DEDUCTIVE TECHNIQUES

for Model-Based Concurrency Verification

Wytse Oortwijn

Deductive Techniques for Model-Based Concurrency Verification

Wytse Oortwijn

Deductive Techniques for Model-Based Concurrency Verification

Dissertation

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus
Prof.dr. T.T.M. Palstra,
on account of the decision of the Doctorate Board,
to be publicly defended
on Thursday the 12th of December 2019 at 16:45

by

Wytse Hendrikus Marinus Oortwijn

born on the 20th of December 1989
in Zwolle, the Netherlands

This dissertation has been approved by:

Prof.dr. M. Huisman (promotor)

DIGITAL SOCIETY INSTITUTE

DSI Ph.D. Thesis Series No. 19-021
Digital Society Institute
P.O. Box 217, 7500 AE
Enschede, the Netherlands

IPA Dissertation Series No. 2019-13
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 VerDi (Verification of Distributed software) project, funded by the NWO-TOP grant 612.001.403.

ISBN: 978-90-365-4898-4
ISSN: 2589-7721 (DSI Ph.D. Thesis Series No. 19-021)
DOI: 10.3990/1.9789036548984
Available online at `https://doi.org/10.3990/1.9789036548984`

Typeset with LaTeX
Printed by Ipskamp Printing
Cover design by Wytse Oortwijn

Graduation Committee:

Chairman:	Prof.dr. J.N. Kok
Promotor:	Prof.dr. M. Huisman

Members:

Prof.dr. W. Ahrendt	Chalmers University of Technology, Sweden
Dr. C.E.W. Hesselman	University of Twente, the Netherlands
Prof.dr. G.K. Keller	Utrecht University, the Netherlands
Dr. N. Kosmatov	CEA, List, Software Reliability Lab, France
Prof.dr. J.C. van de Pol	University of Twente, the Netherlands

Acknowledgments

While writing these acknowledgments I look back on seven fantastic years of living and working in Twente, four of which devoted to doing a PhD at the FMT research group, and three to doing a Masters at the same group. This was a very rewarding time in which I was fortunate enough to meet and work with many great people as well as visit many interesting places. I would like to extend my gratitude to the people that supported me and made this possible.

First of all, I would like to thank my supervisor, Marieke Huisman, for giving me the chance of doing a PhD at the FMT research group and for your great support, guidance and advice throughout these last four years. Many thanks for giving me the freedom and trust to pursue research topics that I found interesting, such as interactive theorem proving, even though it sometimes took a while before any fruitful results came out of it. Eventually we got several papers out, which form the basis of this thesis, and I am very happy with the result. I also greatly appreciate your generosity in allowing me to attend many interesting conferences, seminars and summer schools, and visit: Oregon, Thessaloniki, Uppsala, Wadern, Turin, Heidelberg, Orlando, Gothenburg, Stockholm and Krakow. Thanks for taking me along on the 'Avond van Wetenschap & Maatschappij' in 2017, which was a great experience. You are always very well organised and I think your management skills are outstanding! I hope we will continue collaborating for many years.

I would also like to express my gratitude to Stefan, Dilian, Sebastiaan and Jaco for fruitful technical discussions regarding various parts of this thesis, and for their collaboration. The work in this thesis could not have achieved the same quality without you. Stefan, thanks for being the main developer of VerCors, the verifier around which a substantial part of my research is centered. Dilian and Sebastiaan, thanks for our technical discussions and for helping me with getting various bits and parts of the formalisations and proofs to work. Jaco and Sebastiaan, thanks for our collaboration in verifying parallel NDFS. I think we did some really interesting

work there and I am proud to have it included in my thesis.

Many thanks also to the members of my graduation committee: Wolfgang, Cristian, Gabriele, Nikolai and Jaco, for taking the time to read and assess my thesis, and for providing helpful suggestions and comments about further improvements.

It has been a pleasure being part of the Formal Methods and Tools (FMT) research group. I thank all (ex-)FMT members for their help, for the social activities we did together and for making me feel welcome. Do keep the Friday afternoon BOCOMs up and active; those were always very enjoyable! I also enjoyed our chats, lunches, coffee breaks and drinks, as well as Vincent's and Stefano's numerous homemade bakes and pies over the years to celebrate yet another accepted paper. Stefano, it was a great experience to have participated in the Batavierenrace twice; many thanks for organising those. Arnd, thanks for being the main organiser of last years FMT outing, which was a great success, and for making such outstanding Grünkohl. Freark, I really enjoyed our puzzle game nights, and if ever the 'Vrijhof burger' reappears we should definitely have a go! Lesley, last year we went on an amazing roadtrip together through England and Scotland. Those really were great times, I hope we will have more of such roadtrips. Jeroen, thanks for helping me out with the final preparations for my defence, some of which are difficult for me to do from Zürich. I still remember quite well that we did 'Principles of Model Checking' together all the way back in 2012, which was the very first course that I took and completed in Twente. Vincent, we started our PhD projects pretty much at the same time, and before that we worked together during the Masters project in an Honours Research programme, which was really nice. And I'd like to give special thanks to Ida, for helping me out a great many times with administrative and organisational tasks. I appreciate that your door was always open for a chat.

During my PhD I had the great opportunity to visit Wolfgang for three months in Gothenburg. Wolfgang, many thanks for inviting me and making me feel welcome at Chalmers. I really enjoyed living and working in Gothenburg. This visit resulted in a fruitful collaboration with Christian, Mauricio and Ludovic, to whom I'd like to extend my gratitude. I also thank Gerardo Schneider for his amazing hospitality.

I find the program verification community a very pleasant and friendly community to be working in, for which I am grateful. I have met with many great and inspiring people during my PhD. Mattias Ulbrich, thanks for joining me for a drink on my birthday at Orlando airport in December 2017, right before I had to spend the rest of my birthday on the plane back to Amsterdam. Gary Leavens, thanks for helping me out a bit when my credit card got blocked during my stay in Orlando. Moreover, I thank the organisers of the OPLSS'16 Summer School in Oregon, as well as the organisers of the IPA events: the IPA Autumn Days were always fun!

I thank Peter Müller for giving me the opportunity to work as a PostDoc at ETH

Zürich. I've only been working here for about 1.5 months at the time of writing, but I am really enjoying it. Thanks to the Viper group for making me feel welcome!

I am lucky to have made some very good friends while living in Enschede. Many thanks in particular to Koen, Laura, Elmer, Nick, Gertjan, and Inge for the many days and evenings of hanging out, bouldering and/or playing board games. Nick en Gertjan, thank you for your willingness to take on the role of paranymphs during my defence. To Koen, for proofreading my thesis and providing feedback. To Martijn (Mu), Peter (Sjefke), Daniel (Oboema), Berend (Kevin) and Pim, for several great and memorable vacations. And especially to Amy, for having been an amazing roommate for the last two years. Really enjoyed it! I miss our many funny moments together, our cookery, the terrible films and TV shows we liked to watch, as well as our two cats, Chip and Dito. I hope to see you all at my defence!

Hilde, I feel really happy to have met you, yet sorry for leaving to Zürich so shortly after! Thank you very much for being there, as well as your ability to always make me smile, even during the stressful times of finalising my thesis and switching jobs.

Finally, I would like to greatly thank my family for supporting me and for always being there for me. Mom and Dad, I know I've often been busy and cranky during these last few years, as doing a PhD is never easy, but I am very grateful that I was always welcome for rest and support in Luttenberg. Roelie, Wim, Suzan, Haiko and Boris, many thanks for all your amazing help and for always being there!

Wytse Oortwijn
Zürich, November 4, 2019

Abstract

Software has integrated deeply into modern society, not only for small conveniences and entertainment, but also for safety-critical tasks. As we increasingly depend on software in our daily life, it becomes increasingly important that such software systems are both reliable and correct with respect to their intended behaviour. However, providing any guarantees about their reliability and correctness is very challenging, as software is developed by humans, who by nature make mistakes. This challenge is further complicated by the increasing demand for parallelism and concurrency, to match the developments in processing hardware. Concurrency makes software even more error-prone, as the concurrent interactions between different subsystems typically constitute far too many behaviours for programmers to comprehend. Software developers therefore need formal techniques that aid them to understand all possible system behaviours, to ensure their reliability and correctness.

This thesis contributes towards such formal techniques, and focusses in particular on deductive verification: a software verification approach based on mathematical logic. In deductive verification, the intended behaviour of software is specified in a program logic, allowing the use of (semi-)automated tools to verify whether the code implementation adheres to this specified behaviour, in every possible scenario.

More specifically, the work in this thesis builds on Concurrent Separation Logic (CSL), a program logic that specialises in reasoning about concurrent programs, targeting properties of functional correctness and safe memory usage. In recent years, there has been tremendous progress on both the theory and practice of CSL-based program verification. Nevertheless, many open challenges remain. This thesis focusses on one such challenge in particular, namely on *how to verify global functional properties of real-world concurrent software, in a sound and practical manner.*

This thesis consists of three parts, each of which addresses the above challenge

from a slightly different perspective.

