postconditions and loop invariants. Assertions do not have to be self-framing: they are well-formed if there is enough permission available from the context to evaluate the asserted expression. For example, in Fig. 2.11 the assertion in the `deduct` method is self-framing and hence well-formed. This is because the precondition of the method provides sufficient permission to read field `balance`. Note that, unfortunately for `bob`, well-formedness does not imply truth. Indeed, when `deduct` is analysed with VerCors, verification will fail. The asserted expression makes more sense as a precondition.

```
requires Perm(bob.balance, 1);
ensures Perm(bob.balance, 1);
void deduct(User bob, int x) {
  assert bob.balance >= x;
  bob.balance = bob.balance - x;
}
```

Figure 2.11: A PVL procedure that deducts `x` from the balance of user `bob`

Adding and removing permissions Permissions can be manipulated directly with `inhale` and `exhale` statements. The `inhale` *R* statement adds the permissions *R* to the current thread. The `exhale` *R* statement first checks if the current thread actually has all the permissions *R*, and then removes them. Verification fails if there are not sufficient permissions available. These statements are verification primitives for encoding programming language semantics. E.g. acquiring a lock that guards write permission to the field `o.x` can be modelled with the statement `inhale Perm(o.x, 1)`. When used with plain boolean expressions, `inhale` and `exhale` behave as `assume` and `assert` respectively.

```
resource nneg(User u) =
  Perm(u.balance, 1) **
  u.balance >= 0;

requires nneg(bob);
ensures nneg(bob);
boolean deduct(
   User bob,
   int x
) {
  unfold nneg(bob);
  boolean enough =
    bob.balance >= x;
  if (enough) {
    bob.balance =
      bob.balance - x;
  }
  fold nneg(bob);
  return enough;
}
```

Figure 2.12: A PVL procedure

Resource predicates Resource predicates group expressions, including permissions, under an opaque name. They are defined using the syntax:

$$\texttt{resource}\ P(\overline{T\ v})\ \texttt{=}\ R\texttt{;}$$

Here, P is the predicate name, $\overline{T\ v}$ is a sequence of typed parameters and R is a resource expression, also called the predicate body. As predicates are opaque, VerCors requires annotations that specify when a predicate body should be exchanged for a predicate name and its arguments, and vice versa. This is done with the `fold` and `unfold` specification statements. More specifically, if the expression R holds in the current verification state, then `fold` $P(\bar{e})$ will cause all permissions in R to be removed from the verification state, while the predicate $P(\bar{e})$ is added to the verification state. The `unfold` statement does the inverse: it removes $P(\bar{e})$, and adds R.

Consider a variation of the previous `deduct` procedure in Fig. 2.12, which returns `true` if there was enough money in the account to deduct. Here, on line 1 the predicate `nneg` is defined, where `nneg` abbreviates "non-negative". This predicate contains a permission for the field `u.balance`, and maintains that it is non-negative. In a way, this predicate specifies a *class invariant* over the `User` class that can be packed and unpacked as necessary [124]. To access the `balance` field, the `nneg` predicate is unfolded on line 11. After trying to deduct `x` from the balance, it is folded on line 18.

[124]: Parkinson (2007), *Class invariants: The end of the road* (link)

There is also the `\unfolding` expression. It unfolds a predicate temporarily for the duration of evaluating the expression, and folds it directly afterwards.

2.3.3 Concurrency in PVL

In this thesis, we use two concurrency features from VerCors and illustrate them using PVL: *lock invariants* and `par` *blocks*.

Lock invariants To mediate access to shared resources, VerCors supports lock invariants, which define resources that are guarded by a lock [65]. A lock invariant is declared on a class using the syntax `lock_invariant` R. The default lock invariant of a class is `true`.

An object can be locked using the statement `lock` H, where H must be an expression of some class type C. When an object is locked, the resource R is added to the state. When unlocking the object with syntax `unlock` H, the lock invariant R needs to hold, after which R is removed from the state.

```
lock_invariant
  Perm(balance, 1) **
  balance >= 0;
class User {
  int balance;

  boolean deduct(int x) {
    lock this;
    boolean enough =
      balance >= x;
    if (enough) {
      balance =
        balance - x;
    }
    unlock this;
    return enough;
  }
}
```

Figure 2.13: The `User` class, with the `deduct` method implemented with a lock invariant

Lock invariants are comparable to predicates, where locking corresponds to unfolding the predicate, and unlocking corresponds to folding the predicate. The difference is that locking and unlocking influences the run-time behaviour of the program, while folding and unfolding are strictly verification annotations. VerCors also checks if the lock invariant is initialized before the object is locked. However, for brevity, we do not further discuss this in our description of PVL.

A variation of the previous `deduct` procedure is shown in Fig. 2.13, this time defined in the context of the `User` class. The lock invariant is defined at line 1, mirroring the contents of the `nneg` predicate from Fig. 2.12. Lines 8 and 15 lock and unlock `this`, giving write access to the `balance` field in the process.

Structured parallelism with the par block For structured parallelism, PVL has the `par` block. It is written using the following syntax:

$$\begin{array}{l}\texttt{par (int } x \texttt{ = } H_l \texttt{ .. } H_h\texttt{) } K \texttt{ \{} \\ \quad \overline{S} \\ \texttt{\}}\end{array}$$

Here, H_l and H_h indicate the bounds of thread identifiers, meaning the `par` block will start $H_h - H_l$ number of threads. The semantics of a `par` block is as follows: when reaching a `par` block, $H_h - H_l$ child threads execute the `par` block body in parallel. The parent thread waits until the child threads finish. This behaviour makes the `par` block structured, in contrast to e.g. starting individual threads. In addition, the contract K required by the par block is asserted for every thread that is started, causing the precondition (resp. postcondition) to be implicitly quantified such that $(\forall i \in [H_l, H_h); R)$ must hold just before (resp. just after) the `par` block, where R is the precondition (resp. postcondition) of K.

```
 1  requires Perm(us[*], 1);
 2  requires
 3    (∀i = 0..N; us[i] != null);
 4  void deductAll(User[] us) {
 5    par (int i = 0 .. N)
 6    requires Perm(us[i], 1) **
 7      us[i] != null;
 8    {
 9      us[i].deduct(5);
10    }
11  }
```

Figure 2.14: A procedure that deducts 5 from all objects in array `us`, where `N` equals `us.length`.

In Fig. 2.14 we revisit `deduct` one final time. Here, `deductAll` deducts 5 from each `User` instance in the array `us`. The syntax used on line 1 is syntax sugar that expands into a quantified permission[7] for all indices of the array.

7: Quantified permissions are orthogonal to the work presented in this thesis. For more information, see the VerCors tutorial [154].

These permissions then get distributed among the individual threads started by the `par` block.

2.4 VeyMont: Choreographic Verification

This thesis builds on earlier work by Van den Bos and Jongmans on verification of choreographies [36, 80, 105]. In this section we will first give an introduction to the essence of choreographies in Section 2.4.1. Then we will describe the verification approach, as implemented in VeyMont and presented in earlier work, in Section 2.4.2.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*
[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*
[105]: Montesi (2023), *Introduction to Choreographies*

2.4.1 Simple Choreographies

Choreographies are a notation to describe interactions between parts of a system. This way, choreographies 1. provide a global view of a system, encompassing all parts of the system simultaneously, while 2. also providing a bridge between the global and the local view through the *endpoint projection*. In this section we define simple choreographies, in which the only interaction allowed is a message exchange between two endpoints. The section is strongly inspired by chapters 1 and 2 of the book "*Introduction to Choreographies*" by Montesi [105].

Syntax A simple choreography consists of a set of participants, which we call *endpoints* in this thesis, and a sequence of communications between these endpoints. We use the following syntax:

$$\begin{aligned} e, a, b, c, d &::= \text{endpoint names} \\ C &::= C; C \mid a \to b \mid \textbf{skip} \end{aligned}$$

Here, e, a, b, c and d are symbolic endpoint names. For concrete endpoint names, we use a sans serif font, e.g. "Alex". C is the choreography, a sequence of communications. **skip** is the empty choreography, and indicates no communication between any endpoints. For example, this is a simple choreography called Ex1 where Alex sends a message to Bob, and then Bob proceeds to send a message to Charlie:

$$\mathsf{Ex1} \equiv \mathsf{Alex} \rightarrow \mathsf{Bob}; \mathsf{Bob} \rightarrow \mathsf{Charlie}$$

Because of their extremely restricted definition, it is clear that choreographies have two favourable properties:

- They are *deadlock free*, because the syntax of choreographies only allows stating communication actions as pairs of endpoints. This syntactically ensures there is always one sender and one receiver in any communication.
- Communications are always *well-typed*. This means that the receiving party will always know what message type they are supposed to receive, and that the sending party has enough information to check they are sending the proper type. This is called *message fidelity*.

For ease of presentation, in this section we omit the types from the communications. However, in a more general setting, each communication is annotated with a type. For example, $\mathsf{Alex} \xrightarrow{\texttt{int}} \mathsf{Bob}$ means that Alex sends a message of type `int` to Bob. Because of message fidelity, Bob will always receive a message of type `int`, and not of some other type.

In this thesis, choreographies are *partially synchronous*: endpoints will block waiting for a message to receive, but they will not wait for the message to be received after they have sent it. Partial synchronicity implies that a seemingly sequential choreography allows multiple interleavings. Consider the simple choreography Ex2:

$$\mathsf{Ex2} \equiv \mathsf{Alex} \rightarrow \mathsf{Bob}; \mathsf{Charlie} \rightarrow \mathsf{Dom}$$

Ex2 allows the obvious execution where Alex sends first. However, because of partial synchronicity, the execution where Charlie sends first is also allowed. In contrast to Ex2, Ex1 allows only one execution: first Alex sends a message to Bob, then Bob sends to Charlie. This is because Bob participates in both communications, and hence the two communications of Ex1 cannot be reordered.

Essentially, in choreographies, the sequential composition operator ";" commutes if the endpoints that appear on the left do not overlap with endpoints on the right:

$$a \rightarrow b; c \rightarrow d \equiv c \rightarrow d; a \rightarrow b \quad \textit{iff} \quad \{a, b\} \cap \{c, d\} = \emptyset$$

The intuition here is that as long as the endpoints involved in two communications do not overlap, the set of allowed executions does not change if you flip a sequential composition. Combined with associativity of sequential composition, choreographies allow significant reordering of communications.

Endpoint projection Choreographies define the *endpoint projection* operator $[\![\cdot]\!]_e$, which accepts a choreography and an endpoint e, and generates code that executes the choreography from the perspective of e. To define this operator, we first need to define the imperative language the endpoint projection generates code for:

$$\begin{aligned} e, a, b &::= \text{endpoint names} \\ s &::= s; s \mid \textbf{send } e \mid \textbf{recv } e \mid \textbf{skip} \mid s \| s \end{aligned}$$

Again, e, a and b are symbolic endpoint names. The statements of the imperative language are elements of s: sequential composition of two statements, sending *to* an endpoint, receiving *from* an endpoint, the skip statement and parallel composition of two statements. The parallel composition interleaves execution of both statements, and finishes when one of its subprograms reduces to skip.

We can now define the endpoint projection by pattern matching on the choreography:

$$\begin{aligned} [\![e \rightarrow a]\!]_e &= \textbf{send } a & [\![a \rightarrow e]\!]_e &= \textbf{recv } a \\ [\![a \rightarrow b]\!]_e &= \textbf{skip} & [\![C_0; C_1]\!]_e &= [\![C_0]\!]_e ; [\![C_1]\!]_e \\ [\![\textbf{skip}]\!]_e &= \textbf{skip} \end{aligned}$$

Summarizing, if the argument e of the endpoint projection occurs in the choreography term currently being matched, a **send** or **receive** statement is returned, depending on the position of e. If e does not occur in the term, the endpoint projection returns a **skip**, meaning the actions involved with the given choreography are not relevant for e. When pattern matching choreographies against these equalities, we assume that if the matched parts are equal, they are assigned to equal variables. For example, $[\![\mathsf{Alex} \to \mathsf{Bob}]\!]_{\mathsf{Alex}}$ can only be reduced to "**send** Bob". We also assume choreographies have no self-communications (e.g. Alex → Alex).

We can now use the endpoint projection to generate the imperative programs for Ex1, one for each participant:

$$\begin{aligned}
[\![\mathsf{Ex1}]\!]_{\mathsf{Alex}} &= \textbf{send}\ \mathsf{Bob}; \textbf{skip} \\
[\![\mathsf{Ex1}]\!]_{\mathsf{Bob}} &= \textbf{receive}\ \mathsf{Alex}; \textbf{send}\ \mathsf{Charlie} \\
[\![\mathsf{Ex1}]\!]_{\mathsf{Charlie}} &= \textbf{skip}; \textbf{receive}\ \mathsf{Bob}
\end{aligned}$$

Finally, the individual endpoint projections must be composed in parallel to get an imperative implementation of the input choreography:

$$[\![\mathsf{Ex1}]\!]_{\mathsf{Alex}} \quad \| \quad [\![\mathsf{Ex1}]\!]_{\mathsf{Bob}} \quad \| \quad [\![\mathsf{Ex1}]\!]_{\mathsf{Charlie}}$$

We can now informally state the correctness criterion of the endpoint projection: every execution allowed by the parallel composition of endpoint projections, is also allowed by the choreography, and vice versa.

For a formal treatment of the definition and semantics of choreographies, and correctness of the endpoint projection, we refer the reader to "Introduction to Choreographies" by Montesi [105].

2.4.2 Verification

Van den Bos and Jongmans introduce a variant of choreographies that support local actions, loops, and verification [36, 80]. They do this by extending simple choreographies and its endpoint projection with additional primitives. In this thesis we refer to this extension of simple choreographies as *local choreographies*. They then show how the choreography can be verified with VerCors. In this section we will summarize the definition of local choreographies, briefly explain how to verify them, and define the endpoint projection.[8]

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

8: While the presentation of local choreographies in this section differs from the work of Van den Bos and Jongmans in terms of syntax, the essence remains the same.

$$
\begin{array}{rcl}
e, a, b & ::= & \text{endpoint names} \\
\textbf{decl} & ::= & \cdots \mid \textbf{local choreography} \\
\textbf{local chor} & ::= & K\ \texttt{local\_choreography(}\overline{T\ v}\texttt{)}\ \{\overline{D_{ep}};\ K\ \texttt{run}\ \{\ \overline{S_{\texttt{chor}}}\ \}\} \\
D_{ep} & ::= & \texttt{endpoint}\ e = C(\overline{H}); \\
S_{chor} & ::= & \texttt{if}\ (H_{chor})\ S_{chor}\ S_{chor} \\
 & \mid & \texttt{loop\_invariant}\ R\texttt{;}\ \texttt{while}\ (H_{chor})\ S_{chor} \\
 & \mid & \texttt{communicate}\ a.x\ \texttt{->}\ b.y\texttt{;} \\
 & \mid & S_{ep} \\
S_{ep} & ::= & e.m(\overline{H})\texttt{;} \mid e.x\ \texttt{:=}\ H\texttt{;}
\end{array}
$$

Figure 2.15: PVL syntax extension for local choreographies

Syntax Figure 2.15 shows the syntax for local choreographies. It reuses syntactical elements of Fig. 2.9: R are resource expressions, possibly containing permissions, H are heap-dependent expressions, x and y indicate symbolic field names, and K refers to a contract.

Local choreographies are top-level declarations that are declared at the same level as PVL classes. A local choreography consists of a set of endpoints, as well as a *run declaration.* Each endpoint is declared with a PVL class type C, of which the constructor is used to initialize the endpoint. Effectively, each endpoint is treated in expressions as being of type C. A run declaration consists of a contract and a sequence of choreographic statements. A choreographic statement can be an if, while loop, communication, or endpoint statement. An endpoint statement is either a method call on an endpoint, or an assignment.

Endpoint statements differ from the other choreographic statements in that they involve only one endpoint. For example, the communication statement involves both a sending and receiving endpoint.

A choreographic expression H_{chor} is a sequence of n expressions $H_1 \&\& \cdots \&\& H_n$. Choreographic expressions have a well-formedness condition: for each H_i, only one endpoint may be mentioned. In other words, if endpoints a and b occur in H_i, it must be the case that $a = b$. For example, assuming `a != b`, the choreographic expression `a.x == 3 && b.y`

`== b.z` is well-formed: the left-hand side contains only `a`, while the right-hand side contains only `b`. In contrast, the choreographic expression `a.x == b.y` is not well-formed, as it contains both `a` and `b`. This property is important for the endpoint projection, as it allows excluding parts of the expression that are not relevant for the target endpoint.

Endpoint statements are similarly constrained: the expressions H of an endpoint statement can only mention the endpoint e, which is fixed at the top-level of the endpoint statement. This restriction ensures that each endpoint statement only uses memory accessible to the endpoint.

To ensure deadlock freedom of local choreographies, there is also a syntactic restriction over endpoints in branches. Specifically, an endpoint can only appear within any branch of an `if` if that endpoint also appears in the condition of the `if`. If this is not the case, endpoints can get out of sync, where one endpoint takes the true branch and the other the false branch. If that happens, and communicate statements are in either branch, this will result in a deadlock. This restriction also applies to `while` loops. A semantic check that is necessary for deadlock freedom is branch unanimity, which is discussed in the next subsection.

Verification transformation To verify choreographies, we define a translation from choreographies to the OOP fragment of PVL. In this thesis, we refer to this type of transformation for choreographies as the *choreographic projection*, written with the curly brace syntax $\{\!|\cdot|\!\}$.

The transformation rules for the choreographic projection, $\{\!|\cdot|\!\}$, for local choreographies are shown in Fig. 2.16. In rule and entry point LcpLocalChor a local choreography is transformed into a method with name `encodedChor`, and each endpoint into an instance of its corresponding class. Each endpoint class instance is assigned to a local variable with the same name as the endpoint, and initialized in order of declaration. Then, the precondition of the `run` declaration is asserted, after which the choreographic statements are transformed and included. After all the choreographic statements, the postcondition of the `run` declaration is asserted.

LcpLocalChor
```
⟦ $K_c$ local_choreography($\overline{T\ v}$) {
     $\overline{D_{ep}}$;
     requires $P$; ensures $Q$; run { $S_{chor}$ }
  } ⟧ =
    $K_c$ void encodedChor($\overline{T\ v}$) {
       ⦃$\overline{D_{ep}}$⦄;
       assert $P$;
       ⦃$S_{chor}$⦄;
       assert $Q$;
    }
```

LcpEndpoint
```
⦃endpoint e = $C(\overline{H})$;⦄ =
     $C$ $e$ = new $C(\overline{H})$;
```

LcpComm
```
⦃communicate $a.x$ -> $b.y$;⦄ =
     $b.y$ = $a.x$;
```

LcpIf
```
⦃if ($H$) $S_t$ $S_f$⦄ =
     assert unanimous($H$);
     if ($H$) ⦃$S_t$⦄ ⦃$S_f$⦄
```

Figure 2.16: Choreographic projection rules for local choreographies

The choreographic equivalents of the statements `if`, `while`, assignment and method invocation are rewritten to the versions in PVL as is. In addition, as shown in rule LcpIf, for `if` and `while`, the *branch unanimity* condition is asserted just before the statement. Similar to the syntactical condition over `if` and `while`, endpoints also need to agree on the condition value, otherwise both branches could be taken at run-time, risking a deadlock. In rule LcpIf, assuming $H = H_1$ && ⋯ && H_n, the function unanimous(H) constructs the expression H_1==H_2&&⋯&&H_{n-1}==H_n. This encodes the condition $\forall i \neq j; H_i == H_j$, ensuring branch unanimity.

The communication statement is rewritten to a plain assignment to model the sending of a message (rule LcpComm).

The PVL method resulting from this encoding simulates the behaviour of the choreography, and can be verified as-is with VerCors. If an error is found, e.g. maybe a precondition of a method call on an endpoint does not hold, this error can be translated to an error at the level of the input choreography in a straightforward manner.

```
local_choreography(boolean dir) {
  endpoint alex = C(dir);
  endpoint bob = C(dir);

  run {
    if (alex.dir && bob.dir) {
      communicate alex.v -> bob.v;
    } else {
      communicate bob.v -> alex.v;
    }
  }
}
```

(a) Input choreography

```
void encodedChor(boolean dir) {
  C alex = new C(dir);
  C bob = new C(dir);

  assert alex.dir == bob.dir;
  if (alex.dir && bob.dir) {
    bob.v = alex.v;
  } else {
    bob.v = alex.v;
  }
}

```

(b) After applying the choreographic projection

Figure 2.17: An example application of the choreographic projection, showing both the input and output

Local choreographic projection example Fig. 2.17 shows both an input choreography in Fig. 2.17a, as well as the encoded PVL output in Fig. 2.17b. There is a rough line-by-line correspondence between the two listings, s.t. lines on the right represent an encoding of lines on the left. The simple condition of the `if` illustrates the usage of local state of endpoints in branches.

Endpoint projection The endpoint projection generates an implementation specialized for one particular endpoint. It is written using the syntax $[\![\cdot]\!]_r$, where r represents the endpoint that is currently being specialized for, the *target endpoint*. The essential effect of the endpoint projection is that it drops every statement that does not contain r, and keeps the others.

Figure 2.18 shows the transformation rules for $[\![\cdot]\!]_r$. Rule LepLocalChor is the entry point of the projection. It generates a method that contains the implementation of the input choreography w.r.t. r, and recursively rewrites the statements of the choreography using the endpoint projection. Note that the target endpoint is passed as an argument to the generated implementation. Creation of the runtime representation r is done by additional supporting code generated by VeyMont.

For choreographic statements, the general rule is that a statement is included if the r appears in it. For `if` statements, this amounts to including it if r appears in the condition. If r does not appear in H, the `skip` statement is re-

LepLocalChor

$$\frac{r \text{ is of type } C \text{ in } \overline{D_{ep}}}{\left\llbracket \begin{array}{l} K_c\ \mathtt{local\_choreography(}\overline{T\ v}\mathtt{)\ \{} \\ \quad \overline{D_{ep}}\mathtt{;} \\ \quad \mathtt{requires}\ P\mathtt{;\ ensures}\ Q\mathtt{;\ run\ \{}\ S_{chor}\ \mathtt{\}} \\ \mathtt{\}} \end{array} \right\rrbracket_r = \\ \qquad K_c\ \mathtt{void\ impl\_}r\mathtt{(}\overline{T\ v}\mathtt{,}\ C\ r\mathtt{)\ \{}\ \llbracket S_{chor} \rrbracket_r\ \mathtt{\}}}$$

LepIf

$$\frac{r \in H}{\llbracket \mathtt{if\ (}H\mathtt{)}\ S_t\ S_f \rrbracket_r = \\ \qquad \mathtt{if\ (}\llbracket H \rrbracket_r\mathtt{)}\ \llbracket S_t \rrbracket_r\ \llbracket S_f \rrbracket_r}$$

LepSend

$$\llbracket L\mathtt{:\ communicate}\ r\mathtt{.}x\ \mathtt{->}\ a\mathtt{.}y\mathtt{;} \rrbracket_r = \\ \qquad \llbracket L \rrbracket_r\mathtt{.writeValue(}r\mathtt{.}x\mathtt{)}$$

LepReceive

$$\llbracket L\mathtt{:\ communicate}\ a\mathtt{.}x\ \mathtt{->}\ r\mathtt{.}y\mathtt{;} \rrbracket_r = \\ \qquad r\mathtt{.}y\ \mathtt{=}\ \llbracket L \rrbracket_r\mathtt{.readValue()}$$

Figure 2.18: Endpoint projection rules for local choreographies

turned (rule omitted). This is similar for the rules for `while`, method invocation and assignment.

Projecting `communicate` requires two rules. One rule applies when r is in the sending position, rule LepSend. In this rule, a call to `writeValue` is generated, which ensures the value $r.x$ is written into the appropriate channel. To acquire an instance of the channel, we use the notation $\llbracket L \rrbracket_r$, which returns the appropriate instance of channel axiomatically. As part of the supporting code generated, VeyMont also generates a channel for each `communicate` statement, which is passed by reference to each corresponding sender and receiver. This is out of scope for the formal definition of the endpoint projection, but further discussed in Chapter 5, [36] and the example in the next paragraph. Rule LepReceive is symmetric to rule LepSend, and if neither rule applies, the `skip` statement can be returned (rule omitted).

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

Local endpoint projection example Figure 2.19 shows the local endpoint projection for the choreography in Fig. 2.17a for target endpoint `alex`. Notice how, besides the boolean `dir` argument and the object instance representing `alex`,

```java
void impl_alex(boolean dir, C alex, Channel alex_bob, Channel bob_alex) {
  if (alex.dir) {
    alex_bob.writeValue(alex.v);
  } else {
    alex.v = bob_alex.readValue();
  }
}
```

Figure 2.19: Output of the local endpoint projection of the choreography in Fig. 2.17a for target endpoint `alex`

also two other arguments are added. These are the channels that implement the two `communicate` statements in the choreography. Code to initialize the object instance for `alex` is not included in this listing, as it is an implementation detail; VeyMont does generate code for this. Finally, notice how the condition for the `if` only includes the part of the condition that is relevant for `alex`.

Formal Methods in Industry 3

Contents

3.1 **Introduction** 56
3.2 **Tunnel System** . . . 60
3.3 **Verification** 61
3.3.1 Event Loop 63
3.3.2 Discussion 66
3.4 **Presentation** 67
3.4.1 Design Process . . . 67
3.4.2 Lessons Learned . . 70
3.5 **Future Research** . . 72
3.6 **Conclusion** 73

Despite successful application of formal methods in industry, uptake is still limited. To better understand the gap between mental models of engineers and formal methods, this chapter presents a case study where a concurrent module of a tunnel control system written in Java is verified for memory safety and data race freedom using VerCors, a software verification tool. This case study was carried out in close collaboration with our industrial partner Technolution, which is in charge of developing the tunnel control software. First, we describe the process of preparing the code for verification, and how we make use of the different capabilities of VerCors to successfully verify the module. The concurrent module has gone through a rigorous process of design, code reviewing and unit and integration testing. Despite this careful approach, VerCors found two memory related bugs. We describe these bugs, and show how VerCors could have found them during the development process. Second, we wanted to communicate back our results and verification process to the engineers of Technolution. We discuss how we prepared our presentation, and the explanation we settled on. Third, we present interesting feedback points from this presentation. We use this feedback to determine future work directions with the goal to improve our tool support, and to further narrow the gap between formal methods and industry.

This chapter is based on "On Deductive Verification of an Industrial Concurrent Software Component with VerCors" by Monti, Rubbens, and Huisman [106], published at ISoLA 2022.

3.1 Introduction

Software components for critical infrastructure should be kept to the highest standards of safety and correctness. Traditional methods for acquiring high safety standards include code reviewing and testing. These improve the reliability of software, but do not and cannot guarantee the absence of bugs. Software is also becoming more concurrent every year. The number of execution scenarios in concurrent software is even greater than in classical sequential software, due to interleaving and timing aspects. This makes code reviewing and testing even less effective. Specifically, there are too many interleavings of multiple threads, causing problematic interleavings to easily be missed during code reviewing and testing. Furthermore, concurrency related bugs such as data races and race conditions are intrinsically difficult to analyse with testing, since their effects are platform dependent. For example, changing the OS of the system could cause previously passing tests to fail, due to different scheduling policies. It is also hard to test for specific interleavings. Hence, methods besides testing and code reviewing are needed to achieve the highest standards of safety and correctness in concurrent software.

To complement classical methods in the context of concurrent and critical infrastructure software, we believe formal methods must be considered. In particular, formal methods that can deal with the concurrent context must be used. In contrast to code review and testing, formal verification is exhaustive and can formally guarantee the absence of bugs in different stages of the software development cycle. Moreover, formal verification uses a standard semantics of the language in question, which guarantees consistent behaviour across platforms. Whenever platforms disagree, formal verification ensures that this difference is accounted for in the code. These properties of formal verification makes software more predictable, and hence safer.

Despite recent advances in software verification capabilities, the use of formal methods in industry is still limited. We think that case studies that show the successful application of formal methods will greatly contribute towards further adoption of formal methods in industry in several ways. First, it showcases the advances and capabilities of software verification tools to our industrial partners. Second, it generates valuable feedback, with which we can improve our tools and further adapt them to the software production cycle. This feedback is also instrumental in bringing the interfaces that formal methods tools provide closer to the mental models of engineers working in industry. Third, case studies like this improve our understanding of the implementation gap in the context of formal methods applied in industry. The implementation gap occurs when there is still some conceptual discrepancy between the model that is analysed using a formal method, and the actual system the formal model is supposed to represent. This work discusses such a case study, where we verify a safety critical software for tunnel traffic control using our software verification tool VerCors.

VerCors is a deductive verifier, specialised in the verification of *concurrent* software [13]. It supports the languages Java, C, OpenCL, and a custom input language called PVL, and can prove several useful generic properties about programs, such as memory safety and absence of data races. VerCors can also prove functional correctness properties, such as “the sum of all integers in the array is computed”. To verify programs with VerCors, the programs must be annotated by the user, following a Design by Contract like approach. Annotations are pre- and post-conditions of methods, specifying permissions to access memory locations and functional properties about the program state. VerCors processes the program and the annotations, and verifies if the program adheres to the annotations by applying a deductive program logic optimised for reasoning about concurrent programs. VerCors has been applied to concurrent algorithms [121, 142, 143], and also to industrial code in earlier case studies [74, 120]. For more information about the VerCors tool, see Chapter 2 or Armborst et al. [13].

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

[121]: Oortwijn et al. (2020), *Automated Verification of Parallel Nested DFS*

[142]: Safari et al. (2022), *Formal verification of Parallel Prefix Sum and Stream Compaction algorithms in CUDA*

[143]: Safari et al. (2020), *Formal Verification of Parallel Prefix Sum* (link)

[74]: Huisman et al. (2020), *On the Industrial Application of Critical Software Verification with VerCors*

[120]: Oortwijn et al. (2019), *Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System*

This chapter is the result of a close collaboration with Technolution [151], a Dutch software and hardware development company located in Gouda, the Netherlands, with a recorded experience in developing safety-critical industrial software. It is also the next part in a line of work to investigate the feasibility of applying formal methods within the design and production process of Technolution. For more

[151]: (2022), *Technolution webpage.* (link)

information on the earlier parts, we refer the reader to [74, 120]. Finally, this chapter is also an attempt to approach industrial partners to collaborate on the broader goal of making deductive verification, and specifically VerCors, available for industrial practitioners. This collaboration is carried out in the context of the *VerCors Industrial Advisory Board*, with the goal to learn how to introduce deductive verification into the production cycle of software, and to improve the tool and make it easier to use. Another important goal of the VerCors Industrial Advisory Board is to make industrial partners aware of the guarantees of formal verification, in contrast to the weaker guarantees of testing based quality assurance. This chapter discusses our efforts to communicate our results and explain the verification process to the engineers at Technolution, as well as their feedback on our approach.