In Part I, we investigate how CSL can be used to mechanically verify the correctness of parallel model checking algorithms. Model checking is an alternative approach for verifying software, which relies on exhaustively searching through all possible system behaviours, to check whether they satisfy a given temporal specification. The underlying search procedures are typically algorithmic, and are often parallelised for performance reasons. However, to avoid a false sense of safety, it is essential that these highly-optimised search algorithms are correct themselves. We contribute the first mechanical verification of a parallel graph-based model checking algorithm, called nested depth-first search (NDFS). This verification has been performed using VerCors: an automated verifier that uses CSL as its logical foundation. We also demonstrate how our mechanised proof of correctness supports the easy verification of various optimisations of parallel NDFS.

Part II of this thesis contributes a practical abstraction technique for verifying global behavioural properties of shared-memory concurrent software. Our technique builds on the insight that concurrent program behaviour cannot easily be specified on the level of source code. This is because realistic programming languages have only very little algebraic behaviour, due to their advanced language constructs. Instead, our approach allows specifying their behaviour as a mathematical model, with an elegant algebraic structure. More specifically, we use process algebra as the modelling language, where the actions are abstractions of shared-memory updates in the program. Furthermore, we extend CSL with logical primitives that allow one to prove that a program refines its process-algebraic model. These refinement proofs solve the typical abstraction problem: establishing whether the model is a sound abstraction of the modelled program. This abstraction approach is proven sound with help of the Coq proof assistant, and is implemented in the VerCors verifier. We demonstrate our approach on various examples, including a classical leader election protocol, as well as a real-world case study from industry: the formal verification of a safety-critical traffic tunnel control system that is currently employed in Dutch traffic.

In Part III we lift our abstraction technique to the distributed case, by adapting it to verify message passing concurrency. This adaption uses process-algebraic models to abstract communication behaviour of distributed agents. Moreover, we investigate how the refinement proofs allow deductive verification to be combined with model checking, by analysing program abstractions using a model checker, viz. mCRL2, to reason indirectly about the program's message passing behaviour. This combination builds on the insight that deductive verification and model checking are complementary techniques: the former specialises in verifying data-oriented properties, while the latter targets temporal properties of control-flow. Such a combined verification approach is therefore a natural fit for reasoning about dis-

tributed systems, as these generally deal with both computation (data) and communication (control-flow). Our approach is compositional, is mechanically proven sound with help of the Coq proof assistant, and is implemented as an encoding in the Viper concurrency verifier.

Altogether, this thesis makes a major step forward towards the *practical and reliable* verification of global behavioural properties of real-world concurrent and distributed software. The techniques proposed in this thesis are: *reliable*, by having mechanically proven correctness results in Coq; are *expressive*, as they are compositional and build on mathematically elegant structures; and are *practical*, by being implemented in automated concurrency verifiers.

Contents

CHAPTER 1

Introduction

Software has integrated deeply into modern society. In our everyday life, we make heavy use of software systems, either directly or indirectly, sometimes consciously and often unconsciously. For example, the cheese, tea and milk we may have for breakfast ended up in our fridge as a result of a series of logistical processes, most of which have been planned and controlled by smart algorithms. Then, before going to work, we may check our phones for the weather forecast in order to adapt our clothing, and on the way we let the traffic lights guide us, whose underlying software ensures our safe arrival (assuming that all participants in traffic obey the imposed traffic rules). While at work, we may use the internet to upload or download required documents, or to communicate with colleagues. Our desk, coffee mug, and even the office building itself are the result of computer-aided design and production.

These few examples already illustrate how easy it is to overlook how deeply software has integrated into society, not only for small conveniences and entertainment, but also for safety-critical tasks. Society nowadays depends heavily on software. This poses a very relevant question: to what extent can software be relied upon, and what is the impact of software failure?

Software is inherently error-prone, as it is developed by humans, who by nature make mistakes. Studies have shown that modern software contains 1 to 16 bugs on average in every 1.000 lines of code [OW02, OWB04], already in sequential (single-threaded) software, even after being tested[1]. It may not be so harmful to encounter a software problem while, say, playing a computer game, yet the

[1] Even though software testing can help to reduce the number of bugs in software, it cannot give any guarantees regarding the absence of bugs. This is also supported by the famous quote *"Program testing can be used to show the presence of bugs, but never to show their absence!"* of Edsger Dijkstra, 1969.

occurrence of one in, for example, medical/hospital systems or (air)traffic control systems may have fatal consequences.

There are many classical (in)famous examples of such software disasters resulting from human-made mistakes, including the faulty Therac-25 radiation therapy machine in 1985 [LT93], the Intel pentium FDIV flaw in 1994 [Pra95], and the exploding Ariane 5 airbus in 1996 [LLF+96]. There are also many recent cases of significant software failures, three of which are highlighted below.

1. In 2009, the emergency software system of the Ketelbrug (an 800m long bridge spanning the Ketel-lake in the Netherlands) faulted[2], causing a passenger car to hit the bridge while partially opened. There are various other known cases of software problems in Dutch bridge control systems, like the Merwedebrug in Gorinchem in 2011, which remained open for 2,5 hours due to failing software[3]. In 2019, the Dutch Ministry of Infrastructure and Water Management declared that the control software for bridges and locks in the Afsluitdijk (a major 32km long dam in the Netherlands) is unreliable and contains serious errors[4].
2. In 2013, the internet banking software of the Dutch ING bank seriously malfunctioned[5]. This prevented online access to banking services, money transferring included, and reportedly even altered the balance of bank accounts. Many webstores suffered financially from the inability to transfer money.
3. In 2018, the rail traffic around Schiphol shut down after a shoplifter ran into the restricted airport tunnel[6]. This prevented the railway software to assign arrival platforms to inbound trains, causing it to crash entirely due to an integer overflow after having attempted 32.000 such platform assignments. Ultimately over 70.000 passengers got delayed as result of this software crash.

To prevent such software failures in a society that increasingly relies on software dependability, availability, predictability and correctness, research is *much*-needed

[2] J. de Rooij, *Softwarefout veroorzaakte ongeluk Ketelbrug.* Computable, March 7, 2011. `https://www.computable.nl/artikel/nieuws/security/3814774/250449/softwarefout-veroorzaakte-ongeluk-ketelbrug`.

[3] *Brug Gorinchem 2,5 uur open door softwarefout.* NOS, August 13, 2011. `https://nos.nl/video/264106-brug-gorinchem-2-5-uur-open-door-softwarefout`.

[4] C. van Nieuwenhuizen Wijbenga, *Kamerbrief over bediening sluizen en bruggen Afsluitdijk.* January 31, 2019. `https://www.rijksoverheid.nl/documenten/kamerstukken/2019/01/31/bediening-sluizen-en-bruggen-afsluitdijk`.

[5] A. Eigenraam, *Grote storing bij ING - klanten in paniek wegens afwijkend saldo.* NRC, April 3, 2013. `https://www.nrc.nl/nieuws/2013/04/03/grote-storing-bij-ing-klanten-melden-afwijkend-saldo`.

[6] M. Duursma, *Na 32.000 signalen ging het systeem klapperen.* NRC, August 25, 2018. `https://www.nrc.nl/nieuws/2018/08/25/na-32000-signalen-ging-het-systeem-klapperen`.

to guarantee that safety/business-critical software meets these qualitative aspects, in *every* possible scenario!

The work in this thesis falls into this category of research, and is about software reliability, targeting *parallel, concurrent, and distributed software* in particular: software that performs multiple tasks simultaneously, possibly using multiple physical cores, potentially on different physical machines, connected via some network.

Concurrency in Software

To make matters on software reliability even more challenging and complex, it is an increasingly common practice for software developers to utilise *parallelism and concurrency*, to increase performance and make optimal use of the available hardware resources. This is in sync with modern trends in hardware development: since transistors on processing units cannot be made much smaller due to physical limitations, hardware manufacturers instead increase the number of transistors per processor.

Gordon Moore made a very famous prediction in 1965 [Moo65] regarding these hardware trends, stating that *"the number of transistors on a chip will double every 18 months"*, which is widely known as Moore's Law. However, Moore's Law is now ending; even though Moore's prediction remained valid for multiple decades, it started to slow down roughly around 2010 [Pep17], and has slowed down further ever since. In practice this means that CPUs cannot be made much faster. Instead, hardware manufacturers produce *multi-core* processors—CPUs with multiple processing cores—to cope with the increasing demand of computing power. These multi-core CPUs allow to perform different computations in parallel (at the same time), and therewith obtain computational speedup.

However, these multi-core processors influence the way software is written. To effectively utilise multiple cores, programmers must write their software with multitasking in mind, by clearly identifying what parts of the computation can be executed concurrently. This way of writing software is known as *multithreading*.

The use of parallelism and concurrency makes software extra sensitive to bugs and errors. This is because the (non-deterministic) interactions of different concurrent software components typically constitute an *immense* number of different possible behaviours (usually exponential in the number of concurrent system components). Too many behaviours for a software developer to be able to comprehend. This makes finding errors in concurrent software a very challenging and daunting task, as software bugs tend to reside in only very few of these behaviours.

As a consequence, the current standard in software development industry is to make concessions between performance and correctness [GJS+15, AB18]. Software

developers are generally very reluctant to use parallelism or concurrency, in order to keep their codebase better understandable, maintainable and testable. This is in sharp contrast with the trends and developments in computing hardware, which primarily aim to increase the opportunities for parallelism and concurrency.