[74]: Huisman et al. (2020), *On the Industrial Application of Critical Software Verification with VerCors*
[120]: Oortwijn et al. (2019), *Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System*

In particular, we discuss the verification of software for a tunnel on a road called the Blankenburgverbinding [25]. This tunnel and the hard- and software supporting it will be responsible for funnelling thousands of cars every day. The control software of this tunnel monitors and controls almost every aspect of the tunnel, in both normal and calamity situations. Thus, in order to give some safety guarantees to its daily users, it is highly important that the software conforms to the requirements of the national regulations and to the specifications provided by the engineers who design and develop it. To demonstrate how formal methods can help here, we applied our software verification tool VerCors to a submodule of the control software of the tunnel, in particular to analyse concurrency related issues such as data races and memory safety.

[25]: Rijkswaterstaat (2022), *Welkom bij de Blankenburgverbinding* (link)

To develop the tunnel software, Technolution followed an iterative V-model approach. The customer handed in the requirements in form of the Basic Specification of Technical Installations for Tunnels (BSTTI[1]) [90] and LTS specifications. These were used to derive actual software requirements via system decomposition and design. In addition to the custom validation flow of the V-model, the customer also imposed requirements on the development process. These included, but were not limited to, units being inspected to verify that they implement their requirements via Fagan inspection performed by a developer not

1: *In Dutch:* BasisSpecificatie Tunnel-Technische Installatie
[90]: Rijkswaterstaat (2012), *Landelijke Tunnelstandaard (National Tunnel Standard)* (link)

involved in creation and review of the code, requirements-based testing at software module level (i.e. higher integration level than units) using the MC/DC coverage approach, and UI-design based testing at a software chain level (i.e. integration of multiple systems) with a process flow approach.

This rigorous approach to software development resulted in them spotting some unexpected behaviour in their tunnel software, where a certain condition over the state snapshot of a component was evaluated differently at two spots throughout which the snapshot should remain unchanged. Nevertheless, later they could not reproduce this behaviour and, by the time we were given the code to analyse, they had not been able to spot a bug that might explain this behaviour. The code we received was already in testing phase. As can be seen in this chapter, we discovered concurrency related bugs in this code, which we think were likely the cause of the unexpected behaviour. We show that VerCors can effectively catch this kind of bugs, in production phase, by using simple code annotations in the form of methods pre and post-conditions specifying the memory access pattern of such methods.

The goal of this particular study is three-fold. First we want to investigate how much we can support the verification of industrial Java software with VerCors. Second, we want to focus this time on a *concurrent* piece of software and on concurrency issues such as data races, for which our tool is specialised. Notice that to exploit modern architectures, modern software is often concurrent, and not many deductive verification tools can deal with this. Third, we want to investigate how our verification procedure can be improved for industrial adoption. For this, we are particularly interested in the feedback from the Technolution team with respect to the verification procedure we followed.

Contributions In this chapter we discuss the following:

- Details of the tunnel verification case study, such as the analysis workflow and the problems we discovered.
- The process of communicating our results to Technolution.

- The feedback from Technolution and its engineers regarding our analysis and our presentation of it.
- Future plans for VerCors and the analysis of concurrent industrial software.

Outline Section 3.2 presents the background for this research, i.e., we describe the tunnel software and architecture. In Section 3.3 we discuss our process of verification of the concurrent data manager module, we explain the bugs we spotted, and we show how applying VerCors would have avoided these bugs. Section 3.4 describes the experience of explaining our procedure and reporting our results to the engineers at Technolution, and their feedback. Section 3.5 describes our own reflection and future directions towards our goal of improving VerCors for industrial application. Additionally, we mention some broader goals for the formal methods community. Finally, Section 3.6 summarises and concludes.

3.2 Tunnel System Architecture

In the Netherlands, the architecture of software for tunnels is regulated by the BSTTI [90]. It specifies that the architecture for tunnel software is strictly hierarchical. The system is summarised in Figure 3.1.

[90]: Rijkswaterstaat (2012), *Landelijke Tunnelstandaard (National Tunnel Standard)* (link)

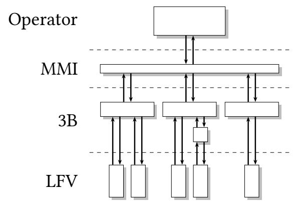

Figure 3.1: Informal overview of the architecture specified by the BSTTI.

At the top layer of this hierarchy are the human operators that operate the system. These operators give commands to the system, and inspect the values of various sensors in the system, using the Human-Machine Interface (MMI[2]) layer. The MMI processes these commands and forwards these to components of the Control, Instruct, Guard (3B[3]) layer. The 3B layer and its components are responsible for the high-level control of the physical subsystems of the tunnel. Examples of 3B components are water drainage, lighting, and electricity systems. As 3B components can be responsible for controlling entire subsystems, they also have a degree of autonomy. The individual 3B elements communicate with components in the Logical Function Fulfiller (LFV[4]) layer. Components in the LFV layer abstract the communication with the sensors and actuators of the tunnel to check and control them. Examples of these sensors

2: Man-Machine Interface

3: Besturing, Bediening, Bewaking

4: Logische Functie Vervuller

and actuators are the smoke sensors and fans, the lights, or the entrance barriers. They can be located at various places in the tunnel and are connected to their LFV counterparts over various kinds of network connections following different protocols

According to the BSTTI [90], the system must follow these general principles:

- Control must flow from the human operator level to the LFV level.
- Communication must take place along the parent-child hierarchy outlined in Figure 3.1. Specifically, sideways communication between neighbouring 3B/LFV components, or between 3B components and LFV components that have no parent-child relation, must not take place.

These principles were prescribed because they make actions taken by the system traceable. If the physical system takes a certain action, the strict hierarchy allows tracing back to which component or decision caused the action. Note that it might not always be a human who caused the action. Since 3B components can have a degree of autonomy, it is possible that an autonomous action causes a physical action to take place.

3.3 Verification of the Concurrent Data Manager Module

When discussing our plans for collaboration with Technolution, the engineers suggested as a case study their new control software for the Baak tunnel. For this tunnel, they have developed a system that is responsible for controlling and reporting on all critical and non-critical components, such as escape doors, fire-prevention measures, water drainage systems, lighting, ventilation, etcetera. In order to reduce the time spent in spotting a concurrent candidate module to analyse, we agreed to meet a first time with an engineer, experienced with the tunnel software, who could guide us through it.

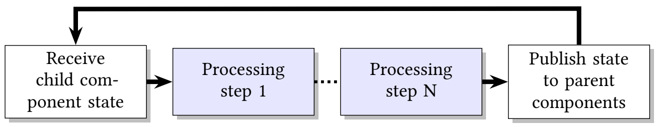
Receive child component state
Processing step 1
Processing step N
Publish state to parent components

Figure 3.2: The 3B function processing event loop.

In this first meeting the engineers from Technolution not only suggested a set of modules to verify, but also pointed out a problem that they would like us to consider, since it was most likely a concurrency issue. The system they had built at that point was functional, behaved properly, and passed all tests. However, sometimes, according to their data logs, certain status data would unexpectedly change during execution of the system. These unexpected changes were never problematic in realistic scenarios, so therefore they considered it benign. However, it was still unexpected, and they would like to understand why this happens.

As a first step, we decided to go through the code base and try to understand the structure. We used a couple of meetings with the Technolution team to get some guidance around the code. Also, several times we asked for further code to inspect, such as supporting libraries of the system.

We found that the components of a tunnel can be quite diverse, and to cope with that diversity, several layers of abstraction and interfacing code had been built into the tunnel software, which made it non-trivial to understand for us. Nevertheless this was not a problem for our verification approach, as it is modular at the level of methods, and annotating the code was straightforward. We ran VerCors on the fly, while annotating the code, and even mocked some library calls, by means of abstract methods and ghost code. We did have problems with VerCors lacking support for some frequently used Java features, such as inheritance and generics. Once we decided on the module to verify, the effort to abstract from unsupported Java features and annotate the code was very little; the annotations were trivial to us, and it took just an afternoon to reach the conclusions. Moreover, due to the simplicity of the specification, VerCors was able to verify the code in just a couple of seconds.

3.3.1 Event Loop Analysis

We inspected the event loop of the main module, because most of the concurrent behaviour happens there. This involved peeling off the abstraction layers of the event loop framework, which is responsible for receiving and dispatching messages and executing each step of the processing loop of 3B components. This process repeats until the main module is shut down. An illustration of the typical processing loop of a 3B function can be found in Figure 3.2. A processing loop starts by obtaining the state of all the child components for this 3B function (first rectangle in the figure). This state is then used to take control decisions along several processing steps inside the loop (shaded rectangles in the figure). It is here that the Technolution engineers where suspecting that something is wrong. In particular, during this control decision period, this state *must not change*. The suspicion was that somehow the state *was being changed*.

Continuing the explanation of Figure 3.2, at the end of the loop our own state is prepared and made available to the upper 3B functions in the hierarchy (see Figure 3.1 for clarification). In general, 3B functions and LFV components work asynchronously. The communication of the state between LFVs and 3B functions is managed by specialised data managers which need to synchronise the state of these asynchronous elements at the start and end of the 3B function processing loop. In a generic *data manager* of the event loop framework, we were able to spot two problems through manual inspection of the source code.

Problem 1: forbidden data sharing The first problem was related to aliasing between references to data structures representing the status of the child components of a 3B function. To better understand this, let us look at Figure 3.3: each 3B function uses two copies of the data structure representing the status of its child LFVs and 3B functions. One of these copies, the “internal” copy, represents the internal knowledge that a 3B function has of its children. It is used by the 3B function processor to make control decisions and should remain unchanged during the time of a processing loop iteration, this is, along the shaded steps in Figure 3.2.

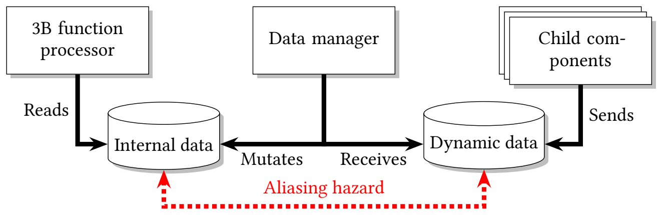

Figure 3.3: Shared data snapshots.

On the other hand, the "dynamic" copy is updated each time a status update message from a child component is received. These messages arrive at any point during a processing loop iteration and the updates are asynchronously applied. At the end of an event iteration, the dynamic copy is used by the *data manager* to update the internal copy.

It turns out that the data manager accidentally aliased both the internal and the dynamic copies. This is a simple but common mistake, and in line with the expectations of the Technolution engineers. The internal copy would then change midway through a processing loop iteration whenever the dynamic copy would receive an update.

As a verification exercise, we decided to annotate the data manager module in order to demonstrate how we could have avoided this mistake by using VerCors. Actually, we simplified the module for the sake of focusing on the interesting aspects, and to avoid incompatibilities with our current support of the Java language. We further discuss this in Section 3.5. As we expected, it turned out to be straightforward to rule out this mistake. Figure 3.4 shows a simplification of the actual aliasing bug and the annotations we used. Lines 9 and 10 are the preconditions specifying that we need permissions to *write* on `internal` and its field `value` while we need to be able to read `dynamic` and its field `value`. Our postconditions, at lines 11 and 12, specify that these permissions should also be returned to the caller. We specify permissions to each of them separately, using the separation conjunction (`**`), since they should correspond to two different data structures.

At line 16, `dynamic` is assigned to `internal`. Therefore, `internal.value` and `dynamic.value` represent the same memory location. At this point VerCors complains about our postcondition. Figure 3.5 shows the VerCors output for

```
 1 class Data{
 2   int value;
 3 }
 4
 5 class Manager{
 6   Data internal;
 7   Data dynamic;
 8
 9   //@ requires Perm(internal, write) ** Perm(dynamic, read);
10   //@ requires Perm(internal.value, write) ** Perm(dynamic.value, write);
11   //@ ensures Perm(internal, write) ** Perm(dynamic, read);
12   //@ ensures Perm(internal.value, write) ** Perm(dynamic.value, write);
13   void sync() {
14     internal.value = dynamic.value;
15     ...
16     internal = dynamic;
17   }
18 }
```

Figure 3.4: Ruling out aliasing with VerCors

this faulty case. The error message

```
PostConditionFailed:InsufficientPermission
```

at line 11 indicates that we are missing permissions to access a memory location. The brackets and dashes at lines 5 and 7 indicate where the problem lies: we do not posses the amount of permission we want to ensure in the second half of line 12 of our code. In fact, we already gave up all the permission we had on this memory location through its alias, in the first half of the same postcondition line. After VerCors indicates something is wrong, the user must find out why this is the case and spot the undesired aliasing.

After analysing this bug with the engineers involved in our case study, we concluded that this aliasing would likely have been the reason of the unexpected behaviour they had detected. It apparently had not affected the overall behaviour of the system, but the reason why such a bug did not extend into a serious fault was not clear. The enormous amount of execution scenarios due to interleaving and timing aspects also makes it difficult to reproduce the immediate effects of this bug. The bug should be fixed since we cannot exclude that it may, under certain circumstances, trigger a major fault in the tunnel control system.

Problem 2: internal data leakage A second bug was spotted while annotating this module for verification with VerCors. Another method of the module was leaking a reference to a

```
Errors! (1)
=== Manager.java                   ===
   //@ requires Perm(internal.value, write) ** Perm(dynamic.value, write);
   //@ ensures Perm(internal, write) ** Perm(dynamic, read);
                                             [-------------------------
   //@ ensures Perm(internal.value, write) ** Perm(dynamic.value, read);
                                              -------------------------]
   void sync() {
     internal.value = dynamic.value;
-----------------------------------------
  PostConditionFailed:InsufficientPermission
=========================================
=== Manager.java                   ===
   //@ requires Perm(internal.value, write) ** Perm(dynamic.value, write);
   //@ ensures Perm(internal, write) ** Perm(dynamic, read);
                                                  [-------------
   //@ ensures Perm(internal.value, write) ** Perm(dynamic.value, read);
                                                   -------------]
   void sync() {
     internal.value = dynamic.value;
-----------------------------------------
  caused by
=========================================
The final verdict is Fail
```

Figure 3.5: VerCors output for alias spotting

```java
class Manager{
  private Data internal; // protected_by(this)

  synchronized Data get_internal() {
    return internal;
  }
}
```

Figure 3.6: Reference to private data leakage

private field of the class. Figure 3.6 illustrates this case. This is not harmful on its own, but it is usually considered bad practice. This may unintentionally allow a user of this class to concurrently access the field without following its synchronisation regime, which may result in a data race. Permission annotations in VerCors will not disallow acquiring the reference, but the annotations will ensure that there is no way to access any fields of this reference without holding the necessary permissions. This restriction rules out any data races.

3.3.2 Discussion on the Discovered Bugs

The two bugs we found are typically overlooked by testing and manual inspection: their effects are triggered by very specific combinations of timings and interleaving that are too complicated to cover by test cases. A manual inspection may mistakenly consider these usages to be safe, or overlook them while searching for functional behaviour bugs instead of memory safety.

The effects of these bugs in a deployed system might be dangerous as it is hard to claim they do not cause incorrect behaviour. To prove that they do not cause incorrect behaviour, one would have to consider all possible interleavings of the processes of the system. The difficulty of this inspection increases exponentially when the number of concurrent processes and timing factors increases. In other words, proving that the system is not affected by the bugs by manual inspection is untractable.

Fortunately, the memory bugs we found are detectable with VerCors, by annotating methods in a straightforward manner with the permissions they require/ensure for the fields that they read/modify. An example of this can be found in the the pre- and post-conditions of Figure 3.4. These annotations are made compulsory by the tool, meaning that if they are not there the tool will terminate with an error. If verification succeeds, then VerCors guarantees that there is no data race in the code.

3.4 Results Presentation

In this section we describe our preparation process and presentation of the results to the larger team of engineers at Technolution, which included a broader group than just those involved in the case study. We also describe our impressions of the final presentation and discuss the most interesting feedback points from the audience.

3.4.1 Design Process

After the case study was analysed by hand and translated to VerCors, we wanted to present our findings to a bigger audience of engineers at Technolution. However, we had experienced in former meetings with the Technolution team that we had not been able to effectively explain what VerCors checks, and how to annotate programs for

VerCors. Therefore, we agreed to be careful and first present the results only to the Technolution team involved in the case study.

It turned out the initial presentation had several shortcomings, which we discuss here, because we think they provide important general insights.

First, the initial presentation tried to explain several useful verification concepts. For example, it discussed the benefits of fractional permissions, compared to non-splittable ownership tickets. It also discussed the difference between annotating only for memory safety, and annotating for functional properties as well. This was done to show how we use VerCors. However, without a formal methods background, the explanation of these tradeoffs is hard to follow. Additionally, most of these concepts are not necessary in order to explain the basis of our approach to verification of memory safety. The solution was to *only* focus on this basis, which is: annotating code with permissions.

Second, the examples used in the initial presentation combined orthogonal concepts to make the examples non-trivial. While engaging for experts, we found out that this is bad for teaching how an approach works. This is especially relevant in the context of a presentation, where the audience needs to understand the slides quickly and explanations need to be short. The solution is to make the examples more targeted. Even when discussing the fundamental basis of our verification approach, each example should only highlight the one relevant aspect of it. For instance, our final presentation contained a code example that had exactly *one* error. The code example on the next slide added exactly *one* annotation, consisting of only one permission, to resolve the error. Additionally, examples from the initial presentation were split up such that each sub-example fit on one screen with a large font. With each example presented in isolation and using as few lines of code as possible, they were also easier to understand.

Third, some of the examples in the initial presentation contained concerns unrelated to verifying concurrency, such as division by zero and rounding. The solution was to ensure that no concerns appear in the example that are unrelated to concurrency or memory safety, since we experienced that this would deviate the attention of the audience to topics we are not interested in discussing.

For the particular case of Technolution, we found out that it was useful to compare our approach with the Rust language, which was familiar to them [100]. This was actually suggested by the Technolution side during our presentation preparation meetings. We also took care with how we phrased certain concepts. Since there might be a difference between what we regard as a permission and what an engineer regards as a permission, we had to ensure this was not a problem from the beginning.

[100]: Matsakis et al. (2014), *The Rust language*

Finally, we made sure to clarify that we do not execute the code, but logically analyse it. For this, we compared it to making a pen and paper proof. This is needed to step away from the usual runtime verification approach of unit and integration testing.

To summarise, we learned that a "good" formal methods presentation to a non-formal audience should have at least the following properties:

- Introduce only key concepts of the formalism in question that are actually needed to understanding the basic idea of the formalism.
- Examples should present only one new concept at a time. Combining orthogonal concepts into one example is not helpful.
- Examples must be short, to ensure they fit on one slide, can be interpreted quickly by the audience, and also be explained quickly by the presenters.
- Examples must not contain unrelated concerns. The domain of the audience might introduce concerns the presenters are not aware of. Therefore, experts in the domain of the audience should be asked beforehand.
- Determine concepts the audience is already familiar with, and draw parallels between those and the concepts in the presentation. However: take care that the audience does not take this analogy too far, to avoid misunderstanding. Avoiding reuse of terms from the audience domain can help.

Additionally, our overall approach consisted of several iterations of refining the presentation using feedback of the smaller group. We think this helped us to narrow down what the Technolution engineers would most likely be interested in, what information would benefit them, and what information could be safely discarded from the presentation. It also helped us to agree on the proper language to transfer this knowledge. The drawback of this approach is that it is time consuming, because the presentation had to be presented twice before the final presentation. Furthermore, the feedback had to be documented by Technolution, and also had to be processed by us. Nevertheless, we think that this process will be quicker next time, due to reusing lessons learned in this case study.

3.4.2 Lessons Learned

During and after the final presentation, several questions were asked and comments were made, both by the presenters and the audience. We have collected the most insightful and applicable ones below.

Testing exceeds verification in short term gains During the presentation it was mentioned that some teams do not even use testing to its fullest. We agree with the observation that it is more beneficial for most projects to first test 80% of their code base, before starting to consider formal verification. Additionally, there are formal methods to enhance and/or multiply the testing effort. Some examples are generation of test cases, mutation testing and QuickCheck-like testing [47, 78, 153].

[47]: Claessen et al. (2000), *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*
[78]: Jia et al. (2011), *An Analysis and Survey of the Development of Mutation Testing*
[153]: Timmer et al. (2011), *Model-Based Testing*

Annotation & specification culture Speaking from the experience of the Technolution engineers, it is impossible to ask engineers to write the annotations needed to use VerCors, or formal verification tools in general. Engineers do not even write comments that you would like to have in the general case. Therefore, there is a big gap between the annotations engineers are willing to write, and what verification tools require. This can be improved upon by the formal verification tools, by having smarter tools, generating some annotations, having design shorthands, and setting effective defaults. But, the difference is so big, that to adopt formal verification tools widely, there also needs to be a culture shift about commenting and annotating code.

Similarities to Rust Most engineers have heard of or worked with Rust. Verification tools can exploit this to lower the barrier for using verification tools, and make them more easily understandable and adoptable.

Optimise for the common case Related to Rust, an approach that the engineers thought could be useful to verification tools is the "optimise for the common case" approach. In this approach, tools optimise for the use case that is most common in practice. For exceptional or unsafe use cases, alternative syntaxes and escape hatches are added. Usually, these alternative syntaxes are also more verbose, making non-standard code also visually distinct. Furthermore, the general case should be safe and hard to get wrong. If applied successfully, we expect that the usage of this approach could reduce the amount of annotation needed for verification, improve the readability and decrease the unwillingness of the programmer to follow the verification path.

Library calls Some engineers expressed concerns about not having contracts for libraries that a team uses. It is true that if a library has no contracts, someone needs to write them. However, it is not a problem that the source of the library is not available due to modular verification, which allows methods to be verified without considering the implementations of other methods. Instead, only the contracts of other methods are necessary. Additionally, there are ways to reduce friction caused by these missing contracts. For example, it is possible to create a central database of library contracts. For cases where the specification for a library is not in the database, the specification language could offer syntax for defining contracts for a library separately.

Why not use automatic static analysis tools instead? An engineer pointed out that he had some experience with various static and automatic analysis tools. They raised the valid question of why code should be annotated for VerCors, when there are tools that can spot memory bugs without annotations. Some examples of such tools are Klocwork [85], FindBugs [61], Coverity [50] and SonarQube [146]. Our answer to this question is that these kind of static analysis tools are not verification tools. Instead, they do a "best effort" analysis to find patterns that *may* relate to bugs. This means such tools are not exhaustive, and can report false positives and warnings which have to be manually inspected. Verification tools, in contrast, give strong formal guarantees on the validity of the queried property over the analysed system. In other words, given a specification that faithfully models the desired behaviour, false positives are rare.

[85]: (2022), *Klocwork homepage* (link)

[61]: (2022), *Findbugs homepage* (link)

[50]: (2022), *Coverity homepage* (link)

[146]: (2022), *SonarQube homepage* (link)

Additionally, code analysis tools often can be used in tandem. Therefore, we think it can be beneficial for teams to use tools with different purposes at different stages of development, or even simultaneously. This way the quality of the final product can be maximised.

3.5 Future Research

Future research for the VerCors team will go in several directions.

One direction of research is to reduce the number of annotations required before VerCors can be used. Currently, if there are no annotations in the code, VerCors cannot make any assumptions about the code. However, for industrial code, simple assumptions are often correct. For example, two fields on one object usually do not contain the same reference. We expect that it will cost less effort to annotate for the exceptions of the previous rule, than to annotate wherever it applies. Additionally, research is already being done to see if some of the required annotations can be generated instead.

Another direction of research is to improve the support of VerCors for Java features such as inheritance and generics. Currently a manual translation to a subset of Java is necessary to verify industrial Java code with VerCors, however efforts are being made to improve the support [130, 145].

[130]: Rubbens (2020), *Improving Support for Java Exceptions and Inheritance in VerCors* (link)

[145]: Şakar (2020), *Extending support for Axiomatic Data Types in VerCors* (link)

Finally, we think future research of the formal methods community as a whole should be about designing simpler specification languages which are closer to the concepts and models of software development teams. We found that the semantics of our specification terminology is distant from the intuition of the engineers. The understandability of specification languages is a common problem in the formal verification community. For example, consider Linear or Branching time logics [87], μ-calculus [38] and other algebras commonly used to specify designs in model checking. The learning curve of these languages is too steep and they become impractical for the daily use of a software engineer. Therefore the next step has to be one of effective reduction: reducing the expressive power of these languages down to a level where they can be easily understood, while retaining enough power to check properties that are of interest to the engineers. Progress is already being made on this with languages such as SALT [19] and Sugar [21], and all the work surrounding the Bandera Specification Language (BSL) [49].

[87]: Kropf (1999), *Introduction to Formal Hardware Verification*

[38]: Bradfield et al. (2018), *The Mu-calculus and Model Checking*

[19]: Bauer et al. (2006), *SALT—Structured Assertion Language for Temporal Logic*

[21]: Beer et al. (2001), *The Temporal Logic Sugar*

[49]: Corbett et al. (2000), *A Language Framework for Expressing Checkable Properties of Dynamic Software* (link)

3.6 Conclusion

We have applied VerCors to a submodule of tunnel control software. This software contained a known benign but unexpected runtime behaviour, which lacked an explanation. Through manual analysis, a bug and a weakness were found, one of which is a possible explanation of the unexpected runtime behaviour. We have communicated our results to the Technolution team and the Technolution engineers through a carefully prepared presentation that underwent multiple feedback rounds from the Technolution team. This allowed us to focus on the information that is most useful to the engineers, and leave out the information that is not directly necessary.

The results of this presentation are suggestions and insights from the engineers of Technolution. For example, it was suggested that there are similarities between Rust and our annotations which VerCors can exploit. It was also suggested that there should be support for easily modelling contracts of software libraries. Another observation we have made is that there is a large gap between the annotations that must be written to apply VerCors, and the maximum amount of annotations engineers are typically willing to write. There was also an observation from an engineer that, in the short term, proper testing practices yield more benefits than formal verification does in the short term.

Finally, we have discussed future directions for our work, such as implementing assumptions about the typical structure of industrial Java code in VerCors, as well as adding more extensive support for the Java language in VerCors.

Acknowledgements

We thank Technolution for the opportunity to analyse their code, their guidance and support.

Towards Verified Concurrent Systems in Java

4

Contents

4.1 Introduction **76**
4.2 Related Work . . . **78**
4.3 New Annotations . **79**
4.4 Architecture **80**
4.5 Implementation . . **80**
4.6 Casino Case Study **82**
4.6.1 Solidity 82
4.6.2 Encoding 83
4.6.3 Verification 86
4.6.4 At Run-Time 87
4.6.5 Fixing the Problem 88
4.7 Conclusion **89**

To narrow the gap between software design and formal methods, we present “Verified JavaBIP”, a tool set for the verification of JavaBIP models. A JavaBIP model is a Java program where classes are considered as components, their behaviour described by finite state machine and synchronization annotations. While JavaBIP guarantees execution progresses according to the indicated state machines, it does not guarantee properties of the data exchanged between components. It also does not provide verification support to check whether the behaviour of the resulting concurrent program is as (safe as) expected. This chapter addresses this by extending the JavaBIP engine with run-time verification support, and by extending the program verifier VerCors to verify JavaBIP models deductively. By combining the two formal methods VerCors and JavaBIP, we narrow the implementation gap that exists between the design phase of software development and deductive verification of Java programs. This is beneficial because the two techniques complement each other: feedback from run-time verification allows quicker prototyping of contracts, and deductive verification can reduce the overhead of run-time verification. We demonstrate our approach on the “Solidity Casino” case study, known from the VerifyThis Collaborative Long Term Challenge.

This chapter is based on “JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java” by Bliudze, Van den Bos, Huisman, Rubbens, and Safina [27], published at FASE 2023.

4.1 Introduction

Modern software systems are inherently concurrent: they consist of multiple components that run simultaneously and share access to resources. Component interaction leads to resource contention, and if not coordinated properly, can compromise safety-critical operations. The concurrent nature of such interactions is the root cause of the sheer complexity of the resulting software [29]. Model-based coordination frameworks such as Reo [10] and BIP [17] address this issue by providing models with a formally defined behaviour and verification tools.

[29]: Bliudze et al. (2021), *On methods and tools for rigorous system design*

[10]: Arbab (2004), *Reo: A channel-based coordination model for component composition*

[17]: Basu et al. (2006), *Modeling Heterogeneous Real-time Components in BIP*

[30]: Bliudze et al. (2017), *Exogenous coordination of concurrent software components with JavaBIP*

JavaBIP [30] is an open-source Java implementation of the BIP coordination mechanism. It separates the application model into *component behaviour*, modelled as Finite State Machines (FSMs), and *glue*, which defines the possible stateless interactions among components in terms of synchronisation constraints. The overall behaviour of an application is to be enforced at run time by the framework's engine. Unlike BIP, JavaBIP does not provide automatic code generation from the provided model; instead it realises the coordination of existing software components in an exogenous manner, relying on component annotations that provide an abstract view of the software under development.

To model component behaviour, methods of a JavaBIP program are annotated with FSM transitions. These annotated methods model the actions of the program components. Computations are assumed to be terminating and non-blocking. Furthermore, side-effects are assumed to be either represented by the change of the FSM state, or to be irrelevant for the system behaviour. Any correctness argument for the system depends on these assumptions. A limitation of JavaBIP is that it does not guarantee that these assumptions hold. This chapter proposes a joint extension of JavaBIP and VerCors [13] providing such guarantees about the implementation statically and at run time.

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

VerCors is a state-of-the-art deductive verification tool for concurrent programs that uses permission-based separation logic [7]. This logic is an extension of Hoare logic that allows specifying properties using contract annotations. These

[7]: Amighi et al. (2015), *Permission-Based Separation Logic for Multithreaded Java Programs*

contract annotations include permissions, pre- and postconditions and loop invariants. VerCors automatically verifies programs with contract annotations. For more information about VerCors, see Chapter 2 or Armborst et al. [13].

To verify JavaBIP models, we (i) extend JavaBIP annotations with verification annotations, and (ii) adapt VerCors to support JavaBIP annotations. VerCors was chosen for integration with JavaBIP because it supports multithreaded Java, which makes it straightforward to express JavaBIP concepts in its internal representation. To analyze JavaBIP models, VerCors transforms the model with verification annotations into contract annotations, leveraging their structure as specified by the FSM annotations and the glue.

For some programs VerCors requires extra contract annotations. This is generally the case with `while` statements and when recursive methods are used. To enable properties to be analysed when not all necessary annotations are added yet, we extend the JavaBIP engine with support for run-time verification. During a run of the program, the verification annotations are checked for that specific program execution at particular points of interest, such as when a JavaBIP component executes a transition. The run-time verification support is set up in such a way that it ignores any verification annotations that were already statically verified, reducing the overhead of run-time verification.

This chapter presents the use of deductive and run-time verification to prove assumptions of JavaBIP models. We make the following contributions:

- We extend regular JavaBIP annotations with pre- and postconditions for transitions and invariants for components and states. This allows checking design assumptions, which are otherwise left implicit and unchecked.
- We extend VerCors to deductively verify a JavaBIP model, taking into account its FSM and glue structure.
- We add support for run-time verification to the JavaBIP engine.
- We link VerCors and the JavaBIP engine such that deductively proven annotations need not be monitored at run-time.

- Finally, we demonstrate our approach on a variant of the Casino case study from the VerifyThis Collaborative Long Term Challenge.

Tool binaries and case study sources are available through the artifact [26].

[26]: Bliudze et al. (2023), *Artefact of: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java*

4.2 Related Work

There are several approaches to analyse behaviours of abstract models in the literature. Bliudze et al. propose an approach allowing verification of infinite state BIP models in the presence of data transfer between components [28]. Abdellatif et al. used the BIP framework to verify Ethereum smart contracts written in Solidity [1]. Mavridou et al. introduce the VeriSolid framework, which generates Solidity code from verified models [102]. André et al. describe a workflow to analyse Kmelia models [8]. They also describe the COSTOTest tool, which runs tests that interact with the model. Thus, these approaches do not consider verification of model implementation, which is what we do with Verified JavaBIP. Only COSTOTest establishes a connection between the model and implementation, but it does not guarantee memory safety or correctness.

[28]: Bliudze et al. (2015), *Formal Verification of Infinite-State BIP Models*

[1]: Abdellatif et al. (2018), *Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models*

[102]: Mavridou et al. (2019), *VeriSolid: Correct-by-Design Smart Contracts for Ethereum*

[8]: Andr et al. (2022), *Combining Techniques to Verify Service-based Components* (link)

There is also previous work on combining deductive and runtime verification. The following discussion is not exhaustive. Generally, these works do not support concurrent Java and JavaBIP. Nimmer et al. infer invariants with Daikon and check them with ESC/Java [115]. However, they do not check against an abstract model, and the results are not used to optimize execution. Bodden et al. and Stulova et al. optimize run-time checks using static analysis [34, 147]. However, Stulova et al. do not support state machines, and Bodden et al. do not support data in state machines. The STARVOORS tool by Ahrendt et al. is comparable to Verified JavaBIP [5]. Some minor differences include the type of state machine used, and how Hoare triples are expressed. The major difference is that it is not trivial to support concurrency in STARVOORS. VerCors and Verified JavaBIP use separation logic, which makes concurrency support straightforward.

[115]: Nimmer et al. (2001), *Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java*

[34]: Bodden et al. (2012), *Partially Evaluating Finite-State Runtime Monitors Ahead of Time*

[147]: Stulova et al. (2016), *Reducing the overhead of assertion run-time checks via static analysis*

[5]: Ahrendt et al. (2017), *Verifying data- and control-oriented properties combining static and runtime verification: theory and tools*

4.3 JavaBIP and Verification Annotations

JavaBIP annotations capture the FSM specification and describe the behaviour of a component. They are attached to classes, methods or method parameters, and were first introduced by Bliudze et al [30]. Figure 4.1 shows an example of JavaBIP annotations. `@ComponentType` indicates a class is a JavaBIP component and specifies its initial state. In the example this is the `WAITING` state. `@Port` declares a transition label. Method annotations include `@Transition`, `@Guard` and `@Data`. `@Transition` consists of a port name, start and end states, and optionally a guard. The example transition goes from `WAITING` to `PINGED` when the `PING` port is triggered. The transition has no guard so it may always be taken. `@Guard` declares a method which indicates if a transition is enabled. `@Data` either declares a getter method as outgoing data, or a method parameter as incoming data. Note that the example does not specify when ports are activated. This is specified separately from the JavaBIP component as glue [30].

[30]: Bliudze et al. (2017), *Exogenous coordination of concurrent software components with JavaBIP*

```java
@Port(
  name = PING,
  type = PortType.enforceable
)
@ComponentType(
  initial = WAITING,
  name = ECHO_SPEC
)
public class Echo {
  @Transition(
    name = PING,
    source = WAITING,
    target = PINGED
  )
  public void ping() {
    System.out.println("pong");
  }
}
```

Figure 4.1: Example of a minimal pinging component in JavaBIP

We added component invariants and state predicates to Verified JavaBIP as class annotations. `@Invariant(expr)` indicates `expr` must hold after each component state change. `@StatePredicate(state, expr)` indicates `expr` holds in state `state`. Pre- and postconditions were also added to the `@Transition` annotation. They have to hold before and after execution of the transition. `@Pure` indicates that a method is side-effect-free, and is used with `@Guard` and `@Data`. Annotation arguments should follow the grammar of Java expressions. We do not support lambda expressions, method references, switch expressions, `new`, `instanceOf`, and wildcard arguments. In addition, as VerCors does not yet support Java features such as generics and inheritance, models that use these cannot be verified. These limitations might be lifted in the future.

4.4 Architecture of Verified JavaBIP

The architecture of Verified JavaBIP is shown in Figure 4.2. The user starts with a JavaBIP model, optionally with verification annotations. The user then has two choices: verify the model with VerCors, or execute it with the JavaBIP Engine.

We extended VerCors to transform the JavaBIP model into the VerCors internal representation, Common Object Language (COL). An example of this transformation is given in Figures 4.3 and 4.4. If verification succeeds, the JavaBIP model is memory safe, has no data races, and the components respect the properties specified in the verification annotations. In this case, no extra run-time verification is needed. If verification fails, there are either memory safety issues, components violate properties, or the prover timed out. In the first case, the user needs to change the program or annotations and retry verification with VerCors. This is necessary because memory safety properties cannot be checked with the JavaBIP engine, and therefore safe execution of the JavaBIP model is not guaranteed. In the second and third case, VerCors produces a verification report with the verification result for each property.

We extended the JavaBIP engine with run-time verification support. If a verification report is included with the JavaBIP model, the JavaBIP engine uses it to only verify at run-time the verification annotations that were not verified deductively. If no verification report is included, the JavaBIP engine verifies all verification annotations at run time.

4.5 Implementation of Verified JavaBIP

This section briefly discusses relevant implementation details for Verified JavaBIP.

Run-time verification in the JavaBIP engine is performed by checking component properties after component construction, and before and after transitions. For example, before the JavaBIP engine executes a transition, it checks the

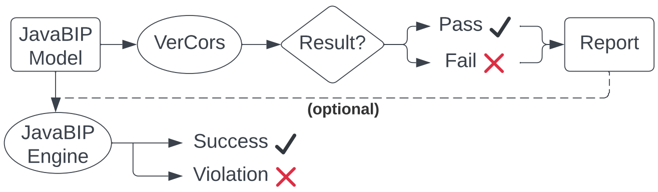

Figure 4.2: Verified JavaBIP architecture. Ellipse boxes represent analysis or execution.

component invariant, the state invariant, and the precondition of the transition. When a property is violated, either execution is terminated or a warning is printed, depending on how the user configured the JavaBIP engine. We expect runtime verification performance to scale linearly, as properties can be checked individually. We have not measured the impact of the use of reflection in the JavaBIP engine.

For deductive verification the JavaBIP semantics is encoded into COL. We describe this with an example. Figure 4.3 shows the `ping` method, where `@Transition` indicates a transition from `PING` to `PING`. The guard indicates that the transition is allowed if there is a ping. `HAS_PING` refers to a method annotated with `@Guard(name=HAS_PING)`, which returns `pingsLeft > 0`.

```
@Guard(name=HAS_PING)
public boolean hasPing() {
  return pingsLeft > 0;
}

@Transition(
  name=PING,
  source=PING,
  target=PING,
  guard=HAS_PING
)
public void ping() {
  pingsLeft--;
}
```

Figure 4.3: Example of a transition and guard in JavaBIP

Figure 4.4 shows the COL form of the `ping` method after encoding the JavaBIP semantics. Line 10 states the precondition, line 13 the postcondition. `PING_state_predicate()` refers to the `PING` state predicate, which constrains the values of the class fields. By default it is just `true`. Since the predicate is both a pre- and a postcondition, it is assumed at the start of the method, and needs to hold at the end of the method. `hasPing()` is the method annotated with he `@Guard(name=HAS_PING)` annotation from Fig. 4.3. The method is called directly in the COL representation. Note the `pure` attribute that was added in Fig. 4.4 on line 5. This ensures the method does not have side-effects, allowing it to be used in contracts. We have implemented such a transformation of JavaBIP to COL for each JavaBIP construct.

```
 1  requires
 2    PING_state_predicate();
 3  ensures
 4    PING_state_predicate();
 5  pure
 6  public boolean hasPing() {
 7    return pingsLeft > 0;
 8  }
 9
10  requires
11    PING_state_predicate()
12    && hasPing();
13  ensures
14    PING_state_predicate();
15  public void ping() {
16    pingsLeft--;
17  }
```

Figure 4.4: Internal representation of `ping` after encoding JavaBIP semantics.

To prove memory safety, we extended VerCors to generate permissions. This ensures verification in accordance with the Java memory model. Currently, each component owns the data of all its fields. This works for JavaBIP models that do not share data between components. For other models, a different approach might be necessary, e.g. VerCors taking into account permissions annotations provided by the user. For more info about permissions, we refer the reader to Chapter 2 or [7].

[7]: Amighi et al. (2015), *Permission-Based Separation Logic for Multithreaded Java Programs*

Finally, scalability of deductive verification of JavaBIP models could be a point of future work, as the number of proof obligations scales quadratically in the number of candidate transitions of a synchronization.

4.6 Casino Case Study

We demonstrate the Verified JavaBIP tool set by applying it to the VerifyThis Casino case study [3]. This case study introduces an implementation of a simple casino in the Solidity smart contract language. It was chosen because it strikes a good balance between being non-trivial, and easy to understand. Additionally, it also contains a problem, detectable with both deductive and run-time verification.

[3]: Ahrendt et al. (2025), *From Model Checking to Deductive Verification: Results from a Smart Contract Community Challenge*

We first discuss the case study and its origin in Section 4.6.1, followed by an explanation of how the case study was encoded in JavaBIP, in Section 4.6.2. We show how to detect the problem in Sections 4.6.3 and 4.6.4, and how to fix the problem in Section 4.6.5. The artifact and documentation to use it is located at [26].

[26]: Bliudze et al. (2023), *Artefact of: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java*

Note: The implementation included with this artifact contains a bug. Because of this, the "Casino Adjusted" case study produces violated invariants which should not happen. In our opinion, the technique presented in this chapter is sound; this bug is caused by a technicality, and not a flaw in the general approach. We suspect that there is an unhandled edge case in the BDD[1] encoding of the JavaBIP model, which the JavaBIP engine uses to determine enabled transitions. The artifact is otherwise fully functional, carefully documented and available online [26].

1: Binary Decision Diagram [22]

4.6.1 Solidity Implementation

This implementation of a casino in Solidity allows players to bet on the outcome of a coin flip. If they guess the correct outcome, they win, and the casino pays out twice the

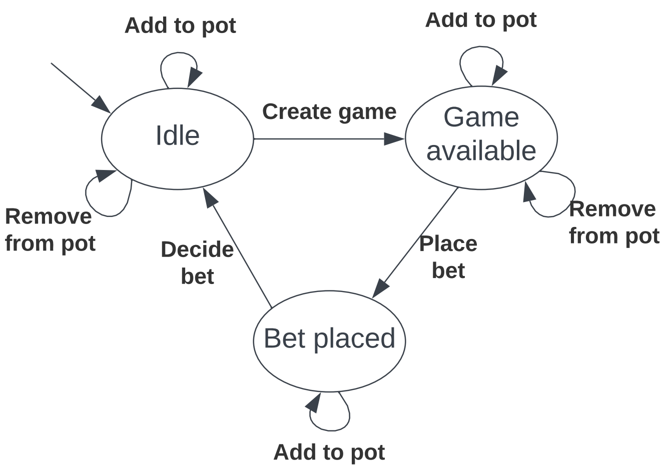

Figure 4.5: Finite state machine representation of the Solidity Casino smart contract. Note that this figure does not visualize the structure of the JavaBIP encoding.

bet. If they guess incorrectly, no money is received, and the casino keeps the money.

Figure 4.5 visualizes the structure of the casino smart contract as a state machine. The operator can add to and remove money from the pot of the casino at any time. Except after a bet is placed, then the operator can only add money to the pot, because placing a bet is only allowed if there is enough money in the pot. If it were possible to remove money from the pot after placing a bet, the casino could end up with a negative balance. For more details, we refer the reader to the VerifyThis website explaining the challenge [155].

[155]: VerifyThis team (2022), *VerifyThis Collaborative Long-term Verification Challenge: The Casino example* (link)

4.6.2 JavaBIP Encoding

The JavaBIP encoding of the Casino diverges from the original casino example to generalize the original case study. Concretely, the concept of “operator” and “player” were factored out and made independent. This allows instantiation of the model with any number of casinos, players, and operators. There are several reasons for the generalization. First, a model that can be instantiated for different parameter values makes doing performance evaluations easier. Unfortunately, because of time constraints, this is still future work. Second, the generalized model also allows use of an advanced feature of the JavaBIP framework that was not

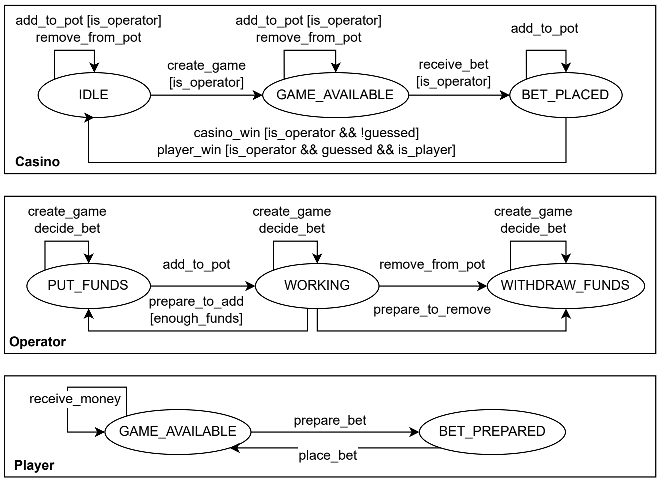

Figure 4.6: Finite state machines of components in JavaBIP Casino case study

relevant before, which is that of a three-way synchronization.

We will now discuss the actual JavaBIP encoding. In this encoding, there are three component types: `Player`, `Operator`, and `Casino`. Changes in the state take place as synchronizations between components. For example, when money is added to the pot of the casino, `Operator` and `Casino` synchronize, transferring the amount to be added from the `Operator` to the `Casino`. Similarly, when `Player` bets, `Player` and `Casino` synchronize, communicating the amount to be betted. Finally, when a bet is decided, a ternary synchronization takes place to establish in all three components that the bet was decided. For `Operator`, this means the amount of money in the `Casino` has either increased or decreased. For `Player`, it either receives twice the bet, or nothing. For the `Casino`, it either needs to pay out, or transfer the bet to the pot. The components and their transitions are visualized in Figure 4.6.

Consider the code in Figure 4.7, which shows a part of the `Casino` component. The component starts in state `IDLE`. As indicated by the `@Transition` annotation, the `ADD_TO_POT` transition can be activated in the `BET_PLACED` state. A requirement of the transition is that only the operator can trigger it, which is fulfilled by the guard part of the transition annotation. Besides that, it is always safe to add money to the pot, so it has no pre- and postconditions. In the full JavaBIP model, the method can also be activated in

```java
@ComponentType(initial = IDLE, name = CASINO_SPEC)
@Invariant("bet >= 0 && pot >= bet")
@StatePredicate(state = IDLE, expr = "bet == 0")
public class Casino {
  int pot;
  Coin guess;
  int bet; // Other variables omitted...

  @Transition(name = ADD_TO_POT, source = BET_PLACED, target = BET_PLACED, guard = IS_OPERATOR)
  public void addToPot(@Data(name = OPERATOR) Integer sender,
                       @Data(name = INCOMING_FUNDS) int funds) {
    pot = pot + funds;
    System.out.println("CASINO" + id + ": " + funds + " received from operator " + sender +
      ", pot: " + pot);
  }

  // Rest of component omitted...
}
```

Figure 4.7: Excerpt of JavaBIP Casino component

other states, but these annotations have been omitted from the chapter for brevity. In the next subsection we will discuss parts of the model that might not be safe, and how these limitations can be expressed in contracts.

The `@Invariant` and `@StatePredicate` annotations in Figure 4.7 indicate the component invariant and state invariant, respectively. The component invariant indicates that `bet` and `pot` should both contain non-negative integers. The state predicate indicates that in the `IDLE` state, the `bet` variable should be equal to zero. The full JavaBIP model contains more annotations, but they are omitted for brevity.

Even though the model was carefully designed, it contains a problem: the operator can withdraw more money than is available in the `Casino` pot. The root cause of this problem is that the operator separately tracks the balance of the casino in its own private field. As this field is private and separate from the casino, it can become out of sync with the actual balance of the casino. If operator decides to withdraw an amount of money based on the number in its out-of-sync private field, there is a chance the balance of the operator becomes negative. In other words, the following chain of events is possible:

1. Casino tells Operator its balance: €100.
2. Operator plans to withdraw €70.
3. Player 2 wins and withdraws its bet of €40. Casino balance is now €60.

```
======================================
100   // Remove money from pot
101   @Transition(name = REMOVE_FROM_POT, source = IDLE, target = IDLE, guard = IS_OPERATOR)
102   public void removeFromPot(@Data(name = OPERATOR) Integer sender,
103                             @Data(name = INCOMING_FUNDS) int funds) {
104     pot = pot - funds;
105     System.out.println("CASINO" + id + ": " + funds +
106     " removed by operator " + sender +
107     ", pot: " + pot);
108   }
--------------------------------------
In this transition the invariant of the component is not maintained, since ...
--------------------------------------
                             [----------
  2   @Invariant("bet >= 0 && pot >= bet")
                              ----------]
--------------------------------------
... this expression may be false
======================================
```

Figure 4.8: Output of VerCors after verifying the Casino JavaBIP model.

4. Operator synchronizes with the casino and withdraws the planned amount of €70. The casino balance is now -€10!

We will show how this problem can be detected with both deductive and run-time verification.

4.6.3 Deductive Verification

To check the Casino model for problems, the source code of the model is analysed with VerCors:

```
$ vercors Casino.java Constants.java Operator.java Player.java
     ↪ Main.java
```

After several seconds of analysis, VerCors reports that several model annotations are not respected. For example, the invariant of the `Casino` component does not hold after removing money from the pot, as shown in Figure 4.8.

Note how both the transition that violates the invariant is shown, as well as which part of the invariant is violated. As `bet` is supposed to be non-negative, and `pot` should be greater or equal to `bet`, it follows that VerCors cannot prove that `pot` remains non-negative.

4.6.4 Run-Time Verification

The JavaBIP engine with run-time verification now checks the contract that, according to the VerCors output, it was not able to verify: `pot >= bet`.

In the following we see that the execution starts with creating the casino, and initializing the operator and players with the predefined amount of money in their wallets.

```
OPERATOR101 created with wallet: 500
CASINO201: INITIALIZED
PLAYER301: INITIALIZED
PLAYER302: INITIALIZED
PLAYER303: INITIALIZED
OPERATOR101: decided to put 446, wallet: 54
PLAYER303: bet 6 prepared, purse: 94
PLAYER301: bet 48 prepared, purse: 52
PLAYER302: bet 20 prepared, purse: 80
CASINO201: GAME CREATED
...
```

The model runs for a while, after which its get into a state where the casino balance is €50. Player two bets €40 on the outcome tails. Meanwhile, the operator plans to withdraw €30. Inbetween planning the withdrawal, and actually withdrawing the money, player two wins the bet. This reduces the casino balance to €10. This does not actually make the pot go negative yet: the casino only takes this transition if it is safe to do so, as part of its transition guards.

Instead, the problem occurs when the *new* casino balance is propagated to the *operator*. In fact, this new balance is incompatible with the amount the operator was planning to withdraw! This triggers an alert in the form of a state predicate violation, as can be seen in the following model trace:

```
CASINO201: received bet: 40, guess: TAILS from player 302
OPERATOR101: decided to withdraw 30, wallet: 50
CASINO201: 40 lost, pot: 10
PLAYER302: won 40, purse: 80
OPERATOR101: making one step in the game
19:06:54.881 [ACTOR_SYSTEM-akka.actor.default-dispatcher-3]
ERROR org.javabip.executor.BehaviourImpl - Operator: State
    ↪ predicate violation: 0 <= amountToMove && amountToMove <=
    ↪ pot
 for the state: WITHDRAW_FUNDS
```

```java
// Remove money from pot
@Transition(name = REMOVE_FROM_POT, source = IDLE, target = IDLE,
            guard = "IS_OPERATOR && ENOUGH_FUNDS")
public void removeFromPot(@Data(name = OPERATOR) Integer sender,
                          @Data(name = INCOMING_FUNDS) int funds) {
  pot = pot - funds;
  System.out.println("CASINO" + id + ": " + funds + " removed by operator " + sender +
                     ", pot: " + pot);
}
```

Figure 4.9: `REMOVE_FROM_POT` transition with the new guard underlined

4.6.5 Fixing the Problem

Generally, there are several ways to fix a broken model: (i) always execute the model with run-time verification, (ii) add extra guards, or (iii) refactor the model.

Always enable run-time verification The user can permanently enable run-time verification. This ensures that any deviations from the model will be detected, meaning execution is always safe. Always enabling run-time verification imposes two limitations. First, the performance penalty of run-time verification has to be acceptable for this particular application. Second, whenever a deviation from the model is detected, the JavaBIP engine will terminate. This possibility of sudden termination also has to be acceptable for this particular application.

Add extra guards Guards can be added to restrict model behaviour. Effectively, this removes the transitions from the model that introduce the problematic behaviour. However, as adding guards might introduce deadlocks, it is not a general solution.

To illustrate the tool set, we will show an example of applying this fix. We add the guard `ENOUGH_FUNDS` to the transition annotation of `REMOVE_FROM_POT`. The updated annotated method is shown in Figure 4.9. This will restrict the transition to only be enabled when there is enough money in the casino pot, ensuring that the casino pot cannot become negative because of withdrawals from the operator.

For deductive verification, the change in the model causes VerCors to succesfully verify the model: it prints “`Verification` `completed successfully.`" upon termination, which indicates that the model is memory safe, data race free, and respects the verification annotations. Furthermore, a verification report is produced, which states that the invariant of the Casino component has been verified. For run-time verification, the change will cause the model to no longer crash at run-time when run-time verification is enabled, as the invariants are no longer violated. Additionally, when the previously produced verification report is included when executing the model, the run-time verifier skips checking the component invariant of Casino while running the model. For a small model such as this one, the difference between doing the run-time checks and not doing them is negligible. However, we speculate that for models with more components, or more complicated annotations, the overhead could be noticeable.

Refactor the model The final solution is to refactor the model to fix the problem. For example, the casino could limit how much the player can bet by first having the player check the current balance. This would effectively encode a kind of compare-and-swap operation into the model. For the JavaBIP casino model, applying this refactoring means that the operator might have to decide how much to withdraw *several times* before actually withdrawing money. This is in contrast to how the original formulation of the model works: in the original model, when the operator decides to withdraw €10, it is only a matter of time until it will actually receives €10. In practice, refactoring the model usually changes the behaviour of the model as well, which might not be acceptable given a particular application.

4.7 Conclusion and Future Work

We presented Verified JavaBIP, a tool set for verifying the assumptions of JavaBIP models and their implementations. The tool set extends the original JavaBIP annotations for verification of functional properties. Verified JavaBIP supports deductive verification using VerCors, and run-time verification using the JavaBIP engine. Only properties that could not be verified deductively are checked at runtime.

In the demonstration we automatically detect a problem on the Casino case study using Verified JavaBIP.

There are several directions for future work. First, support for checking memory safety could be extended by supporting data sharing between components. Second, we want to investigate run-time verification of memory safety. Third, more experimental evaluation can be done on the capabilities and performance of Verified JavaBIP. Fourth and finally, we want to investigate run-time verification of safety properties of the JavaBIP model beyond invariants.

Verified Shared Memory Choreographies 5

Contents

5.1 Introduction . . . **91**
5.2 VeyMont **95**
5.2.1 Workflow 96
5.2.2 Language 96
5.2.3 New Features . . . 99
5.2.4 Example 100
5.3 Verification . . . **101**
5.3.1 Permissions 101
5.3.2 Communication . 103
5.4 Projection **104**
5.4.1 Rules 104
5.4.2 Channels 105
5.5 Case Studies . . . **106**
5.6 Related Work . . **110**
5.7 Conclusion **112**

In the VeyMont tool, choreographies can be used to specify concurrent programs using a sequential format. To support choreography-based development, VeyMont verifies a given choreography for functional correctness and memory safety, and subsequently generates a correct concurrent program. This approach transfers properties verified at the high level of a distributed system to its low-level implementation in a safe manner. It also indirectly narrows the gap between software design and implementation. However, VeyMont initially did not support programs with shared memory, limiting the applicability of VeyMont. In this chapter, we show how we overcome this limitation, by adding support for ownership annotations to VeyMont. Moreover, we also adapted the concurrent program generation, so that it does not only generate code, but also annotations. As a result, further changes and optimizations of the concurrent program can directly be verified. We demonstrate the extended capabilities of VeyMont on illustrative case studies.

5.1 Introduction

In program verification, auto-active verifiers prove correctness of programs automatically, with respect to a given specification. Writing specifications is non-trivial already

This chapter is based on “VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory” by Rubbens, Van den Bos, and Huisman [133], published at iFM 2024.

in a sequential setting, and concurrency makes it even more challenging, as a concurrent program has a combinatorial number of interleavings to be considered. VeyMont [36] addressed this problem by combining choreographies [105], for specifying (concurrent) protocols, with deductive verification. VeyMont generates implementations for choreographies, which can be verified by using the VerCors verifier for concurrent software [13] as back-end. The choreographies of VeyMont allow specifying a concurrent program in a sequential format to ease verification, and then to generate the correct concurrent program.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

[105]: Montesi (2023), *Introduction to Choreographies*

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

In its purest form, a choreography [105] describes a sequence of message exchanges between participants of the choreography, called endpoints. The ordering of messages is partially fixed: an endpoint skips exchanges it does not participate in. Choreographies are deadlock free on the message level, meaning no endpoint will be stuck waiting for a message that will never be sent. Note that in choreographies with shared memory and local actions, as presented in this work, deadlock in general is still possible. This is because endpoints can take local actions, such as acquiring locks, which might deadlock. Choreographies also guarantee that messages are well typed, meaning an endpoint will never receive an `int` when they are expecting a `float`. Finally, for each endpoint a specialized implementation can be generated. When these implementations are executed in parallel, messages are exchanged between processes as specified in the choreography. While similar, choreographies differ from session types [72]: a session type can only be used to type check implementations that are written by a user, choreographies allow automatic derivation of an implementation for each of its endpoints.

[105]: Montesi (2023), *Introduction to Choreographies*

[72]: Honda et al. (1998), *Language Primitives and Type Discipline for Structured Communication-Based Programming*

In the context of VeyMont [36], a choreography specifies a concurrent program, such that its implementation can be generated. VeyMont supports verification of memory safety and functional correctness of these choreographies, which allows reasoning about e.g. program state properties. Such reasoning is not supported for the messages of traditional choreographies, which do not have local actions and shared memory. To support this verification, VeyMont requires users to annotate choreographies with contracts for functional correctness, e.g. pre- and postconditions. Addi-

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

```
 1 choreography incrField() {
 2   endpoint a = Role();
 3   endpoint b = Role();
 4
 5   requires a.x > 0;
 6   ensures a.x == b.x;
 7   ensures a.x > 2;
 8   run {
 9     a.x := a.x + 1;
10     communicate a.x -> b.x;
11     b.x := b.x + 1;
12     communicate b.x -> a.x;
13   }
14 }
```

(a) A shared field is simulated by broadcasting intermediate results between a and b.

```
 1 choreography incrStore(Store s) {
 2   endpoint a = Role(s);
 3   endpoint b = Role(null);
 4
 5   requires a.s.x > 0;
 6   ensures a.s.x > 2;
 7   run {
 8     a.s.x := a.s.x + 1;
 9     // Store reference sent to b
10     communicate a.s -> b.s;
11     b.s.x := b.s.x + 1;
12     // Store reference sent to a
13     communicate b.s -> a.s;
14   }
15 }
```

(b) A shared field is incremented by both a and b. The messages function as barriers.

Figure 5.1: Two choreographies where endpoints a and b each increment a value.

tionally, it generates verification annotations for memory safety: in particular permissions to specify ownership of heap elements, like objects and their fields. The reasoning happens on the level of the choreography, and is preserved in the generated implementation [80]. VerCors [13] is used as the back-end verification engine for VeyMont. For more information about VerCors, see Chapter 2 or Armborst et al. [13].

[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

[13]: Armborst et al. (2024), *The VerCors Verifier: A Progress Report*

An example VeyMont choreography is given in Fig. 5.1a. It defines two endpoints `a` and `b` of class `Role` (lines 2 and 3). The class `Role` has a field `x` of type `int`. The `run` declaration defines actions (line 8): `a` increments `a.x`, and then sends the value stored in `a.x` to `b`. Then `b` increments `b.x`, and sends it back to `a`. The precondition of `run` (line 5) is that `a.x` is more than 0. This is an example of a constraint on input data, and necessary to prove the postconditions: that `a.x` and `b.x` are equal (line 6), and that the value of `a.x` is more than 2 (line 7).

In this chapter we extend VeyMont to address two limitations of the original implementation. The first limitation is the single-owner policy, where each endpoint owns all the fields reachable from it. For example, in Fig. 5.1a, `a` owns its only field `a.x`, while `b` owns `b.x`, in any program state, and sharing is only supported via duplicated values. This choice allowed to automatically generate permission annotations for verifying memory safety with the VerCors back-end.