To bridge this discrepancy between software and hardware developments, software developers need tools and techniques that aid them to understand and manage all possible system behaviours, so that concurrency can safely been employed.

This thesis contributes *formal techniques and tools* that are based on *deductive verification*, that provide mathematically precise guarantees on the reliability of parallel and concurrent software.

1.1 Formal Software Verification

Over the last 50 years there has been tremendous research on *formal techniques and tools* to improve, or guarantee, the reliability of software systems [O'R08, BH14b]. These techniques are *formal* in the sense that they are based on mathematics, allowing them to give mathematically precise correctness results. Techniques for formal verification typically allow to define a *specification* for the software system, capturing its intended behaviour, and then *verify* that the system implementation, or an abstraction of it, adheres to the specified behaviour. The verification step is often computer-aided, to be able to reason automatically about the many different behaviours that software systems may conceal, with the help of a (semi-)automated verification tool.

This thesis focuses primarily on *deductive verification*. More specifically, this thesis contributes *abstraction techniques* that allow to deductively verify concurrent and distributed program behaviour, on a global level. We also investigate how these abstraction techniques can be combined with *model checking*, which is an alternative, algorithmic approach to formal software verification. Furthermore, this thesis investigates how deductive verification techniques can be used to verify the correctness of multi-core model checking algorithms, to increase the reliability of their verdicts.

The remainder of this section gives an overview of the field of deductive verification; first for sequential software (§1.1.1), and then for concurrent software (§1.1.2). After that, §1.1.3 briefly elaborates on model checking, before Section 1.2 discusses the various challenges in these two research fields that this thesis addresses.

1.1.1 Deductive Verification of Sequential Software

Deductive verification is a formal technique to reason about software systems, that has its roots in mathematical logic. In deductive verification, the intended behaviour of software is specified in a *program logic*, allowing the use of (semi-)automated tools to verify whether the system implementation adheres to this specified behaviour, in every possible scenario. These tools reduce the problem of verifying program correctness (with respect to the specified behaviour) to a statement of mathematical logic, which can automatically be proven, e.g., using SAT solvers [CKL04, CKSY05, IYG+08] or more recently, SMT solvers [LQ08, BMR12, Sch16].

The strength of deductive verification is that it can reason precisely about *all* possible software behaviours[7]. Deductive verifiers can therefore provide guarantees about, e.g.: memory safety, freedom of concurrency errors like data-races, and correctness with respect to the intended system behaviour.

Hoare Logic (1969)

The pioneers of deductive verification are Tony Hoare and Robert Floyd, by their contribution of *Hoare logic*, in 1969 [Flo67, Hoa69] (also known as Floyd–Hoare logic). Hoare logic provides a formal technique to reason about the correctness of simple, sequential imperative programs.

The central logical components of Hoare logic are *Hoare triples*, which are of the form $\{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$, where C is a program, and $\mathcal{P}$ and $\mathcal{Q}$ are *logical assertions*, traditionally in first-order (predicate) logic. These Hoare triples give an axiomatic meaning to programs, by describing their semantics as a simple proof system, whose rules are generally referred to as *Hoare (inference) rules*. Hoare triples logically describe the effect of C on the program state, in terms of the assertions $\mathcal{P}$ and $\mathcal{Q}$, which are referred to as C's *precondition* and *postcondition*, respectively. These two assertions together constitute the specification of the program C, sometimes also referred to as C's *contract*. The operational meaning of Hoare triples is: starting from a state satisfying $\mathcal{P}$, the resulting state after execution and termination of C satisfies $\mathcal{Q}$.

Hoare logic reasoning is a *compositional* verification approach; the Hoare axioms and inference rules allow to compose proofs of smaller programs to construct a proof for a larger, composite program. Two examples of such rules are:

$$\frac{\{\mathcal{P}\}\, C_1\, \{\mathcal{Q}\} \qquad \{\mathcal{Q}\}\, C_2\, \{\mathcal{R}\}}{\{\mathcal{P}\}\, C_1; C_2\, \{\mathcal{R}\}} \qquad \frac{\{\mathcal{P} \wedge b\}\, C_1\, \{\mathcal{Q}\} \qquad \{\mathcal{P} \wedge \neg b\}\, C_2\, \{\mathcal{Q}\}}{\{\mathcal{P}\}\, \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2\, \{\mathcal{Q}\}}$$

[7] With respect to some chosen formal operational semantics of the target language.

The above inference rules allow to compose the individual proofs of the programs C_1 and C_2 into a proof of a composite program, e.g., $C_1; C_2$. While composing proofs in this way, *proof obligations* are generated, which can be proven to conclude correctness of the composite program. Dijkstra showed that these proof obligations can be proven mechanically, using SAT or SMT solvers, by using a predicate transformer semantics known as Dijkstra's Weakest Precondition (WP) calculus [Dij76]. This was the first step towards automated, tool supported deductive software verification.

Separation Logic (2002)

Classical Hoare logic is fairly limited, in the sense that it does not easily allow to reason about programs with shared mutable state. This limitation is overcome by *separation logic* [Rey00, ORY01, IO01, Rey02, O'H19a], which is a program logic that extends Hoare logic by adding logical constructs to reason about pointers: data stored on the heap. Separation logic builds on earlier ideas of Burstall [Bur71], and uses an assertion language that is a special case of the resource logic of Bunched Implications (BI) [OP99, IO01].

One of these new logical constructs is the assertion $\ell \mapsto v$, which expresses that the heap contains the value v at heap location ℓ. Another new logical construct is the connective $\mathcal{P} * \mathcal{Q}$ known as the *separating conjunction*, which expresses that $\mathcal{P}$ and $\mathcal{Q}$ are valid on *disjoint* parts of the heap. These two constructs can be used together to reason about pointer aliasing. To give an example, $\ell_1 \mapsto 3 * \ell_2 \mapsto 4$ implies that $\ell_1 \neq \ell_2$ (i.e., ℓ_1 and ℓ_2 are not aliases), since if they were aliases, the heap could not be split into disjoint parts, so that both parts satisfy the corresponding sub-assertion.

Separation logic has been applied to reason about realistic pointer-manipulating programming languages, like Java and C, and has also shown to be mechanisable, e.g., via symbolic execution [BCO05]. One example of such a mechanisation is the static program analyser Infer [CD11], which is used by Facebook to detect memory leaks and null pointer dereferences in their production code [CDD$^+$15]. Infer, however, cannot prove functional properties on the behaviour of the program.

Current State-of-the-Art

The initial developments of Hoare logic, separation logic, and (automated) weakest precondition reasoning inspired tremendous research in the field of deductive verification; not only to propose more elaborate proof systems that target advanced language features [Par10, KJB$^+$17], such as object-orientation, parallelism and (fine-grained) concurrency, but also to develop verification tools that target real-world programming languages [Bey19, EHMU19].

Notably, around 2000 the KeY project started [KeY], which led to the development of the KeY verification system, aiming to verify sequential Java, annotated with Hoare-style specifications. Other tools followed, including: Dafny [Lei10] (on which IronClad [HHL+14] and IronFleet [HHK+15] are build), OpenJML [Cok14], Frama-C [KKP+15], Why3 [FP13], KIV [RSSB98], and many others.

These tools all have their own specialisation, e.g., by targeting a certain programming language, or by supporting a specific logic. OpenJML, for example, targets sequential Java programs that are annotated with specifications written in *JML*, which stands for Java Modeling Language [LBR99, LPC+07]—an extensive specification language that is specific for Java. Frama-C, in turn, targets programs written in C, and requires program behaviour to be specified in ACSL—a specification language that is particular to C. KeY uses dynamic logic as its underlying logical foundation for static analysis, while Frama-C uses weakest precondition methods (among others).

Achievements and Challenges

These deductive verification tools have proven to be successful in practice, and have contributed to the reliability of real-world software systems. A prominent example of such a success is the detection of an intricate bug in the standard implementation of OpenJDK's `Java.utils.Collection.sort()` algorithm [GRB+15], also known as TimSort, using KeY, in 2015. This verification case study had a particular high impact, as the TimSort algorithm is used daily by billions of users worldwide. To give another example, Frama-C has successfully been applied on several safety-critical industrial case studies [CDDL12, KKP+15, SAB+16], comprising up to 50.000 lines of program code.

Nevertheless, many open challenges in deductive software verification remain. One important challenge is reducing the number of annotations (i.e., pre- and postconditions, and invariants) needed to deductively verify a program. Especially for larger programs—say, programs with $\geq$200 lines of code, which is already considered reasonably large in the deductive verification community—it is not unusual for verification tools to require more lines of specifications/annotations than actual code.

Another important such open challenge, is to reason about *real-world parallel and concurrent software*. All verification tools mentioned so far solely target sequential software, with the exception of Frama-C, who has limited support for reasoning about POSIX threads in C, using its `Mthread` plugin [YB12]. This thesis focusses primarily on how to deductively verify concurrent and distributed software, in an expressive, reliable and practical manner.

1.1.2 Deductive Verification of Concurrent Software

Concurrency reasoning is more challenging than reasoning about sequential software, since one has to deal with the additional complexity of considering all possible non-deterministic thread interactions. A concurrent program may behave in different ways, depending on how the threads are interleaved at runtime. It is possible that some of these interleavings bring undesirable concurrency events, such as *data-races*: two threads that access the same location in memory, at the same moment, where at least one of them is a write access.