Unfortunately, the single-owner policy excludes concurrent programs that share memory between threads. This is problematic, because sharing memory is an important pattern in many concurrent programs. Also, choreographies with a single-owner policy do not scale well for more endpoints, and for large data structures the overhead of duplication is large. Finally, the single-owner policy disallows sharing read-only data structures. What we instead wish to have is shared memory as used in Fig. 5.1b. The choreography is the same as Fig. 5.1a, except that endpoint `a` is initialized with a reference to a store `s`, which is used to update the field `x` within. This reference is then communicated between `a` and `b`, instead of the literal integer. While the choreography still includes `communicate` statements to prevent data races, access to `x` is now shared. In this chapter, we extend VeyMont with transfer of ownership using `communicate` statements, i.e. at line 10 of Fig. 5.1b, ownership of `s` should be transferred from `a` to `b`. It is safe to transfer ownership here, because the receiver waits for the message of the sender. This implied synchronization points justifies permission transfers.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

A second limitation of the original VeyMont [36] is that the verification annotations of a choreography are not preserved in the generated implementation. Consequently, the verification properties cannot be directly verified on the implementation. Since the properties have not been proven to hold for any variant of the program, an adapted and verified implementation can only be obtained by adapting and verifying the choreography, and then generating the implementation. From an engineering viewpoint, and especially for small changes, this may cause unnecessary overhead. Also, it is risky to change the generated implementation, e.g. for performance, as this might introduce bugs that cannot be detected by verification. Additionally, the lack of annotations in the generated implementation also prevents application of tools that further process & leverage annotations. One example of such a tool is Alpinist [144], which is a GPU program optimizer that uses annotations to check applicability of optimizations and preserves annotations in the output program.

[144]: Sakar et al. (2022), *Alpinist: An Annotation-Aware GPU Program Optimizer*

Furthermore, because a generated implementation without annotations cannot be verified, bugs in the code generator of VeyMont are not spotted. In this work, we increase confidence in the generated implementation by making it verifiable with VerCors. This allows the user to establish correctness of the generated implementation without depending on the implementation details of VeyMont.

Contributions In this work we present an extension of VeyMont that supports choreographies with shared memory and preserves verification annotations in the generated implementations. This extension allows VeyMont to support fine-grained and dynamic ownership via endpoint annotations. In particular, access to shared memory can be exchanged between endpoints by annotating a `communicate` statement with permissions. By extending this approach to expressions and statements, VeyMont can generate implementations with verification annotations. We demonstrate the extended VeyMont through three consecutive improvements of the Tic-Tac-Toe case study, as presented in the original VeyMont tool paper [36]. These case studies show that even in simple programs, complicated properties can emerge. We provide the full annotated programs and tool implementation in the artifact [132].

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

[132]: Rubbens et al. (2024), *VeyMont Permission Annotations Tic-Tac-Toe Case Studies and Tool Implementation*

Chapter structure We first describe the workflow and choreographic language supported by VeyMont and introduce our approach with an example (Section 5.2). Then, we elaborate how choreographies are verified (Section 5.3), and how the concurrent program with verification annotations is generated (Section 5.4). After, we discuss our case studies (Section 5.5), and related work (Section 5.6). Finally, we conclude and discuss future work (Section 5.7).

5.2 VeyMont Workflow and Choreography Language

We now discuss the workflow and choreography language of VeyMont, including the new features that we added to support shared memory.

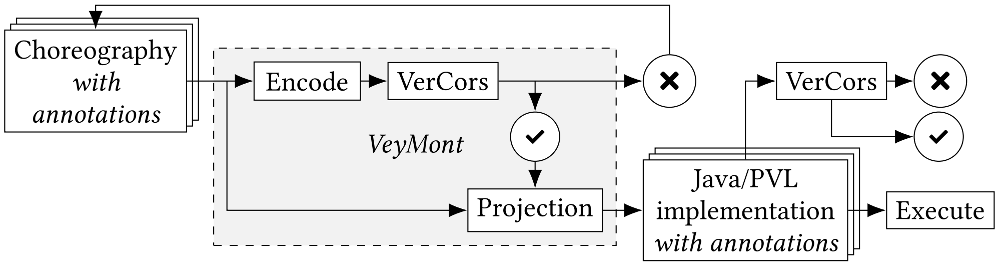
Choreography *with annotations*
Encode
VerCors
VeyMont
Projection
Java/PVL implementation *with annotations*
VerCors
Execute

Figure 5.2: Workflow for using VeyMont choreographies

5.2.1 VeyMont Workflow

The workflow for using VeyMont is shown in Fig. 5.2, and consists of 3 steps:

Step 1: Verify. When a choreography with annotations is input into VeyMont, the semantics of the choreography is encoded (see Section 5.3), such that VerCors can verify it. If verification fails, there is a problem with the choreography: either it has a bug, or the contracts are not properly specified.

Step 2: Endpoint Projection. If verification succeeds, VeyMont applies endpoint projection on the choreography to generate an implementation for each choreography endpoint, with annotations. Section 5.4 discusses the endpoint projection. VeyMont can generate both PVL and Java code.

Step 3: Use. The generated implementation can be used in two ways. First, if Java code was generated, it can be executed. Second, for PVL code, it can be verified with standalone VerCors. If standalone verification fails, there was either a bug in the projection step, the choreography contains annotations that cannot be projected (using `\chor`, see Section 5.2.2), or the user made changes to the generated implementation that are incompatible with the current annotations.

5.2.2 Choreography Language

VeyMont extends the PVL language with syntax for defining choreographies. This syntax extension is summarized informally in Fig. 5.3.

```
⟨GlobalDecl⟩ ::= ⟨Class⟩
    | ⟨Procedure⟩
    | [⟨Contract⟩]
      choreography ⟨Name⟩ (⟨Params⟩) {
        ⟨ChorDecl⟩*
      }

⟨ChorDecl⟩ ::=
    | endpoint ⟨Name⟩ = ⟨Name⟩(⟨Args⟩);
    | [⟨Contract⟩] run { ⟨ChorStmt⟩* }
    | ⟨Method⟩

⟨ChorStmt⟩ ::= ···
    | [⟨Name⟩:] ⟨Expr⟩ . ⟨Name⟩ := ⟨Expr⟩;
    | channel_invariant ⟨Expr⟩;
      communicate [⟨Name⟩:] ⟨Expr⟩ -> [⟨Name⟩:] ⟨Expr⟩;

⟨Expr⟩ ::= ···
    | Perm[⟨Name⟩](⟨Expr⟩, ⟨Expr⟩)
    | (\endpoint ⟨Name⟩; ⟨Expr⟩)
    | (\chor ⟨Expr⟩)
    | \msg
    | \receiver
    | \sender
```

Figure 5.3: VeyMont syntax in EBNF.

Global declarations VeyMont definitions coexist with other PVL definitions such as classes. This way VeyMont programs can use procedures and types from PVL. VeyMont adds a type of global declaration, the `choreography`. This is the root definition for a VeyMont choreography. It consists of an optional contract, a name, parameters, and zero or more choreography declarations. In a VeyMont program, multiple choreographies can co-exist simultaneously.

Choreographic declarations A choreographic declaration is either an `endpoint` declaration, a `run` declaration, or a method declaration. An `endpoint` declaration defines an endpoint that participates in the choreography. Semantically this corresponds to an object created with a constructor. An endpoint has a name, a name of a PVL class, and a list of expressions as arguments for the constructor. The `run` declaration consists of an optional contract, and a body consisting of choreographic statements. Lastly, a regular PVL method definition is also a *ChorDecl*, when its body consists of choreographic statements.

Choreographic statements There are two choreographic statements: choreographic assignment and the `communicate` statement. Choreographic assignment (`:=`) is similar to regular assignment, with the restriction that the value can only be computed using state from one endpoint. It consist of an expression of the object being assigned to, the target field, and an expression. Moreover, the choreographic assignment can optionally be labeled with an endpoint name to enable using a shared field from a different endpoint. The following verifies if `a` has permission for both `b.f` and `a.f`: “`a: b.f := a.f;`”.

The `communicate` statement sends a value from one endpoint to another. It requires a receiving endpoint and field, and a sending endpoint and expression. When endpoints are omitted, they are derived from the expressions, e.g. `a.x` has `a` as the implicit endpoint. `communicate` statements are semi-synchronous: sending is non-blocking, receiving is blocking. Annotations for ownership transfer of a `communicate` statement, as well as functional constraints over the message, can be specified in a `channel_invariant` annotation. For example, the annotation “`channel_invariant \msg > 2; communicate a.x -> b.x;`” can be added to verify that the message sent from `a.x` to `b.x` is bigger than 2. Currently, only the choreographic expressions (explained in the next paragraph) `\msg`, `\sender`, `\receiver` as well as global functions, are allowed within `channel_invariant`. This is purely a limitation of the current implementation, as channels can straightforwardly be extended with extra context.

Within a `choreography` declaration, also selected PVL statements are allowed, such as `assert`, `assume`, `if`, `while` and blocks of statements.

Choreographic expressions There are six choreographic expressions. An endpoint name can be denoted within brackets at a permission annotation, to specify ownership by the endpoint of the stated permission. The keyword `\endpoint` requires the name of an endpoint and an expression. This indicates that, in the encoding (Section 5.3), the expression should only be evaluated for the endpoint, and only included in the implementation of the endpoint (Section 5.4). Within a `channel_invariant`, three additional expressions are available. These are `\msg`, `\sender` and `\receiver`. They are used to indirectly refer to the sender, receiver, and message of the next `communicate` statement.

Finally, the `\chor` keyword wraps an expression. This expression can access memory of all endpoints in the choreography. Specifically, within `\chor` *endpoint ownership annotations* are ignored. Because of this, it cannot be included in the generated implementation. Consequently, if a choreography contains a `\chor` expression, the generated implementation might not verify. More formally, we believe that if a correct choreography does not contain `\chor`, the generated implementation also verifies. We have not proven this, and leave it for future work. The `\chor` keyword is included because it makes the workflow of the original VeyMont [36] possible. It is also useful for prototyping contracts of a choreography, as endpoint ownership annotations prevent asserting expressions that ignore endpoint ownership, which limits the user when debugging annotations. Finally, `\chor` serves a similar role as the `assume` statement. Once a choreography is proven correct, the `\chor` should be removed. An example of `\chor` is in the TTT case study on page 107.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

5.2.3 New Features of VeyMont

This chapter introduces two extensions for VeyMont choreographies: endpoint ownership annotations, and channel invariants.

Endpoint ownership annotations indicate the owner of a permission. When an endpoint e owns a permission `Perm(`$o.x$`, `f`)`, this is written as `Perm[`e`](`$o.x$`, `f`)`. For example, the permission `Perm[alex](`$o.x$`, 1)` allows `alex` to write to field x of object o. When a user writes `Perm(alex.x, 1)`, VeyMont infers automatically that `Perm[alex](alex.x, 1)` was meant. By explicitly writing `Perm[bob](alex.x, 1)` the user specifies that `bob` currently has writing access to `alex.x`, while `alex` has no access. This way, `alex.x` is used as shared memory. Assignments can be annotated similarly: e`:` $o.f$ `:=` v denotes that endpoint e executes assignment $o.f$ `:=` v. Again we allow shorthand notation: `alex.x := 1` denotes `alex: alex.x := 1`. Communications are written as `communicate` s`:` v `->` r`:` u, where endpoint s sends value v to receiver r, which stores it in u. The shorthand notation `communicate alex.x -> bob.y` is also supported.

Channel invariants allow access to memory to be exchanged, i.e. shared. This is done by adding a `channel_invariant`

Figure 5.4: A choreography where the endpoints increment a shared field with ownership annotations. Adapted from Fig. 5.1b. Note that some endpoint ownership annotations that can be inferred by VeyMont are included for clarity.

```
 1 choreography increment(Store s) {
 2   endpoint a = Role(s);
 3   endpoint b = Role(null);
 4 
 5   requires Perm[a](a.s, 1\2) ** Perm[a](a.s.x, 1) ** a.s.x > 0;
 6   ensures Perm[a](a.s, 1\2) ** Perm[a](a.s.x, 1) ** a.s.x > 2;
 7   run {
 8     a: a.s.x := a.s.x + 1;
 9     channel_invariant Perm(a.s, 1\2) ** Perm(a.s.x, 1) ** a.s.x > 1;
10     communicate a: a.s -> b: b.s;
11     assert Perm[b](a.s, 1\2) ** Perm[b](a.s.x, 1);
12     b: a.s.x := a.s.x + 1;
13     channel_invariant Perm(a.s, 1\2) ** Perm(a.s.x, 1\2) ** a.s.x > 2;
14     communicate b: b.s -> a: a.s;
15   }
16 }
```

annotation on a `communicate` statement. E.g. `channel_invariant Perm(alex.x, 1)` gives the receiver write access to `alex.x`, while the sender has lost access after this communication. In other words, permissions are transferred between the sending and receiving party *if and only if* they are stated in the channel invariant.

5.2.4 Motivating Example

We will now further demonstrate these concepts using an example. Figure 5.4 shows how we annotate the program from Fig. 5.1b so that VeyMont can verify the program with the shared field.

Endpoints `a` and `b` are initialized at lines 2 and 3. Then, line 5 states the precondition of the `run` method: `a` has write access to `a.s.x`. On line 8, `a` increments `a.s.x`. This is explicitly denoted with `a:` at the start of the line. On lines 9 and 10, the reference to `Store s` is sent from `a` to `b`. In addition, the channel invariant transfers write access for `a.s.x` from `a` to `b`. This is explicitly verified with the `assert` on line 11. On line 12, `b` performs an increment to `a.s.x`, and then proceeds with the `communicate` statement on line 14, to send write permission back to `a`. The postcondition on line 6 states these write permissions, and additionally that `a.s.x` is more than 2. VeyMont will verify the program, in particular that the postcondition will hold. After that, VeyMont can also be invoked to generate the corresponding concurrent program with threads for endpoints `a` and `b`.

```
 1 choreography setter(Store s) {
 2   endpoint a = Role();
 3   endpoint b = Role();
 4 
 5   requires Perm[a](s.x, 1);
 6   run {
 7     a: s.x := 0;
 8     b: s.x := 1;
 9   }
10 }
```

(a) An incorrect choreography

```
 1 resource perm_x(Role e, Store s) = Perm(s.x, 1);
 2 
 3 requires perm_x(a, s);
 4 void setter_run_a(Store s, Role a, Role b) {
 5   unfold perm_x(a, s);
 6   s.x = 0;
 7   fold perm_x(a, s);
 8   unfold perm_x(b, s);
 9   s.x = 1;
10   fold perm_x(b, s);
11 }
```

(b) Encoding of `run`

Figure 5.5: A choreography and encoding of `run` using permission stratification.

5.3 Choreography Verification

VeyMont generates an encoding of a choreography such that VerCors can verify it. This encoding essentially collapses all endpoint behaviors into one implementation. To prevent permissions of different endpoints from being accidentally combined, permissions are stratified (Section 5.3.1). This also allows encoding the transfer of the message and permissions between two endpoints (Section 5.3.2).

5.3.1 Permission Stratification

To encode permissions labeled with an endpoint owner into PVL, we use PVL predicates to label a permission with its endpoint owner. For each permission annotated with an endpoint owner, we create a predicate wrapping that permission. To this predicate we add an argument that models the endpoint owner. In essence, this argument enforces that a predicate can only be unwrapped if the current endpoint owner is specified. We call this technique “permission stratification”. For example, the permission on line 5 of Fig. 5.5a results in the predicate on line 1 in Fig. 5.5b, where `[a]` in the annotation causes creation of the argument `e` in the predicate. The argument `e` is only used to distinguish stratified permissions with different owners, and therefore does not occur in the predicate.

Adding an extra argument to the predicate, to encode which endpoint owns the permission, works because of the following: unfolding a predicate only succeeds if the correct arguments are used. In this case, unfolding means exchanging a predicate instance for its body, which in turn modifies the verification state. For example, on line 3 in Fig. 5.5b, the predicate `perm_x(a, s)` is required. Within the method, the permission within the predicate can only be accessed using `unfold perm_x(a, s)`. Conversely, the statement `unfold perm_x(b, s)` would fail, as there is only a predicate `perm_x(a, s)` present. Because the arguments have to be stated explicitly to unfold the predicate, the extra argument effectively acts as a "key" to access the permission within the predicate.

VeyMont unfolds wrapper predicates automatically when the endpoint owner of a permission is known, for example, in the case of assignment. This makes the permission in the predicate available for verification. Later, VeyMont folds the predicate, possibly with a new endpoint owner. The `fold` and `unfold` steps are generated by VeyMont according to inferred or user-supplied annotations, and are checked by VerCors. For example, VeyMont generates `unfold` and `fold` annotations before and after assignment to fields. This is shown on lines 5 and 7 of Fig. 5.5b.

The example in Fig. 5.5a shows an incorrect choreography. The two endpoints share a `Store s` and each writes to it. The user specifies that `a` owns the store on line 5. This program contains a data race: `a` and `b` run concurrently and write to the same location. Therefore, verification will fail, with an error on line 8.

The example in Fig. 5.5b shows the encoded choreography with all permissions stratified. Line 1 defines a wrapping predicate for when the field `x` is owned by a given endpoint `Role e`. Line 4 encodes the choreography parameter `Store s`, and endpoints `a` and `b`. Verification with VerCors yields that on line 8 it cannot unfold predicate `perm_x(b, s)` because it is not present in the verification state. Indeed, after line 7, the verification state holds exactly `perm_x(a, s)`, and not `perm_x(b, s)`! One way to fix this example is to send the permission `Perm(s.x, 1)` from `a` to `b` between the two assignment statements using a `communicate` statement. This will exchange `perm_x(a, s)` with `perm_x(b, s)`, at the cost of synchronization at run-time (see Section 5.3.2).

```
choreography incr(int i) {
  endpoint a = Role(i); endpoint b = Role(i);
  requires Perm(a.x, 1) **
    (c() ==> Perm(a.z, 1) ** a.z == a.x);
  requires Perm(b.y, 1);
  run {
    channel_invariant
      c() ==> Perm(a.z, 1) ** \msg == a.z;
    communicate a.x -> b.y;
  } }
```

(a) Input choreography

```
// For each field f ∈ {c, x, y, z}, define:
resource perm_f(Role e, Role r) = Perm(r.f, 1);
int get_f(Role e, Role r) =
  (\unfolding perm_f(e, r) \in r.f)
```

(b) Background definitions for encoding

```
 1 requires perm_x(a, a) **
 2   (c() ==> perm_z(a, a) **
 3     get_z(a, a) == get_x(a, a);
 4 requires perm_y(b, b);
 5 void incr_run(int i, Role a, Role b) {
 6   // Evaluate message
 7   int m = get_x(a, a);
 8   // Assert invariant
 9   assert (c() ==> perm_z(a, a) **
10     m == get_z(a, a));
11   // Transfer invariant
12   if (c()) { unfold perm_z(a, a); }
13   if (c()) { fold perm_z(b, a); }
14   // Store message at target
15   unfold perm_y(b, b);
16   b.y = m;
17   fold perm_y(b, b);
18 }
```

(c) Encoding of choreography

Figure 5.6: Encoding of a choreography with a `channel_invariant` and `communicate` statement. For brevity, method definition `incr` has been omitted in Fig. 5.6c.

By wrapping permissions in predicates, VeyMont can verify the behavior of multiple endpoints within one program. This is the key ingredient that allows verification of choreographies with shared memory.

5.3.2 Encoding of Choreographic Communication

Figures 5.6b and 5.6c show how VeyMont encodes the `communicate` statement and `channel_invariant` of Fig. 5.6a. This is an example to illustrate the encoding, it is not meaningful on its own. All line numbers in this subsection refer to Fig. 5.6c. Summarizing, the encoding consists of 4 steps: line 7 encodes message evaluation, line 9 encodes channel invariant checking, lines 12 and 13 encode the transfer of the channel invariant from `a` to `b`, and lines 15 to 17 encode message reception. The `fold` annotations are required for handling stratified permissions, following the explanation in the previous section.

First, the message to be sent is stored in `m` on line 7. To read `a.x`, the function `get_x` is used. Each `get_f` function unfolds the wrapper predicate `perm_f` to read field *f*. On line 9, the channel invariant is checked using `assert`. Note that the channel invariant was transformed: `m` replaces `\msg`, and a wrapper predicate replaces `Perm(a.z,` 1), following the stratified permissions approach. The owner of this wrapper predicate is a, because a is the sender of the communication.

The channel invariant is transferred from a to b on lines 12 and 13 via the unfold and fold statements. The channel_invariant contains the condition c(), which is an abstract global condition defined for this example. Because of this condition, if statements are also necessary to conditionally unfold and fold the predicates that wrap permissions. For the boolean parts of the invariant, no annotations have to be added, as these are kept track of automatically by the symbolic execution back-end of VerCors. Finally, m is assigned to target location b.y on line 16, which models the receiving of the message by b.

5.4 Endpoint Projection

To generate an implementation for an endpoint of a given choreography, the endpoint projection translates each statement depending on which endpoint is currently the target. We extend the endpoint projection presented in [36] to take into account endpoint ownership annotations. This allows VeyMont to include contracts in the projection, making the generated implementation verifiable if correct annotations are provided. In addition, we show how channel invariants are included in the channel classes that implement communicate statements.

5.4.1 Statement and Expression Projection

Figure 5.7 shows – by example – the endpoint projection rules to generate an implementation for a target endpoint. The top half of the table shows the rules identical to those in [36], the bottom half shows the new rules. Using these new rules, contracts and loop invariants can straightforwardly be transformed and preserved in the generated implementation, which was previously not possible.

We will now further discuss the rules in the bottom half of Fig. 5.7. If the target endpoint participates, i.e. occurs, in a

Choreography with a & b		Projection for: a	Projection for: b
`a.x := 5;`	→	`a.x = 5;`	`/* skip */`
`communicate a.x -> b.y;`	→	`a_b.writeValue(a.x);`	`b.y = a_b.readValue();`
`if (a.x == 5 &&`	→	`if (a.x == 5 &&`	`if (true &&`
`    b.y == 9) {`	→	`    true) {`	`    b.y == 9) {`
`  a.foo(a.x); }`	→	`  a.foo(a.x); }`	`  /* skip */ }`
`b: a.x := 5;`	→	`/* skip */`	`a.x = 5;`
`communicate b: a.x -> a: b.y;`	→	`b.y = a_b.readValue();`	`a_b.writeValue(a.x);`
`Perm[a](x.f, 1)`	→	`Perm(x.f, 1)`	`true`
`(\chor v)`	→	`true`	`true`
`(\endpoint a; v)`	→	`v`	`true`

Figure 5.7: Summary of endpoint projection rules by example. Top half describes rules from [36]. Bottom half describes endpoint ownership annotations.

statement or expression, it is transformed as follows: choreographic assignment (`:=`) is transformed into plain assignment, `communicate` statements are transformed into invocations of `readValue` and `writeValue` methods on channel objects. `Perm[`e`](`$o.x$`, `f`)` is included without `[`e`]` in the generated implementation, and similarly, the keyword `(\endpoint` e`;` v`)` causes v to be included in the generated implementation. If the target endpoint does not participate in a statement or expression, it is omitted or replaced with `true`. The keyword `(\chor` v`)` is handled by always discarding it. This is because `\chor` can freely access the memory of all endpoints, and hence cannot safely be included in the generated implementation (see Section 5.2.2).

5.4.2 Channel Generation

For each `communicate` statement (Fig. 5.8a) VeyMont generates a distinct channel class (Fig. 5.8b). An instance of this class is constructed at the start of the program, and both endpoints of the `communicate` statement are given a reference to it. To send and receive values, the methods `writeValue` and `readValue` can be called. The `lock_invariant` expresses that the `channel_invariant` holds at the moment of transfer, i.e. when `writeValue` has written the communicated value in `msg` and set `hasMsg` to true, and `readValue` has been called after that. Because `write-`

```
channel_invariant
  Perm(\sender.z, 1) **
  Perm(\receiver.z, 1) **
  \sender.z == \receiver.z;
communicate a.x -> b.y;
```

(a) `communicate` with corresponding `channel_invariant`

```
lock_invariant
  Perm(hasMsg, 1) ** Perm(msg, 1) **
  Perm(s, 1\2) ** Perm(r, 1\2) **
  (hasMsg ==> Perm(s.z, 1) **
    Perm(r.z, 1) ** s.z == r.z);
class ChanAB {
  boolean hasMsg; int msg;
  Role s, r; // Sender, receiver

  context Perm(s, 1\8) ** Perm(r, 1\8);
  requires Perm(s.z, 1) ** Perm(r.z, 1) **
    s.z == r.z;
  void writeValue(int m);

  context Perm(s, 1\8) ** Perm(r, 1\8);
  ensures Perm(s.z, 1) ** Perm(r.z, 1) **
    s.z == r.z;
  int readValue(); }
```

(b) Generated channel class.

Figure 5.8: Generated channel class for `channel_invariant` and `communicate`.

`Value` has the `channel_invariant` as precondition, and `readValue` as postcondition, the transfer of the `channel_invariant` is achieved.

Since a channel invariant may refer to both `\sender` and `\receiver`, the generated class contains both endpoints as references `s` and `r`. Read permissions for these fields reside at both endpoints. This way, respective fields of `\sender` and `\receiver`, e.g. `\sender.z`, can be expressed as `s.z`, when calling `writeValue` or `readValue`. The omitted implementations of `writeValue` and `readValue` are standard, where `writeValue` does not block, but `readValue` does.

5.5 Case Studies

To demonstrate the VeyMont extension of this chapter, we present case studies on three variants of Tic-Tac-Toe. Here TTT is the baseline case study, adapted from [36], $\mathrm{TTT}_{\mathrm{msg}}$ uses ownership annotations, and $\mathrm{TTT}_{\mathrm{last}}$ optimizes away a theoretically unnecessary message. We provide the full annotated programs and tool implementation in the artifact [132].

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

[132]: Rubbens et al. (2024), *VeyMont Permission Annotations Tic-Tac-Toe Case Studies and Tool Implementation*

The TTT case study is a variant of the case study discussed in [36]. It is set up to simulate a game of tic-tac-toe on a 3×3 in a distributed setting. This means each endpoint has its own local copy of the board, and as the endpoints take turns they send their moves to each other so the boards stay in sync. When a winning move occurs, or the board runs out of spaces, the game ends.

While each case study has different annotations, the postcondition proven is the same: *after the game terminates, the boards of the two players are identical.*

This postcondition highlights the complexity of verifying an easy to understand choreography. To prove correctness, VeyMont must prove that each move made by one player is also applied to the local board of the other player. This kind of property could also occur when e.g. executing a transaction in a distributed database. When verifying the TTT choreography and ignoring permission stratification, the property is proved automatically. However, once the endpoints are split up into threads with the endpoint projection, a problem arises: the property becomes impossible to state. This is because the property requires player one to make an assertion about the state of player two, and vice versa.

Each case study solves this problem differently. The TTT case study solves the problem by using `\chor`. This allows violating the restriction of stratified permissions, at the cost of missing annotations in the generated implementation. Case studies TTT_{msg} and TTT_{last} use stratified permissions to pass permissions back and forth, ensuring the players can alternate reading and writing to both boards safely. Specifically, TTT_{msg} introduces an extra message at run-time, and TTT_{last} eliminates this run-time overhead by using more complicated annotations. For TTT_{msg} and TTT_{last}, the generated implementations *do* verify.

TTT The TTT case study is similar to the case study presented in [36]. The only changes are the reduction to a 3×3 board instead of an $M \times N$ board, and minor syntactical changes. This is not a limitation of this is chapter, it is merely a simplification for ease of presentation. The choreography of TTT is shown in Fig. 5.9. After the endpoints are initialized, the endpoints enter a loop, where they alternate taking turns. After each turn, the move is send to

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

```
 1 choreography TTT() {
 2   endpoint p1 = Player(0, true);
 3   endpoint p2 = Player(1, false);
 4 
 5   requires p1.myMark == 0 **
 6     p2.myMark == 1 **
 7     (\chor p1.turn != p2.turn **
 8       p1.equalBoard(p2));
 9   ensures (\chor p1.equalBoard(p2));
10   run {
11     loop_invariant /* omitted */ **
12       p1.equalBoard(p2);
13     while(!p1.done() && !p2.done()) {
14       if(p1.turn && !p2.turn) {
15         p1.createNewMove();
16         communicate p1.move.copy() -> p2.move;
17       } else {
18         p2.createNewMove();
19         communicate p2.move.copy() -> p1.move;
20       }
21       p1.doMove();
22       p2.doMove();
23       p1.turn := !p1.turn;
24       p2.turn := !p2.turn;
25     }
26   }
27 }
```

Figure 5.9: Main choreography of the TTT case study. The loop invariant on line 11 is omitted as it is the same as the precondition of run.

the other player so they can update their board. The postcondition is `(\chor p1.equalBoard(p2))`, meaning the board of `p1` is equal to that of `p2` after termination. With automatic permission generation enabled, VeyMont can verify the choreography with the initial approach presented in [36]. The projection on these (old style) choreographies yields generated implementations where the choreography properties hold [80], but verification annotations marked with `\chor` are not present in the generated implementations, and hence the choreography postcondition cannot be verified on it.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

TTT$_{\text{msg}}$ We take a different approach to avoid `\chor`: each endpoint will only keep half permission for their own board. The other halves are pooled and used to establish and maintain board equality. After each turn, these pooled permissions are sent to the other player. Finally, when the game ends, the last player splits the pooled permissions sends half to the other player. This gives both players read permission to both boards, allowing them to state board equality.

To this end, we add to both communications in the while loop the channel invariant of Fig. 5.10a. The sending player provides `1\2` permission for his own board, *and* the other players board, in the channel invariant (using prefix scaling notation `[1\2]` before the predicate on lines 3 and 4). This invariant implies that the sending player is exactly one move ahead (line 5). This makes sense as the receiving player still has to update the board with the communicated

```
channel_invariant
  \msg.movePerm() **
  ([1\2]\sender.boardPerm()) **
  ([1\2]\receiver.boardPerm()) **
  \sender.oneMoveAheadOf(\msg, \receiver);
```

(a) Channel invariant of the communications in `while` loop

```
if (p1.turn && !p2.turn) {
  channel_invariant
   [1\4]\sender.boardPerm() **
   [1\4]\receiver.boardPerm() **
   \receiver.equalBoard(\sender);
  communicate p1: true -> p2;
}
```

(b) Communication at game end

Figure 5.10: Communications in TTT$_{\text{msg}}$ case study.

move after the communication. Each player always keeps `1\2` permission for his own board.

At each point in the game, only one of the players can read both players' boards, thus only one player is able to verify that the boards are equal. When the game ends, one of the players sends `1\4` permission for both boards to the other player. Figure 5.10b shows this for `p1`. This way, postcondition `p1.equalBoard(p2)` can be stated by both endpoints without `\chor`, and hence proven directly for the whole generated concurrent program.