The goal of concurrency verification is showing freedom of such undesired phenomena, and showing correctness with respect to a (Hoare-style) specification, in all possible thread interleavings.

Owicki–Gries (OG) Reasoning (1976)

The pioneers in the direction of deductive concurrency verification were Susan Owicki and David Gries. They contributed extensions to Hoare logic to reason about concurrent programs [OG75]. The most important extension is the following Hoare logic rule for concurrency.

$$\frac{\{\mathcal{P}_1\}\, C_1\, \{\mathcal{Q}_1\} \qquad \{\mathcal{P}_2\}\, C_2\, \{\mathcal{Q}_2\} \qquad \text{the proofs of } C_1 \text{ and } C_2 \text{ are non-interfering}}{\{\mathcal{P}_1 \wedge \mathcal{P}_2\}\, C_1 \parallel C_2\, \{\mathcal{Q}_1 \wedge \mathcal{Q}_2\}}$$

This rule allows composing the individual proofs of two programs, C_1 and C_2, into a proof of their parallel composition, $C_1 \parallel C_2$, given that these two proofs do not interfere (the notion of interference is left informal for now). Intuitively, the proofs of C_1 and C_2 are non-interfering if the proof $\{\mathcal{P}_1\}\, C_1\, \{\mathcal{Q}_1\}$ is stable under modifications done by the program C_2, and vice versa.

A major limitation of OG reasoning, is that the non-interference condition makes the logic non-modular. The classical example that shows this, is the proof of the program $x := x + 1 \parallel x := x + 1$, consisting of two threads that increment a shared variable by one. To satisfy the non-interference condition, *auxiliary state* needs to be maintained, by writing extra annotations purely for the purpose of specification, to specify the exact contributions of both threads as a global property. However, for this classical example, the amount of auxiliary state needed is exponential in the number of proof obligations. Moreover, OG reasoning is also non-compositional: adding a third thread may require one to change the proof of the other two threads. The OG approach therefore does not scale to real-world industrial code.

Rely-Guarantee (RG) Reasoning (1983)

Cliff Jones proposed a concurrency reasoning approach in 1983, known as rely-guarantee (RG) reasoning [Jon83], that improves on the classical Owicki–Gries approach. RG reasoning targets concurrent programs in which threads *are* allowed to interfere. The RG approach is also modular. Instead of requiring auxiliary state, RG requires extra specifications for each thread, that express the *reliances* on the environmental threads under which the current thread executes, as well as *guarantees* that the thread makes to the environmental threads. These rely and guarantee clauses make the approach modular.

On the other hand, RG reasoning is relatively hard to apply in practice. In addition to the standard Hoare-style specifications, users also have to give a separate specification of thread interference, which is often non-intuitive, and non-trivial to come up with and to specify.

Concurrent Separation Logic (2007)

Later, in 2007, *Concurrent Separation Logic (CSL)* has been proposed by O'Hearn [O'H07] and Brookes [Bro07]. CSL is a program logic that extends separation logic to reason thread-modularly about shared-memory concurrent programs. This is done using the following proof rule, that allows reasoning independently about threads that access disjoint parts of shared memory.

$$\frac{\{\mathcal{P}_1\}\, C_1\, \{\mathcal{Q}_1\} \qquad \{\mathcal{P}_2\}\, C_2\, \{\mathcal{Q}_2\}}{\{\mathcal{P}_1 * \mathcal{P}_2\}\, C_1 \parallel C_2\, \{\mathcal{Q}_1 * \mathcal{Q}_2\}}$$

More specifically, CSL uses the separating conjunction from separation logic to express that C_1 and C_2 work on disjoint portions of the heap. This implies freedom of data races, for any program for which a proof can be derived. To give an example, CSL allows proving $\{x \mapsto - * y \mapsto -\}\, [x] := 3 \parallel [y] := 4\, \{x \mapsto 3 * y \mapsto 4\}$, where $[\,\cdot\,]$ denotes heap dereferencing, as the specification implies that x and y are not aliases. For this reason, the above proof rule allows decomposing this proof into two smaller proofs: $\{x \mapsto -\}\, [x] := 3\, \{x \mapsto 3\}$ and $\{y \mapsto -\}\, [y] := 4\, \{y \mapsto 4\}$.

As illustrated already by the above rule and example, CSL comes with a strong notion of *ownership* of shared memory. The specification of threads need to be very explicit in the memory footprint that is needed by the thread's programs, using the $\cdot \mapsto \cdot$ points-to assertion of separation logic. However, in many practical realistic scenarios, threads do not work on purely disjoint memory, but instead work together on a shared portion of memory. To handle such sharing situations, CSL has support for reasoning about atomics—statically-scoped locks—that al-

low threads to obtain access permission for a shared part of the heap, without introducing data races and without breaking thread-modularity.

Modern Logics for Concurrency Reasoning

CSL has had significant impact on the field of concurrency verification [BO16, PSO18], both in theory and in practice. For this reason, Brookes and O'Hearn received the Gödel Prize in 2016, for their invention of CSL. The work in this thesis also builds heavily on CSL.

One of these advances is the merge of CSL with rely-guarantee reasoning, by Vafeiadis and Parkinson in 2007, resulting in a program logic called RGSep [VP07]. This logic simplifies the specification of thread-interference with respect to classical RG reasoning, by exploiting the notion of disjointness that CSL offers. RGSep is supported by the tools SmallFootRG [CPV07] and CAVE [Vaf10a, Vaf10b]. Furthermore, Gotsman et al. [GBC+07] propose extensions to CSL to deal with dynamically-scoped locks. In 2009, *deny-guarantee* was proposed [DFPV09], which is a program logic that deals with dynamic thread creation and fork/join concurrency. This line of research extends further to very elaborate concurrency logics, like CAP (Concurrent Abstract Predicates) [DYDG+10, SB14], TaDa (for abstracting time and data) [RPDYG14, RPDYG15], and ultimately Iris [JSS+15, JKBD16, KJB+17], a higher-order CSL framework. Ilya Sergey maintains a CSL family tree online [CFT], that gives a more complete overview of concurrency logics that extend CSL.

Even though these modern program logics and frameworks are mathematically elegant and very expressive, their usage requires much expertise and insight into the underlying proof method. Moreover, at the time of writing, most of these logics can only be used in pen-and-paper style, or at best semi-automatically, in the context of interactive proof assistants like Coq [CWP, BC10] and Isabelle/HOL [NWP02].

To be able to target real-world programming languages like Java and C, it is important that such program logics are applicable *automatically*, and on the *level of source code*, with help of automated verification tools.

Modern Tools for Concurrency Verification

Several such verification tools have been proposed, most of which build on SMT solvers like Z3, to discharge all generated verification conditions. Compared to theoretical/interactive approaches like for example Iris, such automated tools make a trade-off between expressivity and usability: they do not require user interaction, other than providing a specification (e.g., in pre/postcondition style).

An example of such an automated tool is VeriFast [JSP+11], a verifier that tar-

gets single- and multi-threaded Java and C programs that are annotated with pre- and postconditions written in separation logic. Another such tool is the Viper verifier [JKM+14, MSS16], which provides a verification infrastructure based on separation logic[8], that makes it easy to implement intricate verification techniques for programs with persistent mutable state. The VerCors verifier [BDHO17], which is maintained at the University of Twente, builds on top of Viper, and targets concurrent programs written in Java [AHHH15] and OpenCL [ADBH15] (i.e., GPU kernels). VerCors is logically based on CSL and performs a correctness-preserving translation of any input verification file to the Viper language. This allows delegating the generation of verification conditions to Viper and their verification ultimately to Z3 [MB08].

Despite much effort in research on tool-aided concurrency verification, there are still many open challenges [ZS15, HH17, HJ18]. This thesis focusses on one such open challenge in particular, namely on: *how to verify global functional properties of real-world concurrent software, in a reliable and practical manner.*

The current standard approach of proving global properties of concurrent program behaviour is to specify *global invariants*: logical assertions that remain preserved after the execution of every instruction in the code. However, finding global invariants may sometimes be non-intuitive and cumbersome, and can also be restrictive. For example, in some situations it may be desirable to temporarily violate a global invariant, provided that environmental threads are not able to observe this violation. Sometimes these global invariants take the form of transition systems, like in CAP, or more abstractly, as monoids, like in Iris. Nevertheless, as discussed before, this line of work is mostly theoretical and hard to implement into automated tools.

In contrast, this thesis contributes *practical* techniques that allow to specify global concurrent program behaviour abstractly, as a mathematical model, with elegant algebraic structure. These abstraction techniques are practical, by making a trade-off between expressivity and usability. Rather than aiming for a unified approach to concurrency reasoning, we propose powerful and sound techniques that are implemented in automated concurrency verifiers, to be able to reason about realistic, real-world programming languages.

> This thesis contributes *practical* and *reliable* abstraction techniques for verifying global behavioural properties of real-world concurrent and distributed software.

[8] Or actually a program logic called *Implicit Dynamic Frames* [LMS09], which is shown to be equivalent to separation logic [PS11].

1.1.3 Model Checking

An alternative approach for reasoning about concurrent program behaviour is *model checking* [Cla08, CHVB18]. This is a field of research that was started by Clarke and Emerson in 1981 [CE82], and independently by Quielle and Sifakis in 1982 [QS82].