TTT$_{\text{last}}$ We optimize away the `communicate` statement after the while loop, while still retaining correctness. We do this by introducing additional ghost state and reformulating the annotations to be more general. In doing so, we demonstrate a trade-off: run-time overhead can often be eliminated, at the cost of additional complexity in the contracts and ghost state.

Specifically, we use the following two extra fields to reference ghost state: `p1.lastPlayer` and `p2.lastPlayer`. These store a reference to the same object, which stores the mark of the "last player". In Fig. 5.11b we see that, just before communicating a move, `p1` checks if this move ended the game. If so, `p1.lastPlayer` is set to `p2.myMark`, because `p2` will be the last player updating his board, before the game ends. The predicate `p1.lastPlayerPerms()` specifies write permission to the `mark` field of its `lastPlayer` object. If the game is not finished yet, `p1` includes the full permission (`1\2` + `1\2`) in the channel invariant, so that `p2` may (possibly) edit it. Otherwise, if the game is finished, only `1\2` permission is sent, such that both players can read their `lastPlayer.mark` field to see who was the last

```
ensures (\endpoint p1;
  ([1\2]p1.lastPlayerPerms()) **
  p1.lastPlayer.mark == p1.myMark ==>
    ([1\2]p1.boardPerm() **
    ([1\2]p2.boardPerm()) **
    p1.equalBoard(p2));
```

(a) Postcondition of `run` method.

```
if (p1.gameFinished()) {
  p1.lastPlayer.mark = 1 - p1.myMark;
}
channel_invariant /* ... */
  ([1\2]\sender.lastPlayerPerms()) **
  (!\sender.gameFinished() ==>
   ([1\2]\sender.lastPlayerPerms()));
communicate p1.move.copy() -> p2.move;
```

(b) Code for marking the last player, and the channel invariant extension.

Figure 5.11: Adapted code of $\mathrm{TTT}_{\mathrm{last}}$ with respect to $\mathrm{TTT}_{\mathrm{msg}}$, stated for `p1`. It is symmetric for `p2`.

player. In Fig. 5.11a we show the adapted postcondition of `run`: the player whose mark is stored in its `lastPlayer.mark` field can ensure `p1.equalBoard(p2)`. This postcondition ensures that when both endpoints terminate in the generated implementation, VeyMont will conclude that whichever endpoint is the last player, there will always be one of them that will guarantee board equality. In this way, `p1.equalBoard(p2)` can be proven directly for the whole generated program.

5.6 Related Work

Besides the works we build upon by extending VeyMont [36, 80], the most similar research in the realm of VerCors is the work by Darabi et al. [31]. They introduce the `send` and `receive` statements to model loop dependencies. These statements allow sending permission to other iterations of a loop, and are similar to `communicate`. However, these statements are only supported inside loops, offer no support for sending a value, and conditional sends can only depend on variables not modified inside the loop.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*
[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*
[31]: Blom et al. (2015), *Verification of Loop Parallelisations*

Similar works in the area of choreographies are on logics to reason about the correctness of choreographies [41, 52]. These works could serve as a basis for formalizing the approach outlined in this chapter, but they would have to be extended with support for separation logic.

[41]: Carbone et al. (2010), *A Logic for Choreographies*
[52]: Cruz-Filipe et al. (2023), *Reasoning About Choreographic Programs*

We see interesting correspondences with multiple works on session types. Generally, session types do not support implementation generation. In theory, session type results may be transferable to choreographies, but this step is non-trivial.

Hinrichsen et al. introduce Actris, a Coq framework using Iris for correctness reasoning over session types [70]. Jacobs et al. [76] introduce similar but smaller formalization of dependent session protocols, also in Iris. Both approaches are powerful, but being Coq frameworks, lack the automation we aim for in this chapter. They could be good starting points for formalizing our approach.

[70]: Hinrichsen et al. (2020), *Actris: session-type based reasoning in separation logic*

[76]: Jacobs et al. (2023), *Dependent Session Protocols in Separation Logic from First Principles (Functional Pearl)*

Neykova et al. [112] present SPY, a tool that generates runtime monitors of user-defined constraints on exchanged messages and endpoint state. Our approach works without running the code, and introduces no overhead at runtime.

[112]: Neykova et al. (2013), *SPY: Local Verification of Global Protocols*

Bouma et al. [37] use VerCors to check conformance of Java programs to a multi-party session type (MPST). Specifically, they use permissions only at the implementation level, while we already use permissions at choreography-level.

[37]: Bouma et al. (2023), *Multiparty Session Typing in Java, Deductively*

Marques et al. present an approach to verify that C programs written using MPI [104] follow a protocol defined using a session type [99]. Their tool allows constraints to be expressed over messages sent and received, which is an extended version of session types. However, the constraints are limited to (in-)equalities of arithmetic expressions and variables, while we support general first-order logic expressions. The tool also has no support for shared memory or ghost state.

[104]: Message Passing Interface Forum (2021), *MPI: A Message-Passing Interface Standard Version 4.0* (link)

[99]: Marques et al. (2013), *Towards deductive verification of MPI programs against session types*

Zhou et al. [161] present Session*, a tool that extends the Scribble protocol language [158] with refinement types by compiling Session* protocols to F* [148], a functional programming language with refinement types. Because mutable memory is supported within the generated callbacks implemented in F* through an effect system, Session* supports a limited form of mutability indirectly. We support it generally, allowing sharing mutable memory across implementation callbacks and reasoning about it in contracts.

[161]: Zhou et al. (2020), *Statically verified refinements for multiparty protocols*

[158]: Yoshida et al. (2013), *The Scribble Protocol Language*

[148]: Swamy et al. (2016), *Dependent types and multi-monadic effects in F*

Swamy et al. [149] formalize a minimal 2-party session type framework as an example use of the SteelCore separation logic framework in F* [148]. They do not offer specialized support for correctness reasoning of session types or the

[149]: Swamy et al. (2020), *SteelCore: an extensible concurrent separation logic for effectful dependently typed programs*

transfer of resources via session types, beyond what F* offers natively. We foresee that our approach could be embedded in F* using SteelCore.

[33]: Bocchi et al. (2010), *A Theory of Design-by-Contract for Distributed Multiparty Interactions*

Bocchi et al. [33] present a formal framework for applying design-by-contract to session types. The "global assertions" from their work are similar to contracts in VeyMont choreographies. Besides the difference between session types and choreographies, Bocchi et al. also do not support shared memory. They do define well-assertedness of global assertions to e.g. prevent endpoints from using values they do not know about. We resolve this by using permission stratification.

[128]: Proust et al. (2023), *Verified Scalable Parallel Computing with Why3*

[32]: Bobot et al. (2011), *Why3: Shepherd Your Herd of Provers*

Finally, Proust et al. [128] have integrated the Why3 [32] program verifier with the Bulk Synchronous Parallel (BSP) model. The version of BSP in this work shares some aspects with OpenMP, as it offers parallelized versions of common operations, such as map and fold. In addition, BSP offers choreography-like many-to-many communication. There are two differences with our work. First, code written using the BSP API can only be executed in an environment that provides such an API. VeyMont generates plain Java & PVL code that can be verified and only needs the standard library. Second, Proust et al. only consider purely functional programs, while VeyMont supports reasoning about mutable variables and shared memory.

5.7 Conclusion

VeyMont could already verify choreographies, auto-generate permissions, and use the endpoint projection to generate an implementation. In this work, we added endpoint ownership annotations and channel invariants to VeyMont, such that choreographies can specify concurrent programs with shared memory between threads. Additionally, we transfer verification annotations to the generated implementations, such that they can be verified directly, without the choreography. We showed the new capabilities of extended VeyMont in case studies.

For future work, we first of all aim to introduce parameterized endpoints, such that distributed systems with any n

number of nodes can be formulated as choreography. Also, adding support for one-to-many or many-to-one communications would make VeyMont more expressive. While we now use verification of choreographies and the generated implementations to ensure correctness of the projection, we would also like to formalize our approach, i.e. extend [80]. Finally, by doing more case studies, we will validate our approach more extensively.

[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

Verified Parameterized Choreographies 6

Choreographies are useful for modelling systems with multiple simultaneously executing and communicating participants, e.g. distributed systems. As described in previous chapters, VeyMont can verify correctness of choreographies and generate verifiably correct code that implements the choreography, narrowing the gap between design and implementation of distributed systems. Up until now, it supported only fixed sets of participants. However, realistic systems are often *parameterized*: they scale according to some parameter N. In this chapter we extend VeyMont with *parameterized choreographies*, making VeyMont more usable for realistic case studies. Specifically, we add parameterized primitives such as *participant families* and *parameterized communication*. We encode these primitives using a structured parallelism primitive from the underlying verifier VerCors, and by using conditionals in the endpoint projection, partially delaying projection until run time. We illustrate the encoding with a *distributed summation* choreography, and prove it correct with VerCors.

Contents

6.1 Introduction . . . 115
6.2 Choreographies . 119
6.2.1 Syntax 119
6.2.2 Deadlock Freedom 123
6.2.3 Shared Memory . 124
6.2.4 Running Example 125
6.3 Choreographic Projection 126
6.3.1 Basic Example . . 127
6.3.2 Basic Rules 128
6.3.3 Parameterization 129
6.3.4 Rules 129
6.3.5 Branch Unanimity 131
6.3.6 Running Example 132
6.4 Endpoint Projection 133
6.4.1 Basic Example . . 133
6.4.2 Basic Rules 134
6.4.3 Parameterization 134
6.4.4 Running Example 136
6.5 Related Work . . 136
6.6 Conclusion 139

6.1 Introduction

Distributed systems are not just ubiquitous, they are indispensable for networked systems on a global scale. Unfortunately, guaranteeing robustness of distributed systems is

This chapter is based on "Verified Parameterized Choreographies" by Rubbens, Van den Bos, and Huisman [135], published at COORDINATION 2025.

```
1 choreography summation2() {
2   endpoint a = Node(int());
3   endpoint b = Node(int());
4   run {
5     communicate a.sum -> b.in;
6     communicate b.sum -> a.in;
7     a.update(); b.update();
8   }
9 }
```

(a) For two endpoints `a` and `b`

```
choreography summationN(int N) {
  endpoints nodes[i := 0..N] = Node(i, int());
  run {
    while ((\endpoints nodes[i := 0..N]; nodes[i].n < N-1)) {
      communicate nodes[i := 0..N-1].sum -> nodes[i+1].in;
      communicate nodes[N-1].sum -> nodes[0].in;
      nodes[i := 0..N].update();
    }
  }
}
```

(b) Parameterized for `N` endpoints

Figure 6.1: Distributed summation choreographies

still a challenge. Consider a participant of a distributed system, waiting for a message that will never be sent. Clearly, this system cannot function reliably. This type of bug is called a *communication deadlock*, and ideally a distributed system would be free of deadlocks. Another aspect of robustness is that of functional correctness: maybe the distributed system never deadlocks, but does the system actually compute the *correct* result?

[105]: Montesi (2023), *Introduction to Choreographies*

An approach to improve the reliable development of distributed systems is to use the *top-down* formalism of *choreographies* [105]. In its purest form, a choreography is a series of message exchanges between participants, called *endpoints*. Choreographies have two primary properties [105]. The first is communication deadlock freedom: no endpoint will be stuck waiting for a message that will never be sent. The second is *message fidelity*: an endpoint will never receive a message of a different type than it is expecting. Choreographies also support the *endpoint projection*, which generates an implementation for a given endpoint.

Figure 6.1a shows an example of a choreography that sums the values of two endpoints. Lines 2 and 3 declare the endpoints `a` and `b` of type `Node`, and initialize their `sum` fields with a random integer. On lines 5 and 6 each endpoint sends their local sum to the `in` field of the other. On line 7, they update their `sum` fields with the sum of their initial value and the value of the `in` field. The `sum` field of each endpoint now contains the sum of both the initial values. As an example in Java-like syntax, the endpoint projection of Fig. 6.1a for endpoint `b` is:

```
b.in = chan_ab.readValue();
chan_ba.writeValue(b.sum);
b.update();
```

To verify choreographies like Fig. 6.1a, VeyMont was developed, a verifier and code generator for choreographies [36]. It supports functional correctness verification of choreographies with contract annotations, such as pre- and postconditions and asserts. When generating code with VeyMont, verification annotations are preserved [133], which means correctness of the generated code can be established independently from the initial choreography. This allows safe modification of generated code. VeyMont is built on top of VerCors, a deductive verifier for concurrent and parallel software. Besides languages such as Java and C, VerCors also supports the internal Prototypical Verification Language (PVL), a Java-like language intended for rapid prototyping of verification features.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

[133]: Rubbens et al. (2024), *VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*

To verify a choreography, VeyMont applies the *choreographic projection*, which transforms a choreography into a PVL program that combines the behaviour of *all endpoints* into a *single program* [36]. This is in contrast to the endpoint projection, which slices a choreography in such a way that only the parts relevant for one specific endpoint remains. The choreographic projection has two goals: 1) to make the choreography verifiable with an off-the-shelf program verifier like VerCors, and 2) to add annotations for correctness aspects such as deadlock freedom and memory safety. The first goal is achieved by modelling communication with regular assignment, and preserving composite statements such as `if` and `while`. To illustrate, the choreographic projection for Fig. 6.1a is:

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

```
b.in = a.sum; a.in = b.sum;
a.update(); b.update();
```

The second goal is achieved by encoding correctness aspects into PVL [36, 133]. Then, if the projection is verified, the choreography respects its contracts [80].

[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

However, verifying regular choreographies is not enough. Instead, realistic case studies often scale with some parameter N, and hence require *parameterized choreographies.* In Fig. 6.1b a distributed sum choreography is parameterized by N (line 1). Instead of defining endpoints individually, line 2 defines an *endpoint family*, which is a range of endpoints, its size determined by a symbolic expression. The choreography in Fig. 6.1b generalises the approach from Fig. 6.1a: for $N-1$ rounds, each node will send its partial sum to a neighbouring node, as done on lines 5 and 6. Then, each node will update its partial sum (line 7), after which the while loop will repeat. When the while loop terminates, each node will know the sum of all initial values.

Parameterization of *both* the choreographic and endpoint projection is still an open problem. To enable *verification* and *code generation* for choreographies like Fig. 6.1b, this chapter discusses how to extend VeyMont with parameterization.

Contributions We define choreographies with parameterization by adding parameterized primitives to the choreographic language of VeyMont (Section 6.2). In particular, we add *endpoint families*, which are ranges of endpoints with their size defined by a symbolic expression. We also add a *parameterized communication statement*, which communicates a message according to a user-defined one-to-one mapping between two possibly overlapping endpoint family ranges.

To verify parameterized choreographies, we extend the choreographic projection to use a structured parallelism primitive, the `par` block, to encode the semantics of the parameterized communication statement (Section 6.3). We identify a fragment of the choreographic language for which memory safety annotations can be automatically generated, preserving full automation of the verification process. E.g. we limit the syntax of the parameterized communication statement such that it may only access memory in a certain pattern. Other parameterized syntax, such as endpoint families and parameterized expressions, can be encoded using mathematical sequences and universal quantifiers. We also extend pre-existing VeyMont features (deadlock freedom and shared memory [36, 133]) to support parameterization.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*
[133]: Rubbens et al. (2024), *VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*

To support parameterization, the endpoint projection must generate an implementation for an unknown but fixed number of endpoints (Section 6.4). This is important, because in a parameterized choreography, the sizes of endpoint families are defined by symbolic expressions. We make part of the endpoint projection conditional on the runtime value of endpoint family indexing expressions, effectively delaying projection until *run-time*. That way, instead of generating one implementation *per* endpoint, we generate an implementation for a *range* of endpoints, which behaves based on the run-time value of the current endpoint index. This encoding provides executable code with preserved verification annotations, allowing deductive verification separate of the choreography.

We illustrate the choreographic and endpoint projection with the distributed sum running example, which we have proven correct with VerCors in the artifact [139]. Finally, we discuss related work (Section 6.5).

[139]: Rubbens et al. (2025), *Artefact of: Verified Parameterized Choreographies*

6.2 Choreographies

We will next define the syntax for parameterized choreographies, as well as give an intuition for the semantics (Section 6.2.1). The parameterization extension also requires integration with existing *deadlock freedom* [36] and *shared memory* [133] support of VeyMont (Sections 6.2.2 and 6.2.3). Finally, we introduce the full distributed summation running example (Section 6.2.4).

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

[133]: Rubbens et al. (2024), *VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*

6.2.1 Syntax

The syntax for choreographies in VeyMont is shown in Fig. 6.2. The description in this section follows the order in this figure. We also informally describe its semantics. Sections 6.3 and 6.4 describe the actual semantics of choreographies by defining two transformations into the OOP fragment of PVL.

Choreographies are top-level definitions located in the same scope as PVL classes. Therefore, choreographies can reference types and call methods from PVL classes. E.g. in Fig. 6.1, endpoints are declared using the `Node` PVL class. In addition, in Fig. 6.2 the following syntax elements are

$$
\begin{array}{rl}
e, a, b & ::= \text{endpoint} \\
F, G & ::= \text{endpoint family} \\
\textbf{chor} & ::= K \ \texttt{choreography(}\overline{T\ v}\texttt{) \{}\ \ \overline{D_{chor}}\texttt{\}} \\
D_{chor} & ::= K \ \texttt{run \{}\ S_{\texttt{chor}}\ \texttt{\}} \mid \texttt{endpoint}\ e = C\texttt{(}\overline{H}\texttt{);} \\
 & \mid \texttt{endpoint}\ F\texttt{[}v\ \texttt{:= 0 ..}\ H\texttt{]}\ = \ C\texttt{(}\overline{E}\texttt{);} \\
r, p & ::= e \mid F\texttt{[}E\texttt{]} \\
\alpha, \beta & ::= r \mid F\texttt{[}v\ \texttt{:=}\ E\ \texttt{..}\ E\texttt{]} \\
S_{chor} & ::= \texttt{if (}H_{chor}\texttt{)}\ S_{chor}\ S_{chor} \mid \texttt{assert}\ R_{chor}\texttt{;} \mid \texttt{endpoint}\ \alpha\texttt{:}\ S_{ep} \\
 & \mid \texttt{loop\_invariant}\ R_{chor}\texttt{; while (}H_{chor}\texttt{)}\ S_{chor} \\
 & \mid \texttt{channel\_invariant}\ R_{chan}\texttt{; communicate}\ \alpha\texttt{:}\ H\ \texttt{->}\ \alpha\texttt{:}\ H\texttt{;} \\
S_{ep} & ::= H\texttt{.}m\texttt{(}\overline{H}\texttt{);} \mid H\ \texttt{:=}\ H\texttt{;} \\
H_{chor} & ::= \texttt{(\textbackslash endpoint}\ \alpha\texttt{;}\ H\texttt{)} \mid H_{chor}\ \texttt{\&\&}\ H_{chor} \\
R_{chor} & ::= \texttt{(\textbackslash endpoint}\ \alpha\texttt{;}\ R\texttt{)} \mid R_{chor}\ \texttt{**}\ R_{chor} \mid H_{chor} \\
R_{chan} & ::= \text{Inline}\ R \mid \texttt{\textbackslash msg} \mid \texttt{\textbackslash sender} \mid \texttt{\textbackslash receiver}
\end{array}
$$

Figure 6.2: Choreographic fragment

reused from the PVL grammar from Chapter 2: K is a contract, E is a pure expression (no heap access and no side-effects), H is equal to E except it might read heap locations, and R is equal to H except it can also contain permissions and predicates.

Declarations A choreography consists of a contract, a series of arguments, and a series of choreographic declarations, which can be an endpoint, an endpoint family, or a `run` declaration. A single endpoint has a name (e, a, or b), a class type C and an argument list. The argument list is passed to the constructor at run-time, which creates an instance of C to represent the endpoint.

A parameterized endpoint is an *endpoint family*, which additionally has a size parameter. Note that this parameter can be *symbolic*. Therefore, to verify or generate code for parameterized choreographies, one must either use e.g. an SMT solver that can do so symbolically, or somehow delay inspection of this size until run-time, when the symbolic parameter is instantiated. The parameter is also allowed to depend on the heap. This is because the parameter is evaluated before the endpoint family is initialized. When each member of an endpoint family is constructed at run-time, the binder v is also in scope, which contains the index of the current endpoint within the endpoint family. The endpoint family is represented at run-time as a sequence of instances of C. An example of a parameterized endpoint is shown in Fig. 6.1b on line 2.

The `run` declaration has a contract and a series of choreographic statements. Essentially, each endpoint executes the `run` declaration by only executing the choreographic statements related to the endpoint. The contract of the `run` declaration differs from the choreography contract as follows: the choreography precondition holds before endpoint initialization, the `run` precondition holds after initialization and before `run` is executed. Conversely, the `run` postcondition holds when an endpoint finishes, the choreography postcondition holds after *all* endpoints have finished. Examples of `run` declarations are in Figs. 6.1 and 6.3.

Endpoint references There are two notations for endpoint references: r and α. The notation r refers to a particular endpoint, which can be be either a singular endpoint e, or a family F indexed by a pure expression. E.g. F`[N-1]` selects the last endpoint of a family F of size N. The notation α extends r with ranges of endpoint families as follows: F`[`i `:=` E_l `..` E_h`]`, for a family F, a binder i, and a half-open range $[E_l, E_h)$. E.g. F`[tid := N/2..N]` refers to all endpoints of F with index $\in [\frac{N}{2}, N)$. In this case, the family has to have at least size N. The binder i is also used in endpoint expressions, explained later. This notation is inspired by related work of Ng et al. [114], as discussed in Section 6.5.

[114]: Ng et al. (2015), *Pabble: parameterised Scribble*

Indexing into endpoint families using α is only allowed with pure expressions E. This is deliberate, as it ensures any indexing operation is heap independent, meaning no permission annotations are necessary for endpoint family indexing. This also ensures any indexing operation can be executed by any endpoint of a choreography. This allows evaluating indexing expressions in any endpoint context, which is important for communication statements, discussed later.

Statements Branches, asserts and loops are *choreographically transparent*: they are only executed by the endpoints that occur within them. If an endpoint is not mentioned in a choreographically transparent statement, the endpoint skips it.

An endpoint statement is a local action of an endpoint. It requires an α, meaning the action can apply to a singular endpoint or a family range. We allow method calls and assignments on endpoints. For example, in the statement "`endpoint` e`:` e`.`m`()`" the method call e`.`m`()` will be executed by endpoint e. The formal syntax for endpoint statements is slightly more general than what the choreographic and endpoint projection can handle. For example, parameterized endpoint statements can only do method calls directly on endpoints. We enforce these restrictions syntactically, and further discuss them in Section 6.3.

A `communicate` statement specifies communication. Through the α notation, communication can either be between two singular endpoints, or between two family ranges with an injective mapping. Injectivity is ensured by both the choreographic and endpoint projection through explicit checks. Communication statements consists of a channel invariant, a sending endpoint, the message to be sent, the receiving endpoint, and the destination in which the message will be received. The channel invariant specifies a property over the message, i.e. an invariant over values in the channel. The endpoint projection (Section 6.4) creates a channel for each `communicate` [36, 133]. The primitives `\sender`, `\receiver` and `\msg` may be used to refer to the sender, receiver, and message respectively in the channel invariant. For example, `channel_invariant \msg > 2` specifies all messages sent over the channel must be bigger than 2. Examples of singular and parameterized `communicate` statements are shown in Fig. 6.1.

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*
[133]: Rubbens et al. (2024), *VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*

For both endpoint and communication statements, whenever the endpoint annotations are obvious, they are omitted. E.g. in `communicate a.x -> b.y`, we omit the `a:` and `b:` annotations.

Expressions We distinguish two kinds of choreographic expressions. The first expression type is H_{chor}, which is essentially a list of endpoint expressions, composed using `&&`. An endpoint expression is an expression tagged with the endpoint that should evaluate it. If the endpoint expression introduces a binder, this binder can appear inside the tagged expression. E.g. consider this expression:

```
(\endpoint F[i := 0 .. N]; F[i].x == i)
```

This states that endpoints in family F have a field `x` equal to the endpoint index. In addition, when an endpoint evaluates the endpoint expression, it must do so using only its own memory. This is further explained in Section 6.2.3. The second expression type is R_{chor}, which is similar to H_{chor} except that R_{chor} can also introduce permissions, and compose them with the separating conjunction `**`.

6.2.2 Deadlock Freedom

Deadlocks occur if an endpoint is waiting for a message that will never be sent. This can happen when branches are involved. Consider the choreography `communicate a.x -> b.y; ...`, which is just a series of communications and local actions. Because of the simple structure of this choreography, each send is guaranteed to be paired with a receive, and hence it cannot deadlock. Now consider the choreography `if (a.x && b.x) communicate a.y -> b.y`, which consists of one branch and one communicate. Here, a deadlock is possible: when `!a.x && b.x` holds, `b` will enter the body of the `if` statement, and `a` will skip it. This is because of the semantics of choreographic expressions, where `a` will only execute expressions relevant to `a`, and vice versa for `b`. This will result in `b` waiting for a message, even though it will never be sent, because `a` skipped it.

To prevent deadlocks, choreographies need branch unanimity [36]. A branch is unanimous if the condition of a branch evaluates to the same value for each participating endpoint. In other words, all endpoints have to agree “unanimously” on the condition of the branch. VeyMont checks this automatically [36]. Taking the previous example, the branch is unanimous if `a.x == b.x` always holds before the `if`. Branch unanimity also applies to `while` loops, and hence

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

ensures that either both endpoints encounter the communicate statement, or they both skip it, and not something in between. For a finite set of endpoints, a verification condition for branch unanimity is straightforward to generate, following the example above. We extend branch unanimity for endpoint families in Section 6.3.5.

6.2.3 Shared Memory

Supporting shared memory in choreographies requires memory safety annotations. For example, the statement `endpoint a: b.x := y` (`a` assigns `y` to `b.x`) is safe *if and only if* endpoint `a` has write permission for `b.x`.

VeyMont supports *endpoint ownership annotations* to bind a permission to an endpoint [133]. E.g. `(\endpoint a; Perm(b.x, 1))` states that `a` needs write permission for `b.x`. VeyMont checks if endpoints in a choreography do not use memory that they do not own by transforming the endpoint ownership annotations into PVL permissions. This is done as part of the choreographic projection [133]. Summarizing, the permission from the previous example would be transformed into `Perm(b.x, 1, a)`, using a special encoding to put the endpoint owner as metadata in the permission. The transformation also guards all field accesses as follows. First, the transformation generates the function `read_f`:

[133]: Rubbens et al. (2024), *VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*

```
requires Perm(o.f, 1, e);
int read_f(endpoint e, object o) = o.f;
```

The precondition of `read_f` forces permission with proper metadata to be available at every field access. The transformation also replaces every field access with an invocation of `read_f`. E.g. the expression `(\endpoint a; b.f > 0)` is transformed into `read_f(a, b) > 0`.

The shared memory support of VeyMont is straightforward to integrate with parameterization, because parameterized permissions can be encoded using universal quantifiers. This is sufficient, as quantifiers are natively supported by the underlying verifier VerCors. The only integration required is for the choreographic projection to explicitly apply the

```
requires N >= 2;
choreography summation(int N) {
  endpoints ns[i := 0..N] = Node(i, int());
  requires (\endpoints ns[i := 0..N]; ns[i].sum == ns[i].v);
  ensures (\endpoints ns[i := 0..N]; ns[i].sum == sum(ns, 0, ns[i].n));
  run {
    invariant (\endpoints ns[i := 0..N]; ns[i].sum == sum(ns, ns[i].i, ns[i].n));
    while ((\endpoints ns[i := 0..N]; ns[i].n < N-1)) {
      channel_invariant \msg == sum(ns, \sender.i, \sender.n);
      communicate ns[i := 0..N-1].sum -> ns[i+1].in;
      channel_invariant \msg == sum(ns, \sender.i, \sender.n);
      communicate ns[N-1].sum -> ns[0].in;
      ns[i := 0..N].update(); } } }
```

Figure 6.3: Distributed summation choreography

shared memory encoding. This is done by using the *confined memory mode* operator, introduced in Section 6.3.

6.2.4 Running Example: Distributed Summation

We now formulate a ring-based distributed summation algorithm as a choreography. It will also be used to illustrate the choreographic projections.

Algorithm encoding In the choreography in Fig. 6.3, each endpoint only communicates with its two neighbouring endpoints (resp. predecessor and successor), simulating a ring topology of size N. An endpoint initially knows only its own value. The goal is that each endpoint eventually knows the sum of the values of all endpoints. The sum is calculated using the `sum` function, with parameters: a sequence of nodes, the starting index and the number of indices to include in the sum. At each iteration, the endpoints send the current partial total to their successor. Then, they receive a partial total from their predecessor, and add their own local value to it. This yields a new total. After looping $N-1$ times, each endpoint knows the network total.

A key difference between the algorithm and the choreographic encoding is how the network structure is encoded. Instead of using the modulo operator, we apply an insight from previous work and "linearize" the ring communication into two separate communications [56, 114]. One is parameterized over the range 0 to $N-1$, and the other communication is from $N-1$ to 0, closing the loop. This shows

[56]: Deniélou et al. (2012), *Parameterised Multiparty Session Types*
[114]: Ng et al. (2015), *Pabble: parameterised Scribble*

that circular topologies can be encoded using simpler linear structures.

Verification outline The choreography contains several verification annotations, which give an outline of the correctness proof. Essentially, as partial sums are communicated between endpoints, the partial sum of each endpoint converges towards the true total in $N - 1$ iterations.

Permission annotations, ghost state and proof steps necessary to verify Fig. 6.3 are omitted for ease of presentation. In particular, verification requires a lemma that uses the symmetry of addition to show that a sum starting at endpoint `ns[`i`]` equals the sum starting at endpoint `ns[`j`]`. The version with full verification annotations is available in the artifact [139]. In particular, we verified that each endpoint computes the same network total.

[139]: Rubbens et al. (2025), *Artefact of: Verified Parameterized Choreographies*

6.3 Choreographic Projection

We will now discuss the choreographic projection operator $\{\!\!\{\cdot\}\!\!\}$. Its purpose is to encode the choreography into a PVL program that VerCors can verify. This way, if the PVL program verifies, the choreography is correct. Otherwise, there might be a bug, either in the choreography, or in its specification.

To encode parameterized constructs, we use two primitives from the underlying verifier VerCors: the `par` block for structured parallelism and universal quantifiers. We impose restrictions on the allowed syntax to ensure the required annotations can be generated automatically. For non-parameterized constructs, the encoding transforms all `communicate` statements into plain assignments, and keeps other primitives, i.e. `if`, `while`, assignments and method calls. Effectively, the choreographic projection picks a representative interleaving of all possible interleavings of choreography, and then encodes it in plain PVL. This is sound, as each endpoint is also verified to be memory safe. Memory safety guarantees non-interference, and therefore the behaviours of all interleavings are equivalent.

```
communicate a: a.balance > a.n -> b: b.ok;

if (a.balance > a.n && b.ok) {
  communicate a.n -> b.n
  a: a.balance := a.balance - a.n;
  b: b.balance := b.balance + b.n; }
```

(a) Input choreography.

```
b.ok = a.balance > a.n;
assert a.balance > a.n == b.ok;
if (a.balance > a.n && b.ok) {
  b.n = a.n;
  a.balance = a.balance - a.n;
  b.balance = b.balance + b.n; }
```

(b) Output PVL.

Figure 6.4: Encoding of a choreography that models a bank transfer.

The choreographic projection operator has two modes. The plain mode, written as $\{\!\!\{\cdot\}\!\!\}$, encodes the choreography as a sequential object-oriented PVL program, while adding additional checks for deadlocks. The confined memory mode, written as $\{\!\!\{\cdot\}\!\!\}_r$, ensures the argument is encoded such that only memory of endpoint r is used (see Section 6.2.3). The confined memory mode is used by the plain mode when an endpoint context annotation occurs.

We only list the rules that are key to transforming the examples. The full listing can be found in [136]. Note that the choreographic projection results in a program intended for *verification*. The resulting program is an abstracted version of the choreography, which behaves as if all endpoints are sharing one thread, yet does not exclude any concurrent behaviours. Transformation for the purpose of execution is done by the endpoint projection explained in Section 6.4.

[136]: Rubbens et al. (2025), *Verified Parameterized Choreographies Technical Report*

6.3.1 Non-Parameterized Example

To give an intuition for the choreographic projection, we first show a concrete example. Figure 6.4a shows the input choreographic code. Figure 6.4b shows the output PVL code, created using the choreographic projection $\{\!\!\{\cdot\}\!\!\}$. There is a close correspondence between the left and the right listing: each statement is encoded using the corresponding rule from Fig. 6.5. The only new statement is the `assert`, added by rule CpIf, which checks deadlock freedom (see Section 6.2.2).

CpIf

$$\{\!\{\texttt{if (}H\texttt{)}\ S_{\texttt{true}}\ S_{\texttt{false}}\}\!\} = \begin{array}{l}\texttt{\{ assert unanimous(}H\texttt{);}\\ \quad\texttt{if (}\{\!\{H\}\!\}\texttt{)}\ \{\!\{S_{\texttt{true}}\}\!\}\ \{\!\{S_{\texttt{false}}\}\!\}\ \texttt{\}}\end{array}$$

CpAssign

$$\{\!\{r\texttt{:}\ H_{loc}\ \texttt{:=}\ H_v\texttt{;}\}\!\} = \{\!\{H_{loc}\}\!\}_r\ \texttt{=}\ \{\!\{H_v\}\!\}_r\texttt{;}$$

CpComm

$$\left\{\!\!\left\{\begin{array}{l}\texttt{channel\_invariant}\ R_I(\texttt{\textbackslash msg, \textbackslash sender, \textbackslash receiver});\\ \texttt{communicate}\ r\texttt{:}\ H_{msg}\ \texttt{->}\ p\texttt{:}\ H_{dst}\texttt{;}\end{array}\right\}\!\!\right\} =$$

$$\begin{array}{l}\texttt{\{}\ T\ v\ \texttt{=}\ \left\{\!\!\left\{H_{msg}\right\}\!\!\right\}_r\texttt{;}\\ \quad\texttt{exhale}\ \{\!\{R_I(v,\ r,\ p)\}\!\}_r\texttt{;}\\ \quad\texttt{inhale}\ \{\!\{R_I(v,\ r,\ p)\}\!\}_p\texttt{;}\\ \quad\{\!\{H_{dst}\}\!\}_p\ \texttt{= v; \}}\end{array}$$

Figure 6.5: Non-parameterized choreographic projection rules

6.3.2 Non-Parameterized Projection Rules

We will now discuss the rules required to transform the example in Fig. 6.4, shown in Fig. 6.5. Rule CpAssign pattern matches on the subparts of the choreographic assignment statement on the left side of =, and shows how to construct the projected statement on the right. In this case, the choreographic projection is applied to H_{loc} and H_v, removing the endpoint labels from the resulting statement. Within rules, subscripts such as *loc* and *v* are only to clarify intention. However, for $\{\!\{\cdot\}\!\}_r$, and later $[\![\cdot]\!]_r$, the subscript *is* significant: $\{\!\{\cdot\}\!\}_r$ enables the confined memory mode, confining the expressions to the memory owned by r. Note that this rule is only applicable for any endpoint r, meaning singular endpoints e as well as an indexed family `F[i]`.

The rule CpIf adds a deadlock freedom assert, and then forwards the choreographic projection to the subexpression and sub-statements. The unanimous transformation function, which computes a verification condition for deadlock freedom, is further discussed in Section 6.3.5. As there is no endpoint context on the sub-statements, the confinement memory mode is not used here.

Rule CpComm encodes the sending of message H_{msg} to location H_{dst}, while transferring the channel invariant R_I from r to p. This is done as follows. First, the message value

```
channel_invariant
  \msg == sum(ns, \sender.tid, \sender.n);
communicate
  nodes[i := 0..N-1].sum -> nodes[i+1].in;
```

(a) Input communication

```
assert
  (\forall int i, j = 0..N-1;
    i+1 == j+1 ==> i == j);
par (int i = 0..N-1)
  context Perm(ns[i].sum, ϵ, ns[i]) **
    Perm(ns[i+1].in, 1, ns[i+1])
  requires ns[i].sum == sum(ns, ns[i].tid, ns[i].n);
  ensures ns[i+1].in == sum(ns, ns[i].tid, ns[i].n);
{ int v = ns[i].sum;
  exhale ...; inhale ...;
  ns[i+1].in = v; }
```

(b) Output PVL

Figure 6.6: Encoding of a parameterized communicate from Fig. 6.3

v is computed, confined to the memory of r. Then, the permissions in the channel invariant are removed using `exhale`, using the confined memory mode to ensure only permissions of r are removed. To determine the permissions to be exhaled, a substitution operation is applied. The notation $R_I(v,\ r,\ p)$ replaces, in R_I, every occurrence of `\msg` with v, `\sender` with r and `\receiver` with p. E.g. `\msg > 0` would become v `> 0` after substitution. This substituted invariant is added to the state of p using `inhale`, and then the value is written to the destination location.

6.3.3 Parameterized Example

Figure 6.6 shows an example application of rule CpCommRange (Fig. 6.7). Note the use of the three-argument `Perm` predicate to indicate permissions with additional metadata (see Section 6.2.3). The `inhale`/`exhale` statements respectively repeat the `requires`/`ensures` expressions, and are hence abbreviated in this figure.

6.3.4 Parameterized Projection Rules

The transformation rules for parameterized choreographies are shown in Fig. 6.7. Rule CpCommRange encodes a parameterized communication between two endpoint families. It is essentially a regular communication, wrapped in a `par` block. This is crucial: if all message transfers can happen independently in parallel, they can be safely split up into separate threads, as is done by the endpoint projection. Before the `par` block, an `assert` is generated which checks injectivity of the expression d over the given range $[E_l, E_h)$ of the `communicate`. This assert ensures the sender-receiver relation is injective, i.e. for each sender there must be exactly one receiver, and vice versa.

Note how the syntax of message and destination fields is restricted: fields can only be dereferenced on an indexed family F`[`i`]`. This is in contrast with rule CpComm, where the object is a heap-dependent expression. This restriction ensures the entire process of projection stays automatic. Allowing a heap expression H would require user annotations. A workaround for the restriction is to assign a heap-dependent expression in a preceding method call.

In the generated par block, we require ϵ permission to read the message field f, which means the verifier will pick a positive fraction smaller than the available permission. Using ϵ ensures the location can only be read, and not written to, which allows the verifier to maintain that f does not change. The `context` keyword here is syntactic sugar for a symmetric `requires` and `ensures` clause. Rule CpCommRange performs substitution on R_I like rule CpComm. E.g. in the precondition of the `par` block, the notation $R_I(\ldots)$ replaces, in R_I, `\msg` with F`[`i`].`f, `\sender` with F`[`i`]` and `\receiver` with G`[`$d(i)$`]`.

Rule CpExprRange shows how to project an endpoint expression with a range. Essentially, `\endpoint` is replaced with a universal quantifier, and the inner expression E is confined to the memory accessible to F`[`i`]`.

Rule CpMethodCallRange encodes that a method call is executed in parallel on a range of an endpoint family. It does this by projecting the method call confined to a symbolic element of this endpoint family, F`[`i`]`, and wrapping that in a PVL `par` block. Wrapping the method call in a `par` block encodes that the methods must run in parallel and independently. The contract for the `par` block is taken from the method using pre(m, F`[`i`]`) and post(m, F`[`i`]`) to return the pre-/postcondition of m. They also replace any occurrence of `this` in the return value with the second argument, in this case F`[`i`]`. Note that only method calls directly on F`[`i`]` are allowed, similar to rule CpCommRange. This is to keep projection automatic. A workaround for this restriction is to compute a heap-dependent expression within

```
CpMethodCallRange
$\llbracket$endpoint $F$[$i$ := $E_l$ .. $E_h$]: $F$[$i$].$m$();$\rrbracket$ =
    par (int $i$ = $E_l$ .. $E_h$)
      requires $\llbracket \mathsf{pre}(m, F[i]) \rrbracket_{F[i]}$;
      ensures $\llbracket \mathsf{post}(m, F[i]) \rrbracket_{F[i]}$;
    { $\llbracket$endpoint $F$[$i$]: $F$[$i$].$m$();$\rrbracket$ }

CpExprRange
$\llbracket$(\endpoint $F$[$i$ := $E_l$ .. $E_h$]; $E$)$\rrbracket$ =
  (\forall int $i$ = $E_l$ .. $E_h$; $\llbracket E \rrbracket_{F[i]}$)

CpCommRange
$\left\llbracket \begin{array}{l} \texttt{channel\_invariant } R_I(\backslash\texttt{msg}, \backslash\texttt{sender}, \backslash\texttt{receiver}); \\ \texttt{communicate } F[i := E_l \texttt{ .. } E_h]\texttt{: } F[i].f \texttt{ -> } G[d(i)]\texttt{: } G[d(i)].g; \end{array} \right\rrbracket$ =
  assert (\forall int i, j = $E_l$ .. $E_h$; $d$(i) == $d$(j) ==> i == j);
  par (int $i$ = $E_l$ .. $E_h$)
    context $\llbracket$Perm($F$[$i$].$f$, $\epsilon$)$\rrbracket_{F[i]}$ ** $\llbracket$Perm($G$[$d(i)$].$g$, 1)$\rrbracket_{G[d(i)]}$;
    requires $\llbracket R_I(F[i].f, F[i], G[d(i)]) \rrbracket_{F[i]}$;
    ensures $\llbracket R_I(G[d(i)].g, F[i], G[d(i)]) \rrbracket_{G[d(i)]}$;
  { $T$ $v$ = $\llbracket F[i].f \rrbracket_{F[i]}$;
    exhale $\llbracket R_I(v, F[i], G[d(i)]) \rrbracket_{F[i]}$;
    inhale $\llbracket R_I(v, F[i], G[d(i)]) \rrbracket_{G[d(i)]}$;
    $\llbracket G[d(i)].g \rrbracket_{G[d(i)]}$ = $v$; }
```

Figure 6.7: Parameterized choreographic projection rules

a method m, and then calling the desired method on the result within m.

6.3.5 Branch Unanimity

Branch unanimity is defined through the function unanimous(E) and supporting functions, shown in Fig. 6.8. It evaluates to `true` if all endpoints in E evaluate E to the same result. We split this into two cases: either all α_i evaluate E to `true`, or all evaluate to `false`. Evaluating the condition for each individual endpoint occurring in E takes two steps. The first step is using the confined memory mode, which drops parts of the expression that are not relevant to the given endpoint. For example:

$\llbracket$`(\endpoint a;` E_1`) && (\endpoint b;` E_2`)`$\rrbracket_{\mathsf{a}} = \llbracket E_1 \rrbracket_{\mathsf{a}}$

The second step is wrapping E, confined to $F[i]$, in a uni-

$$\mathsf{unanimous}(E) = \overbrace{(\mathsf{ground}(E,\ \alpha_1,\ \mathtt{true})\ \mathtt{\&\&}\ \cdots)}^{\text{for all } \alpha_1,\ldots,\alpha_n \in E}\ \mathtt{||}\ \overbrace{(\mathsf{ground}(E,\ \alpha_1,\ \mathtt{false})\ \mathtt{\&\&}\ \cdots)}^{\text{for all } \alpha_1,\ldots,\alpha_n \in E}$$

$$\mathsf{ground}(E,\ r,\ b) = \{\!\!\{E\}\!\!\}_r\ \mathtt{==}\ b$$

$$\mathsf{ground}(E,\ F\mathtt{[}i\ \mathtt{:=}\ E_l\ \mathtt{..}\ E_h\mathtt{]},\ b) = (\mathtt{\backslash forall\ int}\ i\ \mathtt{=}\ E_l\ \mathtt{..}\ E_h\mathtt{;}\ \{\!\!\{E\}\!\!\}_{F\mathtt{[}i\mathtt{]}}\ \mathtt{==}\ b)$$

Figure 6.8: Functions for constructing the branch unanimity condition

```
 1 loop_invariant (\forall int i = 0..N; ns[i].n < N-1 == true) || ...;
 2 loop_invariant (\forall int i = 0..N; ns[i].sum == sum(ns, ns[i].i, ns[i].n));
 3 while ((\forall int i = 0..N; ns[i].n < N-1)) {
 4   par (int i = 0..N-1)
 5     context Perm(nodes[i-1].sum, ϵ) ** Perm(nodes[i].in, 1);
 6     requires ns[i].sum == sum(ns, ns[i].i, ns[i].n);
 7     ensures ns[i+1].in == sum(ns, ns[i].i, ns[i].n);
 8   { int v = ns[i].sum;
 9     exhale ...; inhale ...;
10     ns[i+1].in = v; }
11   int v = ns[N-1].sum; exhale ...; inhale ...; ns[0].in = v;
12   par (int i = 0..N)
13     ensures ns[i].sum = ns[i].in + ns[i].v;
14   { ns[i].update(); }
```

Figure 6.9: Choreographic projection of core `while` loop of Fig. 6.3

versal quantifier. This enables reasoning over the entire endpoint family, even though during verification the size of endpoint families remains symbolic.

6.3.6 Choreographic Projection of Distributed Summation

The choreographic projection of Fig. 6.3 is shown in Fig. 6.9. We focus on the `while` loop as it contains the core of the algorithm. Each statement is transformed by an application of the rules CpCommRange, CpMethodCallRange and CpExprRange. Note that the branch unanimity check is added as a loop invariant, instead of a separate assert, to ensure branch unanimity is checked both at loop entry and exit. The `false` branch has been ommitted using "`...`". As the arguments of the `exhale`/`inhale` statements on lines 9 and 11 just repeat the contract of the preceding par block, they have also been ommitted. For the encoding of the parameterized method call, on line 13 we inline the contract of `update`, resulting in only an `ensures` clause.

6.4 Endpoint Projection

We will now discuss the transformation rules for the endpoint projection. It is written as $[\![\cdot]\!]_r$, where we refer to r as the *projection target*. The purpose of the endpoint projection is to transform a choreography such that it only executes parts relevant to the endpoint r. In this process, all choreographic primitives are replaced with plain PVL constructs. If this is done for all endpoints r in the choreography, when all endpoint projections are composed in parallel, the resulting program behaves exactly the same as the original choreography [80].

[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

For non-parameterized choreographies, the endpoint projection can be done using a syntactic check [36]. Summarizing, for a given projection target r, simply retain all choreographic statements that mention r. Parameterized choreographies introduce endpoint families, whose sizes will only be known at runtime. This makes the endpoint projection challenging: how to determine if the endpoint F`[`i`]` falls in the range `0..N`?

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

We resolve this by delaying projection of parameterized primitives until run-time. This is done by wrapping the projected statements in an `if` that checks if F`[`i`]` is in the relevant range. If so, the statement is executed as if projected for F`[`i`]`; otherwise, it is skipped. This way, the endpoint projection can safely simulate the program for any possible endpoint F`[`i`]`, at the cost of including an extra `if`.

6.4.1 Non-Parameterized Example

To give an intuition for the endpoint projection, we show a concrete example. Figure 6.4a is the input choreography, and Fig. 6.10 shows the output PVL of the endpoint projection for both `a` and `b`. Each communication is transformed into a concrete channel operation, depending on whether the projection target is `a` or `b`. For the condition of `if` and endpoint statements, only those relevant to the current projection target are kept, while others are replaced by `true`.

```
chan1.writeValue(a.balance > a.n);
if (a.balance > a.n && true) {
  chan2.writeValue(a.n);
  a.balance = a.balance - a.n;
  /* skip */
}
```

(a) Endpoint projection for a.

```
b.ok = chan1.readValue();
if (true && b.ok) {
  b.n = chan2.readValue();
  /* skip */
  b.balance = b.balance + b.n;
}
```

(b) Endpoint projection for b.

Figure 6.10: Endpoint projection of Fig. 6.4a.

6.4.2 Non-Parameterized Projection Rules

In Fig. 6.11 (top) we summarize the conceptually interesting endpoint projection rules. The full listing can be found in [136].

[136]: Rubbens et al. (2025), *Verified Parameterized Choreographies Technical Report*

Rule EpIf encodes an `if` statement by keeping it and applying the endpoint projection to the condition and substatements. For rule EpExpr, if the endpoint annotation matches the current projection target, the expression is simply kept.

Rule EpSend describes how a `communicate`, with implicit name L, should be encoded if singular endpoint a is in the sending position. Before the endpoint projection is done, VeyMont generates a channel instance for each `communicate` statement and assigns it to L. Then, when applying rule EpSend, the statement is replaced by a method invocation on a channel: $[\![L]\!]_a$`.writeValue(`E_{msg}`)`. Here, $[\![L]\!]_a$ represents the channel instance generated beforehand. The methods `readValue` and `writeValue` are part of the runtime environment that VeyMont generates [36, 133].

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*
[133]: Rubbens et al. (2024), *VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*

6.4.3 Parameterized Projection Rules

Parameterized projection rules are shown in Fig. 6.11 (bottom). These have to account for ranges of endpoint families. This is done by partially delaying the projection until runtime. For each projected statement or expression we check if the index of the current projection target is in the range (or the exact index) specified by the expression or statement. E.g. in rule EpRange, the expression E will only be evaluated if the index of the current projection target falls in the range of $[E_l, E_h)$. Similarly, rule EpRangeSend wraps the call to `writeValue` in an `if` statement to ensure adherence to the range. Rule EpRangeReceive is

EpIf
$[\![\texttt{if (}H\texttt{)}\ S_{true}\ S_{false}]\!]_r = \texttt{if (}[\![H]\!]_r\texttt{)}\ [\![S_{true}]\!]_r\ [\![S_{false}]\!]_r$

EpExpr
$[\![\texttt{(\textbackslash endpoint}\ e\texttt{;}\ E\texttt{)}]\!]_e = E$

EpSend
$[\![L\texttt{: communicate}\ a\texttt{:}\ H_{msg}\ \texttt{->}\ b\texttt{:}\ H_{dst}\texttt{;}]\!]_a^{send} =$
```
  [[L]]_a.writeValue(H_msg) with {
    sender = a; receiver = b; };
```

EpRangeSend
$[\![L\texttt{: communicate}\ F\texttt{[}j\texttt{:}\ E_l\ \texttt{..}\ E_h\texttt{].}f\ \texttt{->}\ G\texttt{[}d(j)\texttt{].}g]\!]_{F[i]}^{send} =$
```
  if (E_l <= i && i < E_h) {
    [[L]]_F[i] [i].writeValue(F[i].f) with {
      sender = F[i]; receiver = G[d(i)]; }; }
```

EpRangeReceive
$[\![L\texttt{: communicate}\ F\texttt{[}j\texttt{:}\ E_l\ \texttt{..}\ E_h\texttt{].}f\ \texttt{->}\ G\texttt{[}d(j)\texttt{].}g]\!]_{G[i]}^{receive} =$
```
  if (E_l <= d^-1(i) && d^-1(i) < E_h) {
    G[i].g = [[L]]_G[i] [d^-1(i)].readValue() with {
      sender = F[d^-1(i)]; receiver = G[i]; }; }
```

EpRange
$[\![\texttt{(\textbackslash endpoint}\ F\texttt{[}j\ \texttt{:=}\ E_l\ \texttt{..}\ E_h\texttt{];}\ E\texttt{)}]\!]_{F[i]} = E_l\ \texttt{<=}\ i\ \texttt{\&\&}\ i\ \texttt{<}\ E_h\ \texttt{==>}\ E$

Figure 6.11: Non-parameterized and parameterized endpoint projection rules

symmetric, in particular the use of d is also inverted as follows.

We require the function d used to compute the receiver index to be invertible. This is important, as the index of the sender determines which channel the receiving party should read from. We use the notation d^{-1} for the inverse function, e.g. in EpRangeReceive. The choreographic projection already guarantees that d is injective (Section 6.3.4). To actually compute the function d^{-1} in the projection, we use simple pattern matching to invert each operation in d. For example, if $d(i) = i + 1$, then $d^{-1} = i - 1$. This is an approach inspired by previous work: Ng et al. present Table II as a basis for such a transformation [114]. We think this step could be improved by reusing results in the field of bidirectional functions, such as [101]. We leave this for future work.

[114]: Ng et al. (2015), *Pabble: parameterised Scribble*

[101]: Matsuda et al. (2015), *"Bidirectionalization for free" for monomorphic transformations*

6

6.4.4 Example Endpoint Projection

The endpoint projection of Fig. 6.3 is shown in Fig. 6.12. There are two major differences between the choreography and its endpoint projection. First, each communication is split up into write and read statements. This is because in a parameterized communication, an endpoint can be both a sender and a receiver. E.g. in the summation choreography, line 10 of Fig. 6.3, node i must send to node $i + 1$, and receive from node $i - 1$. This is in contrast to regular non-parameterized communication, in which case it is statically known if the projection target is *either* a sender or a receiver, meaning less code is generated.

The second difference is that each statement in Fig. 6.12 is wrapped in an `if`, ensuring that the action is only executed if the index of the current endpoint (in this case i) falls in the range specified by each `communicate` statement (e.g. line 3). When a `communicate` is not parameterized, but involves a parameterized endpoint, such an `if` is also necessary, e.g. on line 6. Finally, this transformation is applied similarly to parameterized method calls (line 8).

6.5 Related Work

VeyMont This chapter builds on work around VeyMont. Jongmans et al. formalised verification of choreographies [80]. Van den Bos et al. first implemented the choreographic and endpoint projection in VeyMont [36]. Rubbens et al. extended them with shared memory support and annotation preservation [133].

[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

[36]: Van den Bos et al. (2023), *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*

[133]: Rubbens et al. (2024), *VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*

Choreographies The following works do not consider choreographic verification, or shared memory, but they do concern parameterization. We expect that our insights can be applied to the following works, and vice versa.

Jongmans introduces *first-person choreographic programming* (1CP), which is a novel formulation of choreographies with parameterization [79]. It is event-driven and dynamic. They

[79]: Jongmans (2025), *First-Person Choreographic Programming with Continuation-Passing Communications*

```
loop_invariant 0 <= i && i < N ==> ns[i].sum == sum(ns, i, ns[i].n);
while (0 <= i && i < N ==> ns[i].n < N-1)) {
  if (0 <= i && i < N-1) chan1[i].writeValue(ns[i].sum) with { sender = ns[i]; receiver = ns[i + 1]; };
  if (0 + 1 <= i && i < N-1+1) ns[i].in = chan1[i - 1].readValue()
                                          with { sender = ns[i - 1]; receiver = ns[i]; };
  if (i == N - 1) chan2.writeValue(ns[N-1].sum) with { sender = ns[N-1]; receiver = ns[0]; };
  if (i == 0) ns[0].in = chan2.readValue() with { sender = ns[N-1]; receiver = ns[0]; };
  if (0 <= i && i < N) ns[i].update(); }
```

Figure 6.12: Example of endpoint projection of Fig. 6.3

prove deadlock freedom of well-typed choreographies, support intricate messaging patterns such as pipelined communication, and provide tool support. They do not provide a way to verify functional correctness of the choreographies, nor of the endpoint projections, and also do not support shared memory. A key difference between their and our formulation of parameterized choreographies is that they do not allow indexing into endpoint families. Instead, they fully describe the network topology before the choreography is started, avoiding the need to define indexing of endpoint families in their semantics. We avoid defining the network topology by using verification to ensure injectivity and bounds checking for indexing operations.

Bates et al. support parameterized choreographies in the tool MultiChor [18]. This is achieved through *census polymorphism*, which essentially parameterizes a choreography over a set of endpoints by leveraging the Haskell type system. They fully delay the endpoint projection until runtime, where we only do this for parameterized parts of a choreography. While conditions still need to be propagated between endpoints to maintain deadlock freedom, they employ *enclaves* to limit the scope in which conditions need to be propagated. We use branch unanimity to guarantee deadlock freedom at verification time. They also do not support endpoint family indexing directly, but instead fix the network topology during the initialization phase, similar to Jongmans [79].

[18]: Bates et al. (2024), *Efficient, Portable, Census-Polymorphic Choreographic Programming*

Instead of generating programs from choreographies, Kjær et al. infer choreographies from parameterized programs [84]. They achieve this by parameterizing procedures with endpoint references, such that choreographies can model an endpoint substituting for another. Cruz-Filipe and Montesi similarly extend choreographies with procedures and dynamic participant allocation, allowing e.g. pipelined com-

[84]: Kjær et al. (2022), *From Infinity to Choreographies - Extraction for Unbounded Systems*

[53]: Cruz-Filipe et al. (2017), *Procedural Choreographic Programming*

munication [53]. They do not support endpoint family indexing.

[159]: Yoshida et al. (2006), *Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication*
[73]: Honda et al. (2008), *Multiparty asynchronous session types*
[157]: Yoshida et al. (2020), *A Very Gentle Introduction to Multiparty Session Types*

Session types Session types [159] are related, yet subtly different from choreographies. Session types type check protocol conformance for a given implementation, whereas choreographies allow generating an implementation. Related work in parameterization of session types focuses on the *multi-party* variant [73, 157], which allows more than two parties in the session type.

[43]: Castro-Perez et al. (2019), *Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures* (link)
[44]: Charalambides et al. (2012), *Parameterized Concurrent Multi-Party Session Types*
[56]: Deniélou et al. (2012), *Parameterised Multiparty Session Types*
[67]: Hamers et al. (2022), *The Discourje project: run-time verification of communication protocols in Clojure*
[114]: Ng et al. (2015), *Pabble: parameterised Scribble*

None of the works on parameterized session types we found support verification of functional correctness and shared memory [43, 44, 56, 67, 114]. Their support for indexing into endpoint families is usually restricted to some decidable fragment of arithmetic, where we support general recursive functions. We use verification to show that bounds of endpoint families are respected, and that indexing is injective, guaranteeing that the endpoint projection produces safe code. Except for Hamers et al [67], they all require some form of symmetry in the session type to ensure deadlock-freedom of the projection, whereas we use branch unanimity to guarantee deadlock freedom (Section 6.3.5). The works on session types do support more communication patterns, such as many-to-one or pipelined communication. This is still a challenge for our formulation of choreographies: more annotations will be required from the user, making this extension non-trivial.

Hamers et al. [67] do dynamic checking of session types. This means they do not need a mechanism like branch unanimity to avoid deadlocks. In exchange, a session type might crash because non-compliance is detected at run-time.

[98]: Magee et al. (2006), *Concurrency - state models and Java programs (2. ed.)*

Ng et al. [114] introduce the endpoint family notation we use as well, which is in turn inspired by [98]. Furthermore, Ng et al. also require that indexing of endpoint families is an invertible operation. We generalize this requirement: for the choreographic projection, indexing merely needs to be injective. This suffices for verification, as VerCors can reason about injectivity. The endpoint projection requires an invertible expression, as indices need to be computable.

6.6 Conclusion

We proposed an extension of the automated verifier and code generator VeyMont for parameterized choreographies, opening the door for verification of choreographies with an arbitrary number of participants.

Adding parameterization support required improvements to several VeyMont components. We first defined the syntax of choreographies with parameterized primitives, such as endpoint families and parameterized communication. We then extended the choreographic projection with support for parameterized choreographies, which leverages the `par` block from VerCors. In addition, we restrict the input language of the choreographic projection such that verification annotations can be automatically generated. We also showed that deadlock freedom of parameterized choreographies can be checked by quantification over entire endpoint families using universal quantifiers.

Also, we extended the endpoint projection with support for endpoint families. The endpoint projection generates one program that works for each symbolic index of the family. This is implemented by checking the index at run-time, enabling only statements for the specified index.

We have illustrated and motivated our contribution by verifying a distributed summation choreography, included in the artifact [139]. To the best of our knowledge, there is no prior work on verification of parameterized choreographies.

[139]: Rubbens et al. (2025), *Artefact of: Verified Parameterized Choreographies*

Future work We will complete the implementation of the approach presented in this work, and further evaluate the approach with more case studies. Then, future work will go in several directions. We will investigate the possibility of "ghost" communication statements, which allow communicating ghost state and proof hints between endpoints with zero run-time overhead. Another direction is to make interactions more flexible, for example by allowing heap locations as indices for parameterized communication statements, and by adding pipelined and many-to-one communication. Finally, we wish to integrate the branch unanimity check into the endpoint projection. This would allow modification, reverification and further analysis of deadlock freedom of the generated code.

Conclusion 7

In this thesis, we started with investigating the lack of use of formal methods in industry. Using the insights gained from this investigation, we combined formal methods to acquire new *hybrid* formal methods that, we believe, narrow the gap between mental models and formal verification tools. In particular, we have considered the combinations of

- the program verifier VerCors and the software design framework JavaBIP, and
- deductive verification and choreographies in the choreographic verifier VeyMont.

These contributions were presented in four parts.

Formal methods in Industry One way to improve the reliability of software is to use *formal methods*, which can show correctness of software exhaustively w.r.t. a specification. Despite successes [20], application of formal methods to industrial projects is still relatively rare [64]. To get better insight into why this is, we did a case study where we applied the *auto-active deductive verification* tool VerCors to an industrial software system. During this process we discovered two concurrency bugs. We conclude that the level of support for Java in the VerCors tool would still require improvements to be effective in practice, in particular in the areas of generics, inheritance, and lambdas. However, with this support in place, we argue that both bugs could have been detected with VerCors.