Model checkers consider an *abstract model* of a software system, often given as a finite transition system, and automatically verify properties on this model. These properties are typically specified in a modal/temporal logic, like (variants/extensions of) LTL [Pnu77], CTL [EC80, CE82], or, more generally, the μ-calculus [EC80, Koz82]. This automated verification is done *algorithmically*, rather than deductively, by means of exhaustively searching the underlying state space of the model. These exhaustive searches yield a counter-example in case the specified property does not hold, in the form of a trace in the search space, that represents the violated behaviour.

The main advantage of model checking over deductive verification is that it provides more automation. This is because, unlike deductive verifiers, model checkers generally do not search for a correctness proof of the system under verification[9], but instead explore its underlying state space and analyse all possible execution traces. The only ingredients needed by a model checker are an abstract description of the software system, and a specification of this system in a temporal logic.

One the other hand, a well-known effect of this exhaustive search is the problem of *state-space explosions*. This problem refers to the combinatorial explosion of the size of the search space, in the number of variables and parallel components in the input model. This effect limits the scalability of model checking to real-world industrial software. Moreover, model checking suffers from the well-known *abstraction problem* [PGS01]: does the abstract model soundly reflects the behaviour of the actual system it models?

Combining Model Checking with Deductive Verification

Regarding expressivity, model checkers target different kinds of properties than deductive verifiers. Deductive verification is primarily concerned with reasoning about properties that are *data-oriented* [ACPS17], that is, properties that relate the output of functions to their input (for example, that the `sort(`xs`)` function yields a sorted permutation of the input sequence xs). Model checkers, on the other hand, are mostly concerned with temporal properties of *control-flow* [Sha18], for example, every `logout(`u`)` event must be preceded by a `login(`u`)` of the same

[9] Modern model checking approaches like IC3 [CG12] actually do this, but in a more limited and restrictive manner compared to the deductive verification techniques mentioned earlier.

user u. Even though model checkers often have some limited support for handling data, verifying data-oriented properties is not their primary strength nor concern (partly because they are to a large extend finite-state approaches). For this reason, model checking and deductive verification have shown to be *complementary* in nature [MN95, Uri00, ACPS15, ACPS17, Sha18].

This thesis investigates how model checking and deductive verification can be combined, to exploit their complementarity and resolve the abstraction problem of model checking, for the verification of concurrent and distributed software.

1.2 Challenges in Concurrency Verification

The overview of formal software verification given in the previous section already poses various open challenges for practical concurrency reasoning. This thesis focusses on one such challenge in particular, namely on: *how to verify global behavioural properties of real-world concurrent software, in a reliable and practical manner.*

This thesis is organised into the following three parts, each of which addresses the above challenge from a slightly different perspective:

I. Reliability of software verification techniques. Part I of this thesis investigates how concurrent separation logic can be used to mechanically verify the correctness of heavily optimised, parallel model checking algorithms.

II. Verifying functional properties of shared-memory concurrent software. Part II investigates how global behavioural properties of *shared-memory* concurrent programs can be specified and mechanically be verified, by means of abstraction.

III. Verification of distributed software using complementary techniques. Part III investigates how deductive verification and model checking, which have shown to be complementary techniques, can be combined to verify global behavioural properties of *message-passing distributed* software.

The remainder of this section discusses each of these three challenges in detail. It also presents the problem statements and the three research questions that are addressed in the three parts of this thesis.

1.2.1 Reliability of Software Verification Techniques

As discussed earlier, nowadays there are many techniques for software verification, as well as (semi-)automated tools that support these. Modern verification techniques no longer aim to verify simple artificial languages (e.g., the simple language that Hoare logic is formalised on), but instead increasingly aim to target intricate, real-world language features, ranging from relaxed/weak memory models [vVZN+11, LV15, LGV16, LVK+17, SM18], to compiler optimisations [VBC+15, DBG18], to correct compilation stacks [Ler09, Chl10, KMNO14, LKT+19], as well as real-world modern programming languages like Rust [JJKD17], Python [EM18], and Go [VSC]. Consequently, the underlying mathematical principles of these verification techniques become more and more complex, to deal with these advanced features. This poses a relevant challenge: how to ensure that the mathematical principles of these techniques remain sound, while their complexity increases.

Furthermore, the verification tools that implement these principles are themselves written in software, developed by humans. These tools tend to follow trends in hardware and software development, for example to parallelise their implementations, so that search spaces of large systems can be explored faster. The LTSmin model checker [KLM+15], for example, has multiple parallel back-ends (both explicit-state [LPW11] and symbolic [DP15]), to be able to process billions of states per second. This raises the challenge: how to ensure that their implementations are correct themselves, to prevent such tools from giving a false sense of safety?

Machine-Checked Verification Tools

A popular trend to increase the confidence of the correctness of modern program logics is to embed them into interactive proof assistants, like Coq, Isabelle, or PVS [ORS92]. Such interactive theorem provers in turn guarantee the correctness of their own results, by trusting on a very small critical kernel of axioms, whose validity is widely agreed upon [ARSCT09, HN18].

To give some examples, recent program logics like CAP, Iris, Disel [SWT17], Iron [BGKB19] and Aneris [KJTOL19] all have mechanically checked soundness proofs in Coq, and are all implemented as a shallow embedding in Coq. Another example is the Refinement Framework [LT12, Lam13], which has its roots in Isabelle/HOL.

As a consequence, the ability to reason with these logics is also confined to the Coq or Isabelle environment, in an interactive, semi-automated way, via the use of tactics. This makes it hard to reason about realistic programs, written in real-

world programming languages, as one would then have to guide the interactive proof through all the intricate details of the underlying operational semantics of the targeted programming language, which does not scale very well.

However, here we should remark that, despite this potential issue of scalability, the VerifyThis software verification competition [EHMU19] has been won twice in a row already by an Isabelle team, using the Refinement Framework; in 2018 and in 2019. This success is mainly because interactive proof assistants give better control on the generated proof obligations, and on selecting the strategy for proving these, compared to automated verifiers. Nevertheless, the Refinement Framework solely targets sequential programs that are written in an Isabelle-embedded language. In order to scale to real programming languages, the degree of proof automation should also scale. For exactly this reason, automated verifiers make a compromise between automation and control: they give the verification engineer less control on how the verification conditions are actually proven, in exchange for better automation.

In contrast to interactive verifiers, it is far less common for automated deductive verifiers to have a trusted, machine-checked foundation. This is primarily because automated verifiers aim to target real-world industrial programming languages, e.g., Java and C, whose semantics is very difficult to formally specify, let alone to build upon[10]. Most automated verifiers instead build on mathematical principles that are manually proven sound, and discharge all generated proof obligations to SMT solvers like CVC4 and Z3. However, some verification tools invested in a machine-checked correctness proof of a core subset of their theoretical foundation, e.g., Featherweight VeriFast [JVP15] and Smallfoot [App11b] (the latter in the context of the Verified Software Toolchain [App11a]). Arguably the most realistic and manageable approach for ensuring that automated verifiers are reliable themselves is to mechanise the proofs of their metatheory with a proof assistant.

This thesis also follows the latter, more realistic approach, by contributing verification techniques for concurrency that are proven sound using the Coq theorem prover, and are implemented in the concurrency verifier VerCors.

Verifying Parallel Model Checking Algorithms

For deductive verification it is relatively easy to provide such machine-checked correctness results of the underlying foundations, as these build on mathematical logic. For other formal techniques it may be much harder to establish such a mechanically verified core. Model checkers, for example, have an algorithmic

[10]See for example the complexity of the work of Robbert Krebbers [Kre14, Kre15] on the formalisation of the C standard in Coq, or the K-Java project [BR15], which is a complete executable operational semantics for Java 1.4.

foundation. Moreover, recent model checkers like LTSmin heavily exploit parallelism to quickly search through large state spaces, and thus build on a foundation of intricate, heavily optimised parallel algorithms. Establishing the correctness of such multi-core algorithms is therefore highly non-trivial.

There is some existing work on verifying the outcome of model checkers [Nam01, GRT18], by generating a deductive proof alongside the verdict of model checking, which can be checked independently. Also several fully verified *sequential* model checkers have been proposed [Spr98, ELN+13, Neu14, BL18, WL18], whose implementation are fully machine-checked by either Coq or Isabelle. Nevertheless, to the best of our knowledge, there does not exists a mechanical verification for a *parallel* model checking algorithm (prior to this thesis).

This challenge is addressed in the current thesis, focusing in particular on *graph-based* algorithms for parallel model checking. More specifically, Part I of this thesis addresses the following research question:

> ***RQ 1:*** *How can concurrent separation logic be used to specify and mechanically verify parallel graph-based algorithms for model checking?*

1.2.2 Verifying Behavioural Properties of Concurrent Software

The main challenge of concurrency reasoning over reasoning about sequential software is that one has to deal with the vast number of potential system behaviours that are the result of the many possible thread interleavings. This makes it difficult to precisely specify thread interaction on a global level, e.g., with shared-memory or with other threads. In contrast, it is relatively straightforward to specify the global system behaviour in a sequential setting.