[20]: Ter Beek et al. (2025), *Formal Methods in Industry*

[64]: Gleirscher et al. (2023), *A manifesto for applicable formal methods*

We communicated these results to an audience that had no familiarity with formal methods, taking extra care with preparing the presentation. We did this by introducing formal methods concepts with high granularity, as well as only focusing on core concepts. The audience reported that while formal methods look promising, it is hard to allocate time to write the annotations necessary to apply them. In addition, the distance between the mental models of developers and the abstraction level of verification annotations is currently too large.

Towards Verified Concurrent Systems in Java To reduce the distance between mental models of developers and verification annotations required at the level of the code, we combined two conceptually different formal methods. The first formal method is a *top-down* formal method called JavaBIP, which is a framework for implementing Java systems based on a high-level specification of components and their interactions. The second formal method is a *bottom-up* formal method called VerCors, which shows correctness of a Java implementation with regard to a code-level correctness specification.

Combining these two tools results in a new hybrid formal method: Verified JavaBIP. This new formal method supports annotating JavaBIP models with implementation-level correctness annotations. We implemented support in VerCors to deductively verify annotated JavaBIP models. In addition, we extended the JavaBIP engine to check, at run-time, any annotations that were not verified deductively. We show that the implemented approach is effective by applying it to the VerifyThis Long-Term Challenge casino example.

Verified JavaBIP had two particular limitations that we addressed in the next two parts of this thesis. First, Verified JavaBIP did not support models that share memory between components. Second, it was unclear how JavaBIP models with a dynamic number of components could be verified.

Verified Shared Memory Choreographies We investigate verification of high-level models with shared memory in the context of a different hybrid formal method: VeyMont.

VeyMont combines *choreographies* with *program verification*, and similar to JavaBIP, also lacked shared memory support.

To add shared memory support to VeyMont, we extended its choreography language with primitives for expressing shared memory access patterns. We then described the necessary extensions to two pre-existing choreography transformations. First, we adapt the verification transformation, called the *choreographic projection*, to verify memory safety and functional correctness at the choreographic level. Second, we adapt the code generation transformation, called the *endpoint projection*, to include memory safety annotations.

Adding shared memory support to choreographies facilitates two useful capabilities. First, it is now possible to preserve correctness annotations in the code generated with the endpoint projection. This allows re-verification of the generated code, which is beneficial for robustness and maintainability. Second, it allows expressing proof steps over shared ghost state, which makes some proofs easier to express. We illustrate this new capability with a case study of several variations of the Tic-Tac-Toe choreography. The case study illustrates a trade-off between performance and annotation size.

Verified Parameterized Choreographies We also consider parameterized choreographies. We define the primitives necessary to parameterize choreographies. In particular, we extend choreographies with *endpoint families*, which are collections of endpoints with size determined by a symbolic parameter, and *parameterized communication*, which allows one-to-one communication between endpoint families.

We then adapt the choreographic projection and endpoint projection to support these parameterization primitives. For the choreographic projection, we encode endpoint families as immutable sequences. Parameterized communication statements are encoded using the `par` block, which is a structured parallelism primitive from VerCors for verification of a parameterized number of threads. For the endpoint sprojection, we ensure that code is generated that can execute the choreography for each member of an endpoint family. This behaviour is then conditionally refined at run-time, based on the run-time index of the target endpoint. To be able to generate all necessary annotations automatically, we impose two limitations. First, parameterized communications must not depend on heap variables. Second, indexing expressions into endpoint families must be invertible.

We illustrate our approach by defining a distributed summation algorithm on a ring network using parameterized choreographies. We verify the algorithm with VerCors using a manual encoding into PVL.

To conclude, we have reduced the gap between industrial application and theoretical design of formal methods by:

- Applying a deductive verifier in an industrial setting, determining current strengths and weaknesses and documenting feedback from informed practitioners.
- Combining correctness-by-construction with program verification to create a new hybrid formal method that allows expressing implementation concerns at the design level.
- Extending choreographic verification with shared memory and parameterization, facilitating simpler correctness proofs and enabling verification of generated code. This makes choreographic verification more generally applicable and more valuable in practice.

With these contributions, we believe we have significantly reduced the gap between, on one hand, mental models of software, and on the other hand, their implementation, which is essential for encouraging a higher degree of adoption of formal methods in industry.

7.1 Future Work

We will now discuss several promising directions of future work.

Java support While applying VerCors to an industrial code base, we ran into issues in the category of missing support for Java language features. While there is existing work in this direction, e.g. [48], this has not yet been further refined into a robust and generally applicable approach for concurrent software verification. During our research, we did not encounter a deductive verifier with adequate support for Java as written in an industrial context.

[48]: Cok et al. (2018), *Practical Methods for Reasoning About Java 8's Functional Programming Features*

Moreover, merely implementing support for advanced language futures is not enough. In particular, research is necessary on how verification support for these features interacts with idiomatic Java programming patterns, as well as with major Java frameworks such as Spring Boot[1] and Hibernate[2]. In other words, progress in this research direction can only be made if there is a clear picture of the properties that industrial Java developers want to verify.

1: `https://spring.io`

2: `https://hibernate.org`

Annotation pressure The relation between lines of code and lines of annotation is subject to ongoing research. For example:

- Lathouwers defines the *specification bottleneck*, which occurs when the limiting factor for applying formal methods is the number of annotations required [91].
- For a particular large-scale verification effort done by Pereira et al., the ratio of lines of annotations to lines of code is 2.8 [126].
- Tasche et al. introduce a technique to generate the so-called RASI, specifically to alleviate the user of having to write it. The RASI is a large annotation summarizing the state space the verifier must consider [150].
- Armborst et al. show how to automatically generate certain verification annotations for VerCors using CPAChecker [23], also alleviating the user of the annotation burden [12].

[91]: Lathouwers (2023), *Exploring annotations for deductive verification*

[126]: Pereira et al. (2024), *Protocols to Code: Formal Verification of a Next-Generation Internet Router*

[150]: Tasche et al. (2024), *Deductive Verification of Parameterized Embedded Systems Modeled in SystemC*

[23]: Beyer et al. (2011), *CPAchecker: A Tool for Configurable Software Verification*

[12]: Armborst et al. (2025), *AutoSV-Annotator: Integrating Deductive and Automatic Software Verification*

In Chapter 3 of this thesis, we have also found that developers report there is only a limited amount of time they can allocate to writing annotations. The developers also report that they think the approach of Rust with syntax for ownership and lifetimes is promising. We agree, and expect that concepts popular and effective in Rust, and possibly in the broader setting of ownership systems, can be transferred to the context of deductive verification with separation logic. This is not a novel insight but instead a topic of on-going research [14, 127].

[14]: Astrauskas et al. (2022), *The Prusti Project: Formal Verification for Rust*

[127]: Protopapa (2024), *Verifying Kotlin Code with Viper by Controlling Aliasing* (link)

Beyond Verified JavaBIP There are two directions that are interesting to explore in particular for Verified JavaBIP. The first is runtime verification of memory safety. This can be done in the context of JavaBIP models, but there is a possibility a lightweight approach for Java is straightforward to adapt to JavaBIP. There is already some earlier work [82] and recent work [77] in this direction.

[82]: Kandziora et al. (2015), *Runtime assertion checking of JML annotations in multithreaded applications with e-OpenJML*

[77]: Janssen (2024), *Design and Implementation of new features for Runtime Permission Verification in Concurrent Java Programs using VerCors* (link)

The second direction is to investigate verifying JavaBIP beyond safety properties. In this thesis, we focused on verification of invariants of components and component states, and contracts of component transitions. However, in combination with the foundations of JavaBIP in synchronizing automatons, integration of temporal logic properties in Verified JavaBIP should be possible. For example, the encoding of LTL as presented by Hawblitzel et al. could be a suitable starting point for a deductive approach [68]. Alternatively, Mediouni et al. have already extended BIP with statistical model checking. Integrating this with Verified JavaBIP is a logical next step [103].

[68]: Hawblitzel et al. (2015), *IronFleet: proving practical distributed systems correct*

[103]: Mediouni et al. (2018), S *BIP 2.0: Statistical Model Checking Stochastic Real-Time Systems*

Choreographic verification The earlier mentioned challenge of annotation pressure is also relevant for VeyMont. Because VeyMont combines the logical contexts for each endpoint in one notation, the amount of annotations that must be written in one place also grows linearly in the number of endpoints. Research could be done on how to exploit the choreographic view of a system for more concise annotation syntax. For example, perhaps there are symmetries in the annotations supported by VeyMont that remain to be utilized.

Another direction of research that would be fruitful is extending VeyMont with more communication patterns. For example, generalizing communication in VeyMont to support one-to-many, many-to-one, and pipelined communications should be possible. In addition, at the cost of writing additional annotations, we expect that some form of heap-dependent parameterized communication might be achievable.

Besides improving verification support, it should also be investigated in which domains VeyMont could be applied. A notable direction is that of *infrastructure as code*, which is a domain where scripts written in YAML notation determine deployment schemes for internet platforms [11]. Another is the application of verified choreographies to high-performance computing and GPGPU programming, which are good fits for the choreographic paradigm because of their semi-distributed[3] nature. Some progress in this direction has already been made in the area of session types [96, 113].

[11]: Arfi et al. (2024), *An Overview of the Decentralized Reconfiguration Language Concerto-D through its Maude Formalization*

3: We mean "semi-distributed" here in the sense of a distributed system with participants known upfront and constant, e.g. available nodes in a computing cluster. In a fully general definition of a distributed system, the set of participants can vary over time.

[96]: López et al. (2015), *Protocol-based verification of message-passing parallel programs*

[113]: Ng et al. (2010), *High performance parallel design based on session programming* (link)

Finally, there are also still foundational aspects of VeyMont that could be further explored. For example, while there is a semantics for the initial implementation of VeyMont [80], the extensions presented in this thesis have not yet been formally described.

[80]: Jongmans et al. (2022), *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*

Another foundational matter is that the code generated using the endpoint projection does not encompass all properties encoded with the choreographic projection. In particular, deadlock freedom is not included. We think it could be both theoretically and practically challenging, as well as enlightening from a foundational perspective, to design new annotations which the endpoint projection could use to encode deadlock freedom at the implementation level. Crucial progress has already been made in this direction by Bílý et al., who introduce various annotations to show liveness properties in a deductive setting [24]. These annotations could be a possible primitive to target in an extended endpoint projection.

[24]: Bílý et al. (2023), *Refinement Proofs in Rust Using Ghost Locks*

Bibliography

[1] Tesnim Abdellatif and Kei-Leo Brousmiche. “Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models”. In: *9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018-02, pp. 1–5. DOI: 10.1109/NTMS.2018.8328737 (cit. on p. 78).

[2] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. ISBN: 978-0-521-89556-9. DOI: 10.1017/CBO9781139195881 (cit. on p. 8).

[3] Wolfgang Ahrendt, Jonas Becker-Kupczok, Simon Bliudze, Petra van den Bos, Marco Eilers, Gidon Ernst, Martin Fabian, Paula Herber, Marieke Huisman, Raúl E. Monti, Robert Rubbens, Larisa Safina, Jonas Schiffl, Alexander J. Summers, Mattias Ulbrich, and Alexander Weigl. “From Model Checking to Deductive Verification: Results from a Smart Contract Community Challenge”. In: *International Journal on Software Tools for Technology Transfer (STTT)* (2025). Under submission (cit. on p. 82).

[4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6 (cit. on p. 7).

[5] Wolfgang Ahrendt, Jess Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. “Verifying data- and control-oriented properties combining static and runtime verification: theory and tools”. In: *Form. Methods Syst. Des.* 51.1 (2017-08), pp. 200–265. ISSN: 1572-8102. DOI: 10.1007/s10703-017-0274-y (cit. on p. 78).

[6] Wolfgang Ahrendt, Gidon Ernst, Paula Herber, Marieke Huisman, Raúl E. Monti, Mattias Ulbrich, and Alexander Weigl. “The VerifyThis Collaborative Long-Term Challenge Series”. In: *TOOLympics Challenge 2023 - Updates, Results, Successes of the Formal-Methods Competitions, TOOLympics 2023, Paris, France*. Ed. by Dirk Beyer, Arnd Hartmanns, and Fabrice Kordon. Vol. 14550. Lecture Notes in Com-

puter Science. Springer, 2023, pp. 160–170. DOI: 10.1007/978-3-031-67695-6_6 (cit. on p. 19).

[7] Afshin Amighi, Clment Hurlin, Marieke Huisman, and Christian Haack. “Permission-Based Separation Logic for Multithreaded Java Programs”. In: *Logical Methods in Computer Science* 11.1 (2015-02). DOI: 10.2168/LMCS-11(1:2)2015 (cit. on pp. 76, 82).

[8] Pascal Andr, Christian Attiogb, and Jean-Marie Mottu. *Combining Techniques to Verify Service-based Components.* [Online; accessed 26. Sep. 2022]. 2022-09. URL: https://www.scitepress.org/Link.aspx?doi=10.5220/0006212106450656 (cit. on p. 78).

[9] Krzysztof Apt and Ernst-Rüdiger Olderog. “Fifty years of Hoare’s logic”. In: *Formal Aspects of Computing* 31.6 (2019-12), pp. 751–807. DOI: 10.1007/s00165-019-00501-3 (cit. on p. 26).

[10] Farhad Arbab. “Reo: A channel-based coordination model for component composition”. In: *Mathematical Structures in Computer Science* 14.3 (2004), pp. 329–366. DOI: 10.1017/S0960129504004153 (cit. on pp. 13, 76).

[11] Farid Arfi, Hélène Coullon, Frédéric Loulergue, Jolan Philippe, and Simon Robillard. “An Overview of the Decentralized Reconfiguration Language Concerto-D through its Maude Formalization”. In: *Proceedings 17th Interaction and Concurrency Experience, ICE 2024, Groningen, The Netherlands, 21st June 2024.* Ed. by Clément Aubert, Cinzia Di Giusto, Simon Fowler, and Violet Ka I Pun. Vol. 414. EPTCS. 2024, pp. 21–38. DOI: 10.4204/EPTCS.414.2 (cit. on p. 147).

[12] Lukas Armborst, Dirk Beyer, Marieke Huisman, and Marian Lingsch-Rosenfeld. “AutoSV-Annotator: Integrating Deductive and Automatic Software Verification”. In: Accepted at FMICS 2025. 2025 (cit. on p. 145).

[13] Lukas Armborst, Pieter Bos, Lars B. van den Haak, Marieke Huisman, Robert Rubbens, Ömer Sakar, and Philip Tasche. “The VerCors Verifier: A Progress Report”. In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II.* Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14682. Lecture Notes in Computer Science. Springer, 2024, pp. 3–18. DOI: 10.1007/978-3-031-65630-9_1 (cit. on pp. 11, 12, 16, 23, 38, 57, 76, 77, 92, 93).

[14] Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. “The Prusti Project: Formal Verification for Rust”. In: *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings.* Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Vol. 13260. Lecture Notes in Computer Science. Springer, 2022, pp. 88–108. DOI: 10.1007/978-3-031-06773-0_5 (cit. on p. 146).

[15] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008. ISBN: 978-0-262-02649-9 (cit. on p. 7).

[16] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24 (cit. on p. 34).

[17] Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling Heterogeneous Real-time Components in BIP". In: *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006)*. IEEE Computer Society, 2006-09, pp. 3–12. DOI: 10.1109/SEFM.2006.27 (cit. on pp. 13, 76).

[18] Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. *Efficient, Portable, Census-Polymorphic Choreographic Programming*. Submitted at PLDI'25. 2024. DOI: 10.48550/ARXIV.2412.02107 (cit. on p. 137).

[19] Andreas Bauer, Martin Leucker, and Jonathan Streit. "SALT—Structured Assertion Language for Temporal Logic". In: *Formal Methods and Software Engineering*. Ed. by Zhiming Liu and Jifeng He. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 757–775. ISBN: 978-3-540-47462-3 (cit. on p. 73).

[20] 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". In: *Formal Aspects Comput.* 37.1 (2025), 7:1–7:38. DOI: 10.1145/3689374 (cit. on pp. 4–7, 9, 11, 141).

[21] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. "The Temporal Logic Sugar". In: *Computer Aided Verification*. Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 363–367. ISBN: 978-3-540-44585-2 (cit. on p. 73).

[22] Mordechai Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition*. Springer, 2012. ISBN: 978-1-4471-4128-0. DOI: 10.1007/978-1-4471-4129-7 (cit. on pp. 24, 82).

[23] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16 (cit. on p. 145).

[24] Aurel Bílý, João C. Pereira, Jan Schär, and Peter Müller. “Refinement Proofs in Rust Using Ghost Locks”. In: *CoRR* abs/2311.14452 (2023). DOI: 10.48550/ARXIV.2311.14452 (cit. on pp. 7, 147).

[25] Rijkswaterstaat. *Welkom bij de Blankenburgverbinding*. Accessed: 2022-01-21. 2022. URL: %5Curl%7Bhttps://www.blankenburgverbinding.nl/%7D (cit. on p. 58).

[26] Simon Bliudze, Petra van den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. *Artefact of: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java*. 2023. DOI: 10.4121/21763274 (cit. on pp. 20, 78, 82).

[27] Simon Bliudze, Petra van den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. “JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java”. In: *Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings* (FASE). Ed. by Leen Lambers and Sebastián Uchitel. Vol. 13991. Lecture Notes in Computer Science. Springer, 2023, pp. 143–150. DOI: 10.1007/978-3-031-30826-0_8 (cit. on pp. 20, 75).

[28] Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. “Formal Verification of Infinite-State BIP Models”. In: *Automated Technology for Verification and Analysis*. Ed. by Bernd Finkbeiner, Geguang Pu, and Lijun Zhang. Cham: Springer International Publishing, 2015, pp. 326–343. ISBN: 978-3-319-24953-7 (cit. on p. 78).

[29] Simon Bliudze, Panagiotis Katsaros, Saddek Bensalem, and Martin Wirsing. “On methods and tools for rigorous system design”. In: *Int. J. Softw. Tools Technol. Transf.* 23.5 (2021), pp. 679–684. DOI: 10.1007/s10009-021-00632-0 (cit. on p. 76).

[30] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. “Exogenous coordination of concurrent software components with JavaBIP”. In: *Software: Practice and Experience* 47.11 (2017-04), pp. 1801–1836. DOI: 10.1002/spe.2495 (cit. on pp. 11, 76, 79).

[31] Stefan Blom, Saeed Darabi, and Marieke Huisman. “Verification of Loop Parallelisations”. In: *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Alexander Egyed and Ina Schaefer. Vol. 9033. Lecture Notes in Computer Science. Springer, 2015, pp. 202–217. DOI: 10.1007/978-3-662-46675-9_14 (cit. on p. 110).

[32] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. “Why3: Shepherd Your Herd of Provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, 2011-08, pp. 53–64 (cit. on p. 112).

[33] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. "A Theory of Design-by-Contract for Distributed Multiparty Interactions". In: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. Ed. by Paul Gastin and François Laroussinie. Vol. 6269. Lecture Notes in Computer Science. Springer, 2010, pp. 162–176. DOI: 10.1007/978-3-642-15375-4_12 (cit. on p. 112).

[34] Eric Bodden, Patrick Lam, and Laurie Hendren. "Partially Evaluating Finite-State Runtime Monitors Ahead of Time". In: *ACM Trans. Program. Lang. Syst.* 34.2 (2012-06), pp. 1–52. ISSN: 0164-0925. DOI: 10.1145/2220365.2220366 (cit. on p. 78).

[35] Tabea Bordis and K. Rustan M. Leino. "Free Facts: An Alternative to Inefficient Axioms in Dafny". In: *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part I*. Ed. by André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi. Vol. 14933. Lecture Notes in Computer Science. Springer, 2024, pp. 151–169. DOI: 10.1007/978-3-031-71162-6_8 (cit. on p. 33).

[36] Petra van den Bos and Sung-Shik Jongmans. "VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs". In: *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings*. Ed. by Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker. Vol. 14000. Lecture Notes in Computer Science. Springer, 2023, pp. 321–339. DOI: 10.1007/978-3-031-27481-7_19 (cit. on pp. 11, 16, 24, 45, 48, 53, 92, 94, 95, 99, 104–108, 110, 117–119, 122, 123, 133, 134, 136).

[37] Jelle Bouma, Stijn de Gouw, and Sung-Shik Jongmans. "Multiparty Session Typing in Java, Deductively". In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13994. Lecture Notes in Computer Science. Springer, 2023, pp. 19–27. DOI: 10.1007/978-3-031-30820-8_3 (cit. on p. 111).

[38] Julian Bradfield and Igor Walukiewicz. "The Mu-calculus and Model Checking". In: *Handbook of Model Checking*. Cham: Springer International Publishing, 2018, pp. 871–919. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_26 (cit. on p. 73).

[39] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. "The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*. Ed. by Tomás Vojnar and Li-

jun Zhang. Vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 21–39. DOI: 10.1007/978-3-030-17465-1_2 (cit. on p. 7).

[40] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification". In: *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings.* Ed. by Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 9058. Lecture Notes in Computer Science. Springer, 2015, pp. 3–11. DOI: 10.1007/978-3-319-17524-9_1 (cit. on p. 11).

[41] Marco Carbone, Davide Grohmann, Thomas T. Hildebrandt, and Hugo A. López. "A Logic for Choreographies". In: *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010.* Ed. by Kohei Honda and Alan Mycroft. Vol. 69. EPTCS. 2010, pp. 29–43. DOI: 10.4204/EPTCS.69.3 (cit. on p. 110).

[42] Marco M. Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. "Heartbleed 101". In: *IEEE Secur. Priv.* 12.4 (2014), pp. 63–67. DOI: 10.1109/MSP.2014.66 (cit. on p. 2).

[43] David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. "Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 29:1–29:30. DOI: 10.1145/3290342. URL: https://doi.org/10.1145/3290342 (cit. on p. 138).

[44] Minas Charalambides, Peter Dinges, and Gul Agha. "Parameterized Concurrent Multi-Party Session Types". In: *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012.* Ed. by Natallia Kokash and António Ravara. Vol. 91. EPTCS. 2012, pp. 16–30. DOI: 10.4204/EPTCS.91.2 (cit. on p. 138).

[45] Raymond Chen. *On finding the average of two unsigned integers without overflow.* URL: https://devblogs.microsoft.com/oldnewthing/20220207-00/?p=106223 (visited on 2025-02-07). (Internet Archive link) (cit. on p. 35).

[46] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings.* Ed. by Ed Brinksma and Kim Guldstrand Larsen. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 359–364. DOI: 10.1007/3-540-45657-0_29 (cit. on p. 7).

[47] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *SIGPLAN Not.* 35.9 (2000), pp. 268–279. ISSN: 0362-1340. DOI: 10.1145/357766.351266 (cit. on p. 70).

[48] David R. Cok and Serdar Tasiran. “Practical Methods for Reasoning About Java 8’s Functional Programming Features”. In: *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*. Ed. by Ruzica Piskac and Philipp Rümmer. Vol. 11294. Lecture Notes in Computer Science. Springer, 2018, pp. 267–278. DOI: 10.1007/978-3-030-03592-1_15 (cit. on p. 145).

[49] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. “A Language Framework for Expressing Checkable Properties of Dynamic Software”. In: *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*. Ed. by Klaus Havelund, John Penix, and Willem Visser. Vol. 1885. Lecture Notes in Computer Science. Springer, 2000, pp. 205–223. DOI: 10.1007/10722468_13. URL: https://doi.org/10.1007/10722468%5C_13 (cit. on p. 73).

[50] *Coverity homepage*. accessed on: 2022-05-18. 2022. URL: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html (cit. on p. 72).

[51] CrowdStrike. *External Technical Root Cause Analysis — Channel File 291*. 2024. URL: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf (visited on 2025-03-21). (Internet Archive link) (cit. on p. 1).

[52] Luís Cruz-Filipe, Eva Graversen, Fabrizio Montesi, and Marco Peressotti. “Reasoning About Choreographic Programs”. In: *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*. Ed. by Sung-Shik Jongmans and Antónia Lopes. Vol. 13908. Lecture Notes in Computer Science. Springer, 2023, pp. 144–162. DOI: 10.1007/978-3-031-35361-1_8 (cit. on p. 110).

[53] Luís Cruz-Filipe and Fabrizio Montesi. “Procedural Choreographic Programming”. In: *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017*. Ed. by Ahmed Bouajjani and Alexandra Silva. Vol. 10321. Lecture Notes in Computer Science. Springer, 2017, pp. 92–107. DOI: 10.1007/978-3-319-60225-7_7 (cit. on p. 138).

[54] Lukasz Czajka and Cezary Kaliszyk. “Hammer for Coq: Automation for Dependent Type Theory”. In: *J. Autom. Reason.* 61.1-4 (2018), pp. 423–453. DOI: 10.1007/S10817-018-9458-4 (cit. on p. 6).

[55] Saeed Darabi, Stefan C. C. Blom, and Marieke Huisman. “A Verification Technique for Deterministic Parallel Programs”. In: *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceed-*

ings. Ed. by Clark W. Barrett, Misty D. Davies, and Temesghen Kahsai. Vol. 10227. Lecture Notes in Computer Science. 2017, pp. 247–264. DOI: 10.1007/978-3-319-57288-8_17 (cit. on p. 40).

[56] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. "Parameterised Multiparty Session Types". In: *Log. Methods Comput. Sci.* 8.4 (2012). DOI: 10.2168/LMCS-8(4:6)2012 (cit. on pp. 125, 138).

[57] Farzaneh Derakhshan, Stefan Muller, Jim Sasaki, and Mike Gordon. *Hoare triples I & II.* 2023. URL: http://cs.iit.edu/~smuller/cs536-f23/lectures/07-hoaretrp.pdf. (Internet Archive link) (cit. on p. 26).

[58] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (1975), pp. 453–457. DOI: 10.1145/360933.360975 (cit. on p. 9).

[59] Mark Dowson. "The Ariane 5 software failure". In: *ACM SIGSOFT Softw. Eng. Notes* 22.2 (1997), p. 84. DOI: 10.1145/251880.251992 (cit. on p. 1).

[60] T. Dubbeling. *Predicate subtyping in VerCors.* 2024-07. URL: http://essay.utwente.nl/100843/ (cit. on p. 36).

[61] *Findbugs homepage.* accessed on: 2022-05-18. 2022. URL: http://findbugs.sourceforge.net/ (cit. on p. 72).

[62] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML.* Elsevier, 2015. ISBN: 9780128002025. DOI: 10.1016/c2013-0-14457-1 (cit. on p. 11).

[63] Sergiu Gatlan. *Windows 11, Tesla, Ubuntu, and macOS hacked at Pwn2Own 2023.* 2023. URL: https://www.bleepingcomputer.com/news/security/windows-11-tesla-ubuntu-and-macos-hacked-at-pwn2own-2023/ (visited on 2025-02-11). (Internet Archive link) (cit. on p. 3).

[64] Mario Gleirscher, Jaco van de Pol, and Jim Woodcock. "A manifesto for applicable formal methods". In: *Softw. Syst. Model.* 22.6 (2023), pp. 1737–1749. DOI: 10.1007/S10270-023-01124-2 (cit. on pp. 4, 12, 141).

[65] Christian Haack, Marieke Huisman, and Clément Hurlin. "Reasoning about Java's Reentrant Locks". In: *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings.* Ed. by G. Ramalingam. Vol. 5356. Lecture Notes in Computer Science. Springer, 2008, pp. 171–187. DOI: 10.1007/978-3-540-89330-1_13 (cit. on p. 43).

[66] Christian Haack, Marieke Huisman, Clément Hurlin, and Afshin Amighi. "Permission-Based Separation Logic for Multithreaded Java Programs". In: *Logical Methods in Computer Science* Volume 11, Issue 1 (2015-02). ISSN: 1860-5974. DOI: 10.2168/lmcs-11(1:2)2015 (cit. on pp. 12, 37).

[67] Ruben Hamers, Erik Horlings, and Sung-Shik Jongmans. “The Discourje project: run-time verification of communication protocols in Clojure”. In: *Int. J. Softw. Tools Technol. Transf.* 24.5 (2022), pp. 757–782. DOI: 10.1007/S10009-022-00674-Y (cit. on p. 138).

[68] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. “IronFleet: proving practical distributed systems correct”. In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015.* Ed. by Ethan L. Miller and Steven Hand. ACM, 2015, pp. 1–17. DOI: 10.1145/2815400.2815428 (cit. on pp. 12, 146).

[69] Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, Marko C. J. D. van Eekelen, and Stijn de Gouw. “Verifying OpenJDK’s LinkedList using KeY”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II.* Ed. by Armin Biere and David Parker. Vol. 12079. Lecture Notes in Computer Science. Springer, 2020, pp. 217–234. DOI: 10.1007/978-3-030-45237-7_13 (cit. on p. 7).

[70] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. “Actris: session-type based reasoning in separation logic”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 6:1–6:30. DOI: 10.1145/3371074 (cit. on p. 111).

[71] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259 (cit. on p. 26).

[72] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. “Language Primitives and Type Discipline for Structured Communication-Based Programming”. In: *Programming Languages and Systems - ESOP’98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings.* Ed. by Chris Hankin. Vol. 1381. Lecture Notes in Computer Science. Springer, 1998, pp. 122–138. DOI: 10.1007/BFB0053567 (cit. on p. 92).

[73] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty asynchronous session types”. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008.* Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 273–284. DOI: 10.1145/1328438.1328472 (cit. on p. 138).

[74] Marieke Huisman and Raúl E. Monti. “On the Industrial Application of Critical Software Verification with VerCors”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Applications.* Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2020, pp. 273–292. ISBN: 978-3-030-61467-6. DOI: 10.1007/978-3-030-61467-6_18 (cit. on pp. 57, 58).

[75] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55. DOI: `10.1007/978-3-642-20398-5_4` (cit. on p. 6).

[76] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. "Dependent Session Protocols in Separation Logic from First Principles (Functional Pearl)". In: *Proc. ACM Program. Lang.* 7.ICFP (2023), pp. 768–795. DOI: `10.1145/3607856` (cit. on p. 111).

[77] Dylan Damian Janssen. *Design and Implementation of new features for Runtime Permission Verification in Concurrent Java Programs using VerCors*. 2024-05. URL: `http://essay.utwente.nl/98745/` (cit. on p. 146).

[78] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: `10.1109/TSE.2010.62` (cit. on p. 70).