The standard, classical approach in concurrency verification to specify global properties is to impose *global invariants*, either on the contents of the heap [Mey88, CK05, O'H07] or on the message exchanges between threads [WL89]. The classical work on CSL, for example, has a built-in notion of global invariants, called resource invariants, which can only temporarily be violated by a thread when it holds the global lock. However, such invariant properties are limited in the sense that they are "static": they cannot easily express how system behaviour evolves over time. This is apparent in the classical Owicki–Gries example mentioned earlier, which consists of a very simple concurrent program, whose correctness proof requires a global invariant that is exponential in size, relative to the number of threads. This example clearly demonstrates that global invariants do not always scale.

To resolve this, many alternative approaches have been proposed that build on ideas of assume-guarantee (AG) reasoning [RHH+01]. Notably, modern program logics like (impredicative) CAP and Iris provide protocol-like specification mechanisms, allowing to formally define how shared state is permitted to evolve over time, in shared-memory concurrent programs. Such "protocols" typically take the form of state-transition systems (e.g., in CAP or Disel), or more abstractly, as user-defined monoids [JSS+15] (e.g., in Iris, as well as logics building on Iris). However, this line of work is mostly theoretical, or can at best be applied semi-automatically, via a shallow embedding in Coq. Making these ideas to also work with *automated* concurrency verifiers for real-world languages is still an open challenge.

This challenge is addressed in the current thesis, focussing in particular on *abstraction techniques* to specify global system behaviour, in a sound and practical manner. More specifically, Part II of this thesis gives an answer to the following research question:

RQ 2:	*How can global behavioural correctness properties of shared-memory concurrent programs be specified and mechanically verified, by means of abstraction?*

1.2.3 Combining Complementary Verification Techniques for Reasoning About Distributed Software

As discussed earlier, there are several potential approaches to formally verifying concurrent and distributed systems, two of which are deductive verification and model checking. However, these different approaches focus on different aspects of correctness. Deductive verification focusses primarily on *data-oriented* properties, for example by relating the output of functions to their input (e.g., the function `sort` correctly sorts the input array). Algorithmic verification, on the other hand, primarily targets *control-oriented* properties, and is mostly concerned with the order in which certain atomic events may occur (e.g., users may withdraw money only after having successfully logged-in). These two notions of data and control-flow are *complementary in nature*. Of course, one could for example encode transition systems as invariants during deductive verification, or incorporate limited support for data in model checkers (which is what mCRL2 does). But such handling of data and control-flow are not the primary strengths of these respective verification approaches.

Especially in a distributed setting it would make sense for verification techniques to address both aspects of data and control-flow, as distributed programs typically deal with both computation (data) and communication (control-flow). The use of

only a single verification approach, e.g., deductive verification or model checking alone, may be insufficient to capture all program aspects.

Deductive verification, for example, has its power in modularity and compositionality: they require modular independent proofs of the (distributed) threads, and allow to compose these into a single proof of the global system. However, due to the communicational nature of distributed systems, it can be hard to give an isolated proof for each thread, as their behaviour for a large part depends on their interaction with the environment. Another option would be the imposition of *network invariants* [WL89]: global invariants that span over a network of distributed agents. However, as already discussed in §1.2.2, these are often limited in their expressivity.

Model checkers, on the other hand, have their strength in automation: they only require an abstract description of the concrete system implementation, together with a temporal specification. However, model checking is mostly a finite-state approach, which limits its ability to reason about data. Some model checking approaches, like nuXmv [CCD+14] (which is based on IC3), allow to reason about infinite domains, e.g., integers, reals and uninterpreted functions, but still in a more limited and restricted sense than deductive verification. This is primarily because the specification language has less support for data incorporation. Model checking also suffers from the typical abstraction gap: is the model a sound abstraction of the modelled system?

Practical verification techniques that exploit the complementary nature of data- and control-flow are therefore needed, but are currently relatively unexplored [APS16, ACPS17]. This thesis addresses the challenge of soundly combining algorithmic and deductive techniques for verifying distributed message-passing software in a practical, modular, and compositional manner. To do so, we further explore the abstraction techniques that Part II of this thesis contributes. In particular, we investigate if these can be used to capture the communication behaviour of message passing programs, and can subsequently be model checked, in such a way that the verified properties can soundly be projected onto program behaviour.

More specifically, Part III of this thesis answers the following research question:

RQ 3:	*How can the strengths of deductive and algorithmic verification soundly be combined, to specify and mechanically verify global behavioural properties of distributed message-passing software?*

1.3 Contributions

This thesis contributes to both theory and the practice of deductive concurrency verification. The main contributions of this thesis are threefold, as summarised below, corresponding to the three research questions posed earlier.

I. Automated verification of parallel model checking algorithms. Part I of this thesis contributes the mechanised verification of a parallel graph-based model checking algorithm, called nested depth-first search (NDFS). We prove soundness and completeness of NDFS using concurrent separation logic, and mechanise this correctness proof in VerCors. Moreover, we demonstrate how this mechanised proof allows establishing correctness of various optimisations of parallel NDFS. As far as we are aware, this is the first mechanised verification of a multi-core model checking algorithm.

II. Practical abstractions for verifying shared-memory concurrency. Part II of this thesis contributes a practical deductive verification technique to reason about global functional properties of shared-memory concurrent programs, by means of abstraction. The main idea is that concurrent program behaviour is not specified directly on the level of source code, but rather as a mathematical model, with more elegant algebraic structure. More specifically, we use process algebra as the modelling language. The key novelty is that this approach is expressive as well as practical, by being supported by the VerCors verifier. Moreover, the metatheory of this approach has been proven sound in Coq. The approach is demonstrated on various case studies, including a classical leader election protocol, as well as a real-world case study from industry: the formal verification of a safety-critical traffic tunnel control system that is currently employed in Dutch traffic.

III. Practical abstractions for verifying message-passing concurrency. In Part III, we lift this abstraction approach to the distributed setting, by using process algebra to abstract message passing behaviour of distributed threads. Moreover, we combine deductive verification with model checking, by allowing to model check the process-algebraic model for properties regarding communication, and use their results in the deductive proof of the program. This combined verification approach thereby allows reasoning about both data and communication of distributed systems. The approach is modular and compositional, proven sound with Coq, and practical by being implemented in Viper.

The remainder of this section briefly elaborates on the approach and contributions.

1.3.1 Automated Verification of Parallel Nested DFS

Part I of this thesis answers ***RQ 1***, and covers the mechanical verification of multi-core model checking algorithms using concurrent separation logic. More specifically, we mechanically verify a multi-core version of the nested depth-first search (NDFS) algorithm [CVWY92, HPY96, SE05], which solves the LTL model checking problem.

NDFS is a graph algorithm that takes a (Büchi) automaton as input, and searches for *accepting cycles* in this automaton. To clarify, automata have a notion of *accepting states* (i.e., states can be marked as being accepting), so that accepting cycles are cycles that contain at least one accepting state. This is useful, since the problem of LTL model checking can be reduced to finding accepting cycles in automata [BK08].

In 2011, Laarman et al. parallelised the NDFS algorithm [LLP+11]. This parallelised version of NDFS, referred to as *parallel NDFS* in the sequel, spawns multiple threads that all perform an instance of NDFS, while sharing information regarding (nested) search progress. Parallel NDFS is currently deployed in the LTSmin model checker.

The sharing of progress information between workers makes it very difficult to establish correctness (i.e., soundness and completeness) of parallel NDFS, already manually, as shown by the handwritten proof in the original paper. A particular difficulty of parallel NDFS, is that workers may get in each other's way, by blocking their search progress via information sharing, possibly preventing them from detecting accepting cycles. It can, however, be shown that not all accepting cycles can be missed in this way.

This thesis demonstrates that verification tools for concurrent separation logic, like Viper and VerCors, are nowadays mature enough to be able to mechanise the correctness proofs of such intricate parallel graph algorithms. More specifically, we verify soundness and completeness of parallel NDFS, using concurrent separation logic, and mechanise this proof using the VerCors concurrency verifier. While doing so, many proof steps that were left implicit in the original proof have been made explicit. This mechanised proof is inspired by the original, handwritten proof, and extends an earlier verification of *sequential* NDFS [Pol15] in Dafny. We also show how having a mechanised proof allows various optimisations of parallel NDFS to be easily verified.

To the best of our knowledge, this is the first mechanised verification of a multi-core model checking algorithm. This verification effort increases the reliability of parallel NDFS, and therewith also of the model checkers that implement this algorithm.

1.3.2 Practical Abstractions for Shared-Memory Concurrency

Part II of this thesis answers ***RQ 2***, and contributes a practical abstraction technique for verifying global behavioural properties of shared-memory concurrent software. The key idea is that such properties are not specified on the source code level, which usually has only very little algebraic behaviour (e.g., due to complex language features for locking and concurrency), but are instead specified as a mathematical model with elegant properties and structure. These models give an abstract view on concurrent program behaviour, by describing the atomic changes to the shared state, thereby hiding all irrelevant implementation details.

More specifically, our approach allows specifying the behaviour of shared-memory concurrent programs as a *process-algebraic model*. Many believe that process algebra provides a mathematically elegant way of describing concurrent program behaviour, at the right level of abstraction [ABC10]. In our approach, the actions of process-algebraic models correspond to shared-memory updates during program execution. This correspondence is formally proven: we extend CSL with primitives to deductively prove that the program *refines* its process-algebraic model, by linking the actions to concrete instructions in the program. By doing so, our approach resolves the typical abstraction problem: establishing whether the abstract model is a sound abstraction of program behaviour. The established refinement relation between programs and models preserves *safety properties*: we may reason about action sequences on the abstraction level, and project this reason onto program behaviour.

To briefly illustrate how this abstraction approach works, we revisit the Owicki–Gries (OG) example mentioned earlier, whose implementation is given in Figure 1.1a. The goal is to verify that, after termination of both threads, the value at heap location x has been incremented by 2. The traditional approach of verifying this property, is to maintain auxiliary state to keep track of the contributions of each thread. This is, however, a non-modular approach, as it requires an exponential amount of auxiliary state relative to the number of threads.

Our approach is to abstract the shared-memory update $[x] := [x] + 1$ as a process-algebraic action incr, so that the behaviour of the program can be globally described as the process $\mathsf{incr} \parallel \mathsf{incr}$. Moreover, our approach allows Hoare-style *contracts* to be assigned to each action which describe the corresponding change in the program. The contract of the incr action is: $\{\mathsf{true}\}\,\mathsf{incr}\,\{X = \backslash\mathbf{old}(X) + 1\}$, where the variable X is linked to the heap location $[x]$. One can then analyse all possible traces of $\mathsf{incr} \parallel \mathsf{incr}$ to conclude that this process indeed describes the desired OG property.

The next step is to establish that the program adheres to this process-algebraic

```
1 atomic {                    4 atomic {
2   [x] := [x] + 1;           5   [x] := [x] + 1;
3 }                           6 }
```

(a) The classical Owicki–Gries example, where $[\cdot]$ denotes heap dereferencing.

```
1 atomic {                    6 atomic {
2   action (incr) {           7   action (incr) {
3     [x] := [x] + 1;         8     [x] := [x] + 1;
4   }                         9   }
5 }                           10 }
```

(b) The above example, but now specified with so-called *action annotations*.

Figure 1.1: The Owicki–Gries example, consisting of two threads that both increment a common heap value. Figure (a) shows the example without annotations, whereas (b) gives the action annotations required by our abstraction approach.

description. This is done by means of extra program annotations, which are shown in Figure 1.1b. To clarify, *action annotations* are used to connect the concrete instructions $[x] := [x] + 1$ in the program to the incr actions in the process-algebraic model. These annotations are enough to automatically prove that the OG program refines the incr $\|$ incr process. This implies that the process-algebraic specification is a sound abstraction of the program behaviour, with respect to updates to $[x]$.

This verification approach is compositional and thread-modular. Moreover, the metatheory of this technique is mechanically proven correct using the Coq proof assistant, and thus reliable. Furthermore, this approach has been implemented in the VerCors concurrency verifier, and is therefore also practical. We demonstrate this abstraction approach on various examples, including a leader election protocol, as well as an industrial case study, concerning the formal verification of a safety-critical traffic tunnel control system.

1.3.3 Practical Abstractions for Message Passing Concurrency

Part III of this thesis answers ***RQ 3***, and builds further on the process-algebraic program abstractions, by investigating their use in a distributed setting. Here we use process algebra as a language to abstract *message passing behaviour of distributed programs* (instead of updates to shared memory, as in Part II).

In particular, we investigate how process-algebraic reasoning can be combined

```
1 send 8;          ||  4 y := recv;
2 x := recv;       ||  5 send (y * 3);
3 assert x = 24;   ||
```

Figure 1.2: A simple example of synchronous message passing, using two threads.

with deductive reasoning, to reason about both computational and communicational aspects of distributed programs. As discussed earlier, deductive verification (viz. concurrent separation logic) specialises in reasoning about computational properties, while algorithmic verifiers (e.g., model checkers like mCRL2) specialise in verifying temporal properties. We investigate how both of these complementary aspects can be combined, into a practical approach for verifying distributed programs.

To illustrate how this abstraction approach works, consider the example program of Figure 1.2. This program consists of two threads that communicate via synchronous message passing, meaning that all **send** operations block until they are matched by a **recv** in another thread. The goal is to verify whether the assertion on line 3 holds.

This program is hard to reason about using standard approaches like CSL. This is because the asserted property is inherently global: it validity depends on the computation of, and interaction with, the other thread. More advanced program logics and approaches, like assume-guarantee reasoning, might help, but the extra specifications for assumptions and guarantees from/to the environment may not always be easy to specify, especially when the program becomes more complicated.

Our approach is to specify the send/receive behaviour of this program as a process-algebraic model, for example $P_1 \parallel P_2$, where P_1 and P_2 are defined as:

$$P_1 \triangleq \mathsf{send}(8) \cdot \Sigma_x\, \mathsf{recv}(x) \cdot ?(x = 24) \qquad P_2 \triangleq \Sigma_y\, \mathsf{recv}(y) \cdot \mathsf{send}(y * 3)$$

In this approach, all send actions correspond to **send** instructions in the program, and likewise for recv and **recv**. Moreover, the assertion in the program can be modelled as the ? action. One may now proceed to analyse all traces of $P_1 \parallel P_2$ using a model checker, viz. mCRL2, to reason indirectly about the send/recv behaviour of the program. For this particular process, there is only one such trace possible, namely "$\mathsf{comm}(8) \cdot \mathsf{comm}(24) \cdot ?(24 = 24)$", where comm denotes the *synchronisation* of a send with a recv, i.e., a communication action. This trace gives evidence that the property of interest indeed holds. Finally, by using our refinement technique, we may use this property in the deductive proof of the program, to proof the assertion.

This verification approach is thread-modular and compositional, is proven sound with help of the Coq proof assistant, and is implemented in the Viper concurrency verifier.

1.4 Thesis Structure

Part I of this thesis is organised as follows:

Chapter 2 gives preliminaries on the theory of deductive verification, first in a sequential setting (Hoare logic), and then in a concurrent setting (Concurrent Separation Logic). After that, we introduce VerCors and explain how VerCors implements the concepts of CSL to reason automatically about concurrent software. The chapter concludes with a short verification example. This chapter is based on the following publications:

- S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In N. Polikarpova and S. Schneider, editors, *integrated Formal Methods (iFM)*, volume 10510 of *LNCS*, pages 102–110. Springer, 2017
- S. Joosten, W. Oortwijn, M. Safari, and M. Huisman. An Exercise in Verifying Sequential Programs with VerCors. In A. Summers, editor, *Formal Techniques for Java-like Programs (FTfJP)*. ACM, 2018

Chapter 3 answers ***RQ 1***, by contributing a formal, mechanised proof for the verification of the parallel NDFS algorithm in VerCors. The chapter first gives an informal correctness proof of parallel NDFS, and then explains how this proof is encoded and proven by VerCors. The chapter concludes by explaining how this verification allows easily verifying correctness of various optimisations of parallel NDFS. This chapter is based on the following publication:

- W. Oortwijn, M. Huisman, S. Joosten, and J. van de Pol. Automated Verification of Parallel Nested DFS. In *Submitted*, 2019

Part II of this thesis is organised as follows:

Chapter 4 answers ***RQ 2***, by illustrating our process-algebraic abstraction approach on various examples, including: a concurrent implementation of a GCD algorithm, a locking protocol, and on the formal verification of a classical leader election protocol, implemented on shared-memory. This chapter is based on the following publication:

- W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. An Abstraction Technique for Describing Concurrent Program Behaviour. In A. Paskevich and T. Wies, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 10712 of *LNCS*, pages 191–209, 2017

Chapter 5 presents the formalisation of the abstraction approach introduced in Chapter 4, consisting of (1) the syntax and semantics of the process algebra language for the abstractions; (2) the syntax and semantics of a simple programming language that is used to formalise the approach on; (3) the syntax and semantics of the assertion language; (4) the proof rules of the program logic, which extends on CSL; (5) soundness of the program logic; and (6) details on its implementation in VerCors. This chapter is based on the following publication:

- W. Oortwijn, D. Gurov, and M. Huisman. Practical Abstractions for Automated Verification of Shared-Memory Concurrency. In D. Beyer and D. Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS. Springer, 2020. To appear

Chapter 6 illustrates our abstraction approach on an industrial case study, covering the formal verification of a safety-critical tunnel system. In this case study, the mCRL2 process algebra language is used to model the state-machine specification of the tunnel system, which is provided by the Dutch government. After this, we use our abstraction technique to formally prove that the resulting process-algebraic model is a sound abstraction of the actual code implementation. We also verified some properties over this model, and found some undesired behaviour, which we could reflect back onto program behaviour. This chapter is based on the following publication:

- W. Oortwijn and M. Huisman. Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System. In W. Ahrendt and S. L. Tapia Tarifa, editors, *integrated Formal Methods (iFM)*, LNCS. Springer, 2019. To appear

Part III of this thesis is organised as follows:

Chapter 7 answers ***RQ 3***, by discussing how process-algebraic abstractions can be adapted and used in a distributed setting, to reason about message passing behaviour. In addition, we show how this combines with algorithmic analysis, allowing the use of model checking to conclude properties about message exchanges, while using deductive verification to reason about computations. This chapter is based on the following publication:

- W. Oortwijn and M. Huisman. Practical Abstractions for Automated Verification of Message Passing Concurrency. In W. Ahrendt and S. L. Tapia Tarifa, editors, *integrated Formal Methods (iFM)*, LNCS. Springer, 2019. To appear

Chapter 8 concludes the thesis, by revisiting the three research questions, and by giving an overview of further challenges and future work.