[79] Sung-Shik Jongmans. "First-Person Choreographic Programming with Continuation-Passing Communications". In: *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*. Ed. by Viktor Vafeiadis. Vol. 15695. Lecture Notes in Computer Science. Springer, 2025, pp. 62–90. DOI: `10.1007/978-3-031-91121-7_3` (cit. on pp. 136, 137).

[80] Sung-Shik Jongmans and Petra van den Bos. "A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming". In: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. Ed. by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 520–547. DOI: `10.1007/978-3-030-99336-8\_19` (cit. on pp. 16, 24, 45, 48, 93, 108, 110, 113, 117, 133, 136, 147).

[81] Anne Kaldewaij. *Programming - the derivation of algorithms*. Prentice Hall international series in computer science. Prentice Hall, 1990. ISBN: 978-0-13-204108-9 (cit. on p. 9).

[82] Jorne Kandziora, Marieke Huisman, Christoph Bockisch, and Marina Zaharieva-Stojanovski. "Run-time assertion checking of JML annotations in multithreaded applications with e-OpenJML". In: *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015, Prague, Czech Republic, July 7, 2015*. Ed. by Rosemary Monahan. ACM, 2015, 8:1–8:6. DOI: `10.1145/2786536.2786541` (cit. on p. 146).

[83] Andrea Keusch. *Adding Debugging Functionality to Viper*. URL: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Andrea_Keusch_PW_Report.pdf (visited on 2025-05-24). (Internet Archive link) (cit. on p. 6).

[84] Bjørn Angel Kjær, Luís Cruz-Filipe, and Fabrizio Montesi. "From Infinity to Choreographies - Extraction for Unbounded Systems". In: *Logic-Based Program Synthesis and Transformation - 32nd International Symposium, LOPSTR 2022, Tbilisi, Georgia, September 21-23, 2022, Proceedings*. Ed. by Alicia Villanueva. Vol. 13474. Lecture Notes in Computer Science. Springer, 2022, pp. 103–120. DOI: 10.1007/978-3-031-16767-6_6 (cit. on p. 137).

[85] *Klocwork homepage*. accessed on: 2022-05-18. 2022. URL: https://www.perforce.com/products/klocwork (cit. on p. 72).

[86] Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012. ISBN: 978-3-642-27918-8. DOI: 10.1007/978-3-642-27919-5 (cit. on p. 9).

[87] Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer Berlin Heidelberg, 1999. DOI: 10.1007/978-3-662-03809-3 (cit. on p. 73).

[88] Peter Lammich. "Generating Verified LLVM from Isabelle/HOL". In: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 22:1–22:19. DOI: 10.4230/LIPICS.ITP.2019.22 (cit. on p. 10).

[89] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X (cit. on p. 11).

[90] Rijkswaterstaat. *Landelijke Tunnelstandaard (National Tunnel Standard)*. Accessed: 2025-07-03. 2012. URL: https://standaarden.rws.nl/link/standaard/6080-1-0 (cit. on pp. 58, 60, 61).

[91] Sophie Lathouwers. "Exploring annotations for deductive verification". English. PhD thesis. Netherlands: University of Twente, 2023-10. ISBN: 978-90-365-5845-7. DOI: 10.3990/1.9789036558464 (cit. on pp. 32, 145).

[92] Sophie Lathouwers and Vadim Zaytsev. "Modelling program verification tools for software engineers". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*. Ed. by Eugene Syriani, Houari A. Sahraoui, Nelly Bencomo, and Manuel Wimmer. ACM, 2022, pp. 98–108. DOI: 10.1145/3550355.3552426 (cit. on pp. 8, 9).

[93] K Rustan M Leino and Michał Moskal. *Usable auto-active verification*. 2010. URL: https://fm.csl.sri.com/UV10/index.shtml (visited on 2025-05-20). (Internet Archive link) (cit. on p. 6).

[94] Jannis Limperg and Asta Halkjær From. “Aesop: White-Box Best-First Proof Search for Lean”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023.* Ed. by Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic. ACM, 2023, pp. 253–266. DOI: 10.1145/3573105.3575671 (cit. on p. 6).

[95] Ling Liu and M. Tamer Özsu, eds. *Encyclopedia of Database Systems, Second Edition.* Springer, 2018. ISBN: 978-1-4614-8266-6. DOI: 10.1007/978-1-4614-8265-9 (cit. on pp. 15, 16).

[96] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. “Protocol-based verification of message-passing parallel programs”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015.* Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 280–298. DOI: 10.1145/2814270.2814302 (cit. on p. 147).

[97] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. “Armada: low-effort verification of high-performance concurrent programs”. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020.* Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 197–210. DOI: 10.1145/3385412.3385971 (cit. on p. 12).

[98] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.)* Wiley, 2006. ISBN: 978-0-470-09355-9 (cit. on p. 138).

[99] Eduardo R. B. Marques, Francisco Martins, Vasco T. Vasconcelos, Nicholas Ng, and Nuno Martins. “Towards deductive verification of MPI programs against session types”. In: *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013.* Ed. by Nobuko Yoshida and Wim Vanderbauwhede. Vol. 137. EPTCS. 2013, pp. 103–113. DOI: 10.4204/EPTCS.137.9 (cit. on p. 111).

[100] Nicholas D Matsakis and Felix S Klock II. “The Rust language”. In: *ACM SIGAda Ada Letters.* Vol. 34. ACM. 2014, pp. 103–104 (cit. on p. 69).

[101] Kazutaka Matsuda and Meng Wang. ““Bidirectionalization for free” for monomorphic transformations”. In: *Sci. Comput. Program.* 111 (2015), pp. 79–109. DOI: 10.1016/J.SCICO.2014.07.008 (cit. on p. 135).

[102] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. “VeriSolid: Correct-by-Design Smart Contracts for Ethereum”. In: *Financial Cryptography and Data Security.* Cham, Switzerland: Springer, 2019-09, pp. 446–465. DOI: 10.1007/978-3-030-32101-7_27 (cit. on p. 78).

[103] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, Mahieddine Dellabani, Axel Legay, and Saddek Bensalem. “*S* BIP 2.0: Statistical Model Checking Stochastic Real-Time Systems”. In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 536–542. DOI: 10.1007/978-3-030-01090-4_33 (cit. on p. 146).

[104] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. 2021-06. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (cit. on p. 111).

[105] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023. DOI: 10.1017/9781108981491 (cit. on pp. 15, 23, 24, 45, 48, 92, 116).

[106] Raúl E. Monti, Robert Rubbens, and Marieke Huisman. “On Deductive Verification of an Industrial Concurrent Software Component with VerCors”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I* (ISoLA). Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13701. Lecture Notes in Computer Science. Springer, 2022, pp. 517–534. DOI: 10.1007/978-3-031-19849-6_29 (cit. on pp. 19, 55).

[107] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 34).

[108] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2 (cit. on pp. 6, 34).

[109] Thomas Sebastiaan Neele. “Reductions for parity games and model checking”. English. Proefschrift. PhD thesis. Mathematics and Computer Science, 2020-09. ISBN: 978-90-386-5089-0 (cit. on p. 14).

[110] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. “Scaling symbolic evaluation for automated verification of systems code with Serval”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed.

by Tim Brecht and Carey Williamson. ACM, 2019, pp. 225–242. DOI: 10.1145/3341301.3359641 (cit. on p. 32).

[111] Chris Newcombe. “Why Amazon Chose TLA +”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. Ed. by Yamine Aït Ameur and Klaus-Dieter Schewe. Vol. 8477. Lecture Notes in Computer Science. Springer, 2014, pp. 25–39. DOI: 10.1007/978-3-662-43652-3_3 (cit. on p. 12).

[112] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. “SPY: Local Verification of Global Protocols”. In: *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*. Ed. by Axel Legay and Saddek Bensalem. Vol. 8174. Lecture Notes in Computer Science. Springer, 2013, pp. 358–363. DOI: 10.1007/978-3-642-40787-1_25 (cit. on p. 111).

[113] Nicholas Ng and Nobuko Yoshida. “High performance parallel design based on session programming”. In: *MEng thesis, Department of Computing, Imperial College London* (2010). URL: https://www.doc.ic.ac.uk/teaching/distinguished-projects/2010/n.ng.pdf (cit. on p. 147).

[114] Nicholas Ng and Nobuko Yoshida. “Pabble: parameterised Scribble”. In: *Serv. Oriented Comput. Appl.* 9.3-4 (2015), pp. 269–284. DOI: 10.1007/S11761-014-0172-8 (cit. on pp. 121, 125, 135, 138).

[115] Jeremy W. Nimmer and Michael D. Ernst. “Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java”. In: *Electronic Notes in Theoretical Computer Science* 55.2 (2001). RV’2001, Runtime Verification (in connection with CAV ’01), pp. 255–276. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)00256-7 (cit. on p. 78).

[116] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. ISBN: 978-3-319-10541-3. DOI: 10.1007/978-3-319-10542-0 (cit. on pp. 24, 27, 28).

[117] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9. URL: https://doi.org/10.1007/3-540-45949-9 (cit. on p. 24).

[118] Bashar Nuseibeh. “Soapbox: Ariane 5: Who Dunnit?” In: *IEEE Softw.* 14.3 (1997), pp. 15–16. DOI: 10.1109/MS.1997.589224 (cit. on p. 1).

[119] Gerard O’Regan. *Mathematical Foundations of Software Engineering - A Practical Guide to Essentials*. Texts in Computer Science. Springer, 2023. ISBN: 978-3-031-26211-1. DOI: 10.1007/978-3-031-26212-8 (cit. on p. 9).

[120] Wytse Oortwijn and Marieke Huisman. “Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System”. In: *Integrated Formal Methods*. Ed. by Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa. Cham: Springer International Publishing, 2019, pp. 418–436. ISBN: 978-3-030-34968-4 (cit. on pp. 57, 58).

[121] Wytse Oortwijn, Marieke Huisman, Sebastiaan J. C. Joosten, and Jaco van de Pol. “Automated Verification of Parallel Nested DFS”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Armin Biere and David Parker. Cham: Springer International Publishing, 2020, pp. 247–265. ISBN: 978-3-030-45190-5 (cit. on p. 57).

[122] Wytse Hendrikus Marinus Oortwijn. “Deductive techniques for model-based concurrency verification”. PhD thesis. Netherlands: University of Twente, 2019-12. ISBN: 978-90-365-4898-4. DOI: 10.3990/1.9789036548984 (cit. on p. 9).

[123] Susan S. Owicki. “Axiomatic Proof Techniques for Parallel Programs”. PhD thesis. Cornell University, USA, 1975. URL: https://hdl.handle.net/1813/6393 (cit. on p. 26).

[124] Matthew Parkinson. “Class invariants: The end of the road”. In: *Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO)* 9 (2007). URL: https://people.dsv.su.se/~tobias/iwaco/p3-parkinson.pdf (cit. on p. 43).

[125] Lawrence C. Paulson and Jasmin Christian Blanchette. “Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers”. In: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*. Ed. by Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska. Vol. 2. EPiC Series in Computing. EasyChair, 2010, pp. 1–11. DOI: 10.29007/36DT (cit. on p. 6).

[126] João C. Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix A. Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, and Adrian Perrig. *Protocols to Code: Formal Verification of a Next-Generation Internet Router*. 2024. DOI: arXiv:2405.06074 (cit. on p. 145).

[127] Francesco Protopapa. *Verifying Kotlin Code with Viper by Controlling Aliasing*. 2024. URL: https://hdl.handle.net/20.500.12608/70919 (cit. on p. 146).

[128] Olivia Proust and Frédéric Loulergue. “Verified Scalable Parallel Computing with Why3”. In: *Software Engineering and Formal Methods - 21st International Conference, SEFM 2023, Eindhoven, The Netherlands, November 6-10, 2023, Proceedings*. Ed. by Carla Ferreira and Tim A. C. Willemse. Vol. 14323. Lecture Notes in Computer Science. Springer, 2023, pp. 246–262. DOI: 10.1007/978-3-031-47115-5_14 (cit. on p. 112).

[129] Victor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. “Code generation for Event-B”. In: *Int. J. Softw. Tools Technol. Transf.* 19.1 (2017), pp. 31–52. DOI: 10.1007/S10009-015-0381-2 (cit. on p. 10).

[130] Robert Rubbens. *Improving Support for Java Exceptions and Inheritance in VerCors*. 2020-05. URL: http://essay.utwente.nl/81338/ (cit. on p. 72).

[131] Robert Rubbens, Petra van den Bos, and Marieke Huisman. *Artifact of: VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory*. 2024-08. DOI: 10.5281/zenodo.13348213 (cit. on pp. 20, 173).

[132] Robert Rubbens, Petra van den Bos, and Marieke Huisman. *VeyMont Permission Annotations Tic-Tac-Toe Case Studies and Tool Implementation*. 2024. DOI: 10.5281/zenodo.13348214 (cit. on pp. 95, 106).

[133] Robert Rubbens, Petra van den Bos, and Marieke Huisman. "VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory". In: *Integrated Formal Methods - 19th International Conference, IFM 2024, Manchester, UK, November 13-15, 2024, Proceedings* (iFM). Ed. by Nikolai Kosmatov and Laura Kovács. Vol. 15234. Lecture Notes in Computer Science. Springer, 2024, pp. 217–236. DOI: 10.1007/978-3-031-76554-4_12 (cit. on pp. 20, 91, 117–119, 122, 124, 134, 136).

[134] Robert Rubbens, Petra van den Bos, and Marieke Huisman. *Artefact of: Verified Parameterized Choreographies*. 2025-04. DOI: 10.5281/zenodo.14900264 (cit. on pp. 21, 173).

[135] Robert Rubbens, Petra van den Bos, and Marieke Huisman. "Verified Parameterized Choreographies". In: *Coordination Models and Languages* (COORDINATION). Ed. by Cinzia Di Giusto and António Ravara. Cham: Springer Nature Switzerland, 2025, pp. 50–69. DOI: 10.1007/978-3-031-95589-1_3 (cit. on pp. 21, 115).

[136] Robert Rubbens, Petra van den Bos, and Marieke Huisman. *Verified Parameterized Choreographies Technical Report*. 2025. DOI: 10.48550/arXiv.2502.15382 (cit. on pp. 21, 127, 134).

[137] Robert Rubbens, Sophie Lathouwers, and Marieke Huisman. "Modular Transformation of Java Exceptions Modulo Errors". In: *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings*. Ed. by Alberto Lluch-Lafuente and Anastasia Mavridou. Vol. 12863. Lecture Notes in Computer Science. Springer, 2021, pp. 67–84. DOI: 10.1007/978-3-030-85248-1_5 (cit. on p. 34).

[138] Robert Rubbens, Larisa Safina, Petra van den Bos, Simon Bliudze, and Marieke Huisman. *Artefact of: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java*. 2023. DOI: 10.4121/21763274.v1 (cit. on p. 173).

[139] Robert Rubbens, Petra Van den Bos, and Marieke Huisman. *Artefact of: Verified Parameterized Choreographies*. 2025. DOI: 10.5281/zenodo.14900264 (cit. on pp. 119, 126, 139).

[140] Bernhard Rumpe. *Modeling with UML*. Springer, 2016. ISBN: 978-3-319-33932-0. DOI: 10.1007/978-3-319-33933-7 (cit. on p. 11).

[141] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick G. Kourie, and Bruce W. Watson. "Tool Support for Correctness-by-Construction". In: *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by

Reiner Hähnle and Wil M. P. van der Aalst. Vol. 11424. Lecture Notes in Computer Science. Springer, 2019, pp. 25–42. DOI: 10.1007/978-3-030-16722-6_2 (cit. on pp. 8, 10).

[142] Mohsen Safari and Marieke Huisman. "Formal verification of Parallel Prefix Sum and Stream Compaction algorithms in CUDA". In: *Theoretical Computer Science* (2022). ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2022.02.027 (cit. on p. 57).

[143] Mohsen Safari, Wytse Oortwijn, Sebastiaan Joosten, and Marieke Huisman. "Formal Verification of Parallel Prefix Sum". In: *NASA Formal Methods.* Ed. by Ritchie Lee, Susmit Jha, and Anastasia Mavridou. Cham: Springer International Publishing, 2020, pp. 170–186. ISBN: 978-3-030-55754-6. URL: https://doi.org/10.1007/978-3-030-55754-6_10 (cit. on p. 57).

[144] Ömer Sakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. "Alpinist: An Annotation-Aware GPU Program Optimizer". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II.* Ed. by Dana Fisman and Grigore Rosu. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 332–352. DOI: 10.1007/978-3-030-99527-0_18 (cit. on p. 94).

[145] Ömer Şakar. *Extending support for Axiomatic Data Types in VerCors.* 2020-04. URL: http://essay.utwente.nl/80892/ (cit. on p. 72).

[146] *SonarQube homepage.* accessed on: 2022-05-18. 2022. URL: https://www.sonarqube.org/ (cit. on p. 72).

[147] Nataliia Stulova, Jos F. Morales, and Manuel V. Hermenegildo. "Reducing the overhead of assertion run-time checks via static analysis". In: *PPDP '16.* Association for Computing Machinery, 2016-09, pp. 90–103. ISBN: 978-1-45034148-6. DOI: 10.1145/2967973.2968597 (cit. on p. 78).

[148] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. "Dependent types and multi-monadic effects in F". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655 (cit. on p. 111).

[149] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. "SteelCore: an extensible concurrent separation logic for effectful dependently typed programs". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 121:1–121:30. DOI: 10.1145/3409003 (cit. on p. 111).

[150] Philip Tasche, Raúl E. Monti, Stefanie Eva Drerup, Pauline Blohm, Paula Herber, and Marieke Huisman. “Deductive Verification of Parameterized Embedded Systems Modeled in SystemC”. In: *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*. Ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Vol. 14500. Lecture Notes in Computer Science. Springer, 2024, pp. 187–209. DOI: 10.1007/978-3-031-50521-8_9 (cit. on p. 145).

[151] *Technolution webpage*. Accessed April 2022. 2022. URL: https://www.technolution.eu (cit. on p. 57).

[152] The White House. *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. 2024 (cit. on p. 12).

[153] Mark Timmer, Hendrik Brinksma, and Mariëlle Ida Antoinette Stoelinga. “Model-Based Testing”. In: *Software and Systems Safety: Specification and Verification*. Vol. 30. NATO Science for Peace and Security Series D: Information and Communication Security. IOS Press, 2011-04, pp. 1–32. ISBN: 978-1-60750-710-9. DOI: 10.3233/978-1-60750-711-6-1 (cit. on p. 70).

[154] VerCors team. *VerCors tutorial*. 2025. URL: https://vercors.ewi.utwente.nl/wiki (visited on 2025-05-14). (Internet Archive link) (cit. on pp. 37, 38, 44).

[155] VerifyThis team. *VerifyThis Collaborative Long-term Verification Challenge: The Casino example*. (Accessed at: 2022-10-12). 2022. URL: https://verifythis.github.io/casino/ (cit. on p. 83).

[156] Bruce W. Watson, Derrick G. Kourie, Ina Schaefer, and Loek Cleophas. “Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience?” In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9952. Lecture Notes in Computer Science. 2016, pp. 730–748. DOI: 10.1007/978-3-319-47166-2_52 (cit. on p. 7).

[157] Nobuko Yoshida and Lorenzo Gheri. “A Very Gentle Introduction to Multiparty Session Types”. In: *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*. Ed. by Dang Van Hung and Meenakshi D’Souza. Vol. 11969. Lecture Notes in Computer Science. Springer, 2020, pp. 73–93. DOI: 10.1007/978-3-030-36987-3_5 (cit. on p. 138).

[158] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. “The Scribble Protocol Language”. In: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*. Ed. by Martín Abadi and Alberto Lluch-Lafuente. Vol. 8358. Lecture Notes in Computer Science. Springer, 2013, pp. 22–41. DOI: 10.1007/978-3-319-05119-2_3 (cit. on p. 111).

[159] Nobuko Yoshida and Vasco Thudichum Vasconcelos. “Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication”. In: *Proceedings of the First International Workshop on Security and Rewriting Techniques, SecReT@ICALP 2006, Venice, Italy, July 15, 2006*. Ed. by Maribel Fernández and Claude Kirchner. Vol. 171. Electronic Notes in Theoretical Computer Science. Elsevier, 2006, pp. 73–93. DOI: 10.1016/J.ENTCS.2007.02.056 (cit. on p. 138).

[160] Xindi Zhang, Bohan Li, and Shaowei Cai. “Deep Combination of CDCL(T) and Local Search for Satisfiability Modulo Non-Linear Integer Arithmetic Theory”. In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, 125:1–125:13. DOI: 10.1145/3597503.3639105 (cit. on p. 33).

[161] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. “Statically verified refinements for multiparty protocols”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 148:1–148:30. DOI: 10.1145/3428216 (cit. on p. 111).

Appendix

Publications by the Author A

- Robert Rubbens, Sophie Lathouwers, and Marieke Huisman. “Modular Transformation of Java Exceptions Modulo Errors”. In: *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings*. Ed. by Alberto Lluch-Lafuente and Anastasia Mavridou. Vol. 12863. Lecture Notes in Computer Science. Springer, 2021, pp. 67–84. DOI: 10.1007/978-3-030-85248-1_5.

- Raúl E. Monti, Robert Rubbens, and Marieke Huisman. “On Deductive Verification of an Industrial Concurrent Software Component with VerCors”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I* (ISoLA). Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13701. Lecture Notes in Computer Science. Springer, 2022, pp. 517–534. DOI: 10.1007/978-3-031-19849-6_29.

- Simon Bliudze, Petra van den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. “JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java”. In: *Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings* (FASE). Ed. by Leen Lambers and Sebastián Uchitel. Vol. 13991. Lecture Notes in Computer Science. Springer, 2023, pp. 143–150. DOI: 10.1007/978-3-031-30826-0_8.

- Lukas Armborst, Pieter Bos, Lars B. van den Haak, Marieke Huisman, Robert Rubbens, Ömer Sakar, and Philip Tasche. “The VerCors Verifier: A Progress Report”. In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14682. Lecture Notes in Computer Science. Springer, 2024, pp. 3–18. DOI: 10.1007/978-3-031-65630-9_1.

- Robert Rubbens, Petra van den Bos, and Marieke Huisman. “VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory”. In: *Integrated Formal Methods - 19th International Conference, IFM 2024, Manchester, UK, November 13-15, 2024, Proceedings* (iFM). Ed. by Nikolai Kosmatov and Laura Kovács. Vol. 15234. Lecture Notes in Computer Science. Springer, 2024, pp. 217–236. DOI: `10.1007/978-3-031-76554-4_12`.

- Wolfgang Ahrendt, Jonas Becker-Kupczok, Simon Bliudze, Petra van den Bos, Marco Eilers, Gidon Ernst, Martin Fabian, Paula Herber, Marieke Huisman, Raúl E. Monti, Robert Rubbens, Larisa Safina, Jonas Schiffl, Alexander J. Summers, Mattias Ulbrich, and Alexander Weigl. “From Model Checking to Deductive Verification: Results from a Smart Contract Community Challenge”. In: *International Journal on Software Tools for Technology Transfer (STTT)* (2025). Under submission.

- Robert Rubbens, Petra van den Bos, and Marieke Huisman. “Verified Parameterized Choreographies”. In: *Coordination Models and Languages* (COORDINATION). Ed. by Cinzia Di Giusto and António Ravara. Cham: Springer Nature Switzerland, 2025, pp. 50–69. DOI: `10.1007/978-3-031-95589-1_3`.

- Robert Rubbens, Petra van den Bos, and Marieke Huisman. *Verified Parameterized Choreographies Technical Report*. 2025. DOI: `10.48550/arXiv.2502.15382`.

Research Data Management | B

The following research code repositories have been produced during this research:

- Chapter 4: Rubbens, Safina, Van den Bos, Bliudze, and Huisman (2023).
  Artefact of: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java
  Extension of the VerCors tool, extension of the JavaBIP tool. Permanently available on 4TU.ResearchData.
  doi: `https://doi.org/10.4121/21763274.v1`
- Chapter 5: Rubbens, Van den Bos, and Huisman (2024).
  Artifact of: VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory
  Extension of the VeyMont tool. Permanently available on Zenodo.
  doi: `https://doi.org/10.5281/zenodo.13348213`
- Chapter 6: Rubbens, Van den Bos, and Huisman (2025).
  Artefact of: Verified Parameterized Choreographies
  Case study code and scripts. Permanently available on Zenodo.
  doi: `https://doi.org/10.5281/zenodo.14900264`

Development versions of VerCors and VeyMont are available on `https://github.com/utwente-fmt/vercors`.

Samenvatting

Software is alomtegenwoordig geworden, zowel in de industrie als daarbuiten. Helaas zijn er tal van voorbeelden van softwarekwetsbaarheden die het dagelijks leven op grote schaal ontwrichten, zoals de Heartbleed-kwetsbaarheid in 2014 of de wereldwijde Crowdstrike-storing in 2024. Daarom is het waarborgen van de correctheid van deze softwaresystemen cruciaal.

Dit is echter moeilijk. Alleen al in een geïsoleerd proces zijn geheugensveiligheid en functionele correctheid lastig te verifiëren. Daar komt nog bij dat in een softwareomgeving waarbij processen gelijktijdig worden uitgevoerd, processen door elkaar worden geweven. Hierdoor worden sommige kwetsbaarheden alleen geactiveerd bij specifieke doorvlechtingen van processen. Dit maakt het lastiger om handmatig kwetsbaarheden te vinden, en dus nog moeilijker om correcte software te schrijven.

Een manier om de correctheid van software met gelijktijdige processen te waarborgen is via formele methoden, die nagaan of programma's voldoen aan een wiskundige specificatie. Geautomatiseerde tools die deze technieken implementeren, bieden gebruikers de mogelijkheid om de correctheid van hun software op ongekende schaal aan te tonen. Dit proefschrift richt zich op de *auto-actieve* deductieve verifieërder VerCors, die programma's met gelijktijdige processen verifieërt door middel van contracten voor geheugensveiligheid en functionele correctheid.

Hoewel er succesvolle toepassingen zijn geweest van formele methoden op industriële systemen, is het gebruik van formele methoden in de praktijk nog steeds beperkt. We stellen vast dat dit deels komt door een kloof tussen de mentale modellen van ontwikkelaars en de abstracties die formele methoden bieden. We verbeteren de situatie door formele methoden te combineren met, en uit te breiden voor, verschillende ontwikkelmethoden en abstractieniveaus. Deze *hybride* formele methoden hebben het potentieel om de kloof tussen praktische softwareontwikkeling en verificatie met formele methoden te verkleinen. We verkennen dit thema in drie delen.

Het eerste deel van dit proefschrift onderzoekt de mogelijke redenen voor het uitblijven van het gebruik van één specifieke formele methode: auto-actieve deductieve verificatie. We passen de deductieve verifieërder VerCors toe op een industrieël stuurprogramma van het bedrijf Technolution en vinden twee kwetsbaarheden. We concluderen dat de software ondersteuning voor formele methoden verder verbeterd moet worden en dat het moeilijk is om de mentale modellen van ontwikkelaars te verbinden met het abstractieniveau dat vereist is voor verificatie-annotaties.

In het tweede deel van het proefschrift proberen we de kloof tussen mentale modellen en formele methoden te verkleinen door twee verificatietools te combineren: VerCors en het componentgebaseerde ontwikkelraamwerk JavaBIP. De kernfunctie van JavaBIP is

dat het de *implementatie* van componenten in Java scheidt van de *interactie* tussen componenten. We noemen de combinatie van JavaBIP en VerCors *Verified JavaBIP*, waarmee implementaties van JavaBIP-modellen worden geverifieerd op geheugensveiligheid en functionele correctheid. We implementeren ondersteuning voor Verified JavaBIP in de deductieve verifieërder VerCors, en implementeren ook ondersteuning voor dynamische verificatie in JavaBIP. We illustreren Verified JavaBIP aan de hand van de VerifyThis Long Term Verification Challenge.

In het derde deel van dit proefschrift behandelen we choreografieën en deductieve verificatie. Choreografieën zijn notatie voor het beschrijven van protocollen en voor gedistribueerde systemen. In choreografieën hebben berichten altijd het juiste dataformaat en hoeven deelnemers nooit oneindig op berichten te wachten. Choreografieën ondersteunen ook het genereren van code die het beschreven systeem of protocol implementeert. Om choreografieën te verifiëren is de tool VeyMont ontworpen, waarmee verificatie van het gedistribueerde systeem in één gecombineerd overzicht mogelijk is. In dit proefschrift maken we VeyMont breder toepasbaar door het uit te breiden met gedeeld geheugen en door het aantal deelnemers te parameteriseren.

Aanvankelijk ondersteunde VeyMont geen gedeeld geheugen. Dit beperkte de expressiviteit en maakte bewijsstappen die een gedeelde "schaduw"-toestand vereisen onmogelijk. We maken gebruik van gedeeld geheugen mogelijk door *gestratificeerde permissies* toe te voegen aan VeyMont, een nieuw type annotatie dat geheugenannotaties aan deelnemers toekent. VeyMont gebruikt gestratificeerde permissies ook om annotaties tijdens codegeneratie te behouden, zodat de gegenereerde code verifieërbaar is met VerCors. Dit vergroot de robuustheid en onderhoudbaarheid van de gegenereerde code. We verifiëren een boter-kaas-en-eieren-choreografie op drie niveaus van optimalisatie, wat een afweging aantoont tussen het volume aan annotaties dat nodig is om de choreografie te verifiëren en de prestaties als de code wordt uitgevoerd.

We breiden VeyMont choreografieën ook uit met parameterisatie. Voorheen moest de gebruiker het aantal deelnemers vooraf specificeren. Dit maakte het moeilijk om in VeyMont gedistribueerde systemen uit te drukken die van nature kunnen groeien. We breiden VeyMont uit met ondersteuning voor choreografieën met een geparameteriseerd aantal deelnemers. We leggen bescheiden beperkingen op aan de notatie van choreografieën om automatische verificatie te behouden, en illustreren de uitbreiding door een choreografie te verifiëren die gedistribueerd sommeert. Dit toont aan dat, ondanks de opgelegde beperkingen, de voorgestelde ondersteuning voldoende is om interessante choreografieën te verifiëren.

Samenvattend onderzoekt dit proefschrift de kloof tussen formele methoden en software ontwikkeling in de industrie. Dat doen we door een deductieve verifieërder in een industriële context toe te passen en vast te stellen wat er ontbreekt aan de huidig beschikbare techniek. Daarnaast combineren we formele methoden met verschillende ontwikkelmethoden en abstracties, en breiden we zulke hybride formele methoden uit, om de kloof tussen formele methoden en softwareontwerp verder te verkleinen. Dit brengt formele methoden dichter bij de industrie, en zal daarom op de lange termijn de betrouwbaarheid van softwaresystemen verbeteren.

Titles in the IPA Dissertation Series since 2022