A complete list of publications by the thesis' author (including submitted ones) is provided in Appendix C on page 261.

1.4.1 Suggested Method of Reading

Figure 1.3 shows the thesis structure, and gives the suggested method of reading.

1.4.2 Sources

All definitions and proofs in this thesis have been formalised either in VerCors or Coq. The sources can be found in a Git repository available at:

`https://github.com/wytseoortwijn/SupplementaryMaterialsThesis`

We often omit details or proofs from lemmas and theorems given in this thesis, since the formalisation is quite involved. These details are deferred to the appendix, or otherwise to the encodings in VerCors or in Coq.

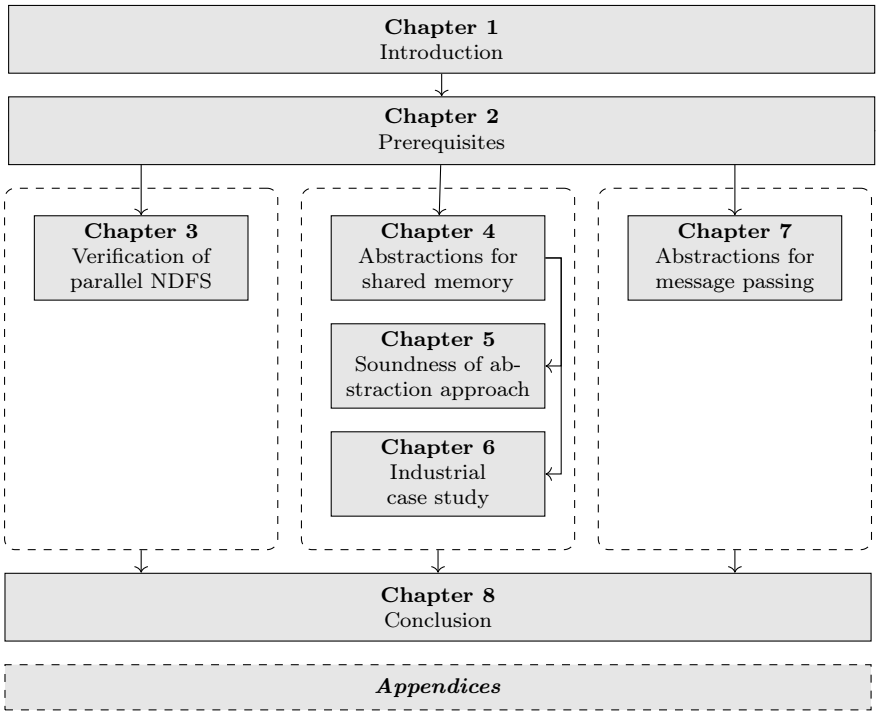


Figure 1.3: Chapter roadmap, showing the dependencies between chapters.

Part I

Background on Deductive Program Verification

CHAPTER **2**

Background on Deductive Concurrency Verification

Most of the techniques in the following chapters rely on (concurrent) separation logic, and discuss verification examples or case studies that have been worked-out in the VerCors concurrency verifier. This chapter gives the required background information on the theory and practice of concurrency verification.

Section 2.1 starts by giving background on Hoare logic and concurrent separation logic. Section 2.2 discusses how VerCors applies these theories to reason automatically about various forms of concurrency[1]. Finally, Section 2.3 demonstrates VerCors on a slightly bigger verification example that is taken from the VerifyThis 2018 verification competition: the verification of a gap buffer[2].

2.1 Deductive Software Verification

This thesis primarily studies *deductive* verification techniques to reason about both sequential and concurrent software. Deductive reasoning here means reasoning logically about the meaning of a program, by deriving logical conclusions about it based on premises or axioms that capture the meaning of its subprograms.

2.1.1 Hoare Logic

The pioneers on deductive software verification are Tony Hoare and Robert Floyd, who invented *(Floyd–)Hoare logic* in 1969 [Flo67, Hoa69]. Hoare logic provides a formal technique to reason about the correctness of sequential imperative pro-

[1] This section is based on the article [BDHO17].
[2] This section is based on the article [JOSH18].

grams. The central logical constructs of Hoare logic are *Hoare triples*, which have the form

$$\{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \tag{2.1}$$

where C is a program, and $\mathcal{P}$ and $\mathcal{Q}$ are *logical assertions*, traditionally in first-order logic. These Hoare triples give an *axiomatic meaning* to programs, by describing their semantics as a proof system. In particular, they describe the effect of the computation of the program C in terms of the assertions $\mathcal{P}$ and $\mathcal{Q}$, which are referred to as C's *precondition* and *postcondition*, respectively. The intuition of Hoare triples in an operational sense is that, starting from any program state satisfying $\mathcal{P}$, the final program state upon termination of C will satisfy $\mathcal{Q}$. This notion of correctness is called *partial correctness*. There is also a notion called *total correctness*, often written $[\mathcal{P}]\, C\, [\mathcal{Q}]$, that requires C to also necessarily terminate. This relation between the axiomatic meaning of Hoare triples and operational meaning of programs is often made explicit, by means of a soundness theorem.

Hoare logic provides a *compositional* proof system, consisting of axioms and inference rules that allow to prove that programs are correct with respect to their specifications, i.e., pre and postconditions. By compositional we mean that these inference rules follow the structure of the program: they allow to compose proofs of smaller programs to construct proofs of larger, composite programs.

We write $\vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ to indicate that a proof can be derived for the Hoare triple $\{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ by using the inference rules of Hoare logic. Furthermore, we say that a specified program $\{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ is *verified* if a proof can be derived for it, that is, if $\vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$.

The most important Hoare logic rules are listed below.

Assignment. The following axiom, named HT-ASSIGN, handles $x := e$ instructions, which assign the evaluation of the expression e to a local variable x. HT-ASSIGN describes that, to prove that an arbitrary assertion $\mathcal{P}$ holds after computing the assignment, one needs to assume that $\mathcal{P}[x/e]$ holds before the assignment, i.e., $\mathcal{P}$ with every free occurrence of x substituted by e:

$$\text{HT-ASSIGN}\quad \vdash \{\mathcal{P}[x/e]\}\, x := e\, \{\mathcal{P}\}$$

Sequential composition. The inference rule HT-SEQ for the sequential composition $C_1; C_2$ of two programs C_1 and C_2 states that, if proofs can be derived for C_1 and C_2, then these can be composed into a proof derivation for $C_1; C_2$, under the condition that C_1's postcondition coincides with C_2's precondition:

$$\text{HT-SEQ}\quad \frac{\vdash \{\mathcal{P}\}\, C_1\, \{\mathcal{Q}\} \qquad \vdash \{\mathcal{Q}\}\, C_2\, \{\mathcal{R}\}}{\vdash \{\mathcal{P}\}\, C_1; C_2\, \{\mathcal{R}\}}$$

Conditionals. The following rule for **if** b **then** C_1 **else** C_2 programs states that, if proofs can be derived for the subprograms C_1 and C_2 in which the Boolean condition b respectively holds or not holds, then these two proofs can be composed into a single proof of the composite **if**-statement:

$$\frac{\vdash \{\mathcal{P} \wedge b\}\, C_1\, \{\mathcal{Q}\} \qquad \vdash \{\mathcal{P} \wedge \neg b\}\, C_2\, \{\mathcal{Q}\}}{\vdash \{\mathcal{P}\}\, \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\, \{\mathcal{Q}\}}\ \text{HT-IF}$$

Loops. The inference rule HT-WHILE for while-loops, **while** b **do** C, requires the existence of a *loop invariant* $\mathcal{P}$, and states that, if $\mathcal{P}$ holds: (i) before entering the loop, (ii) is preserved by every iteration of the loop, and (iii) holds afterwards, then $\mathcal{P}$ is a specification for the program **while** b **do** C:

$$\frac{\vdash \{\mathcal{P} \wedge b\}\, C\, \{\mathcal{P}\}}{\vdash \{\mathcal{P}\}\, \textbf{while}\ b\ \textbf{do}\ C\, \{\mathcal{P} \wedge \neg b\}}\ \text{HT-WHILE}$$

Consequence. The rule of consequence, HT-CONSEQ, allows strengthening the precondition and weakening the postcondition of any proof derivation. The entailment $\mathcal{P} \vdash \mathcal{Q}$ means that $\mathcal{Q}$ is provably a logical consequence of $\mathcal{P}$, that is, $\mathcal{Q}$ can be proven to hold under the hypothesis that $\mathcal{P}$ holds:

$$\frac{\mathcal{P} \vdash \mathcal{P}' \qquad \vdash \{\mathcal{P}'\}\, C\, \{\mathcal{Q}'\} \qquad \mathcal{Q}' \vdash \mathcal{Q}}{\vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}}\ \text{HT-CONSEQ}$$

Dijkstra has shown in 1976 that, given any program C (in the simple language for which Hoare logic reasoning is formalised), determining whether C satisfies a given specification $\mathcal{P}, \mathcal{Q}$, i.e., $\vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$, can be automated and reduced to a problem of satisfiability solving. This is done by defining a predicate transformer semantics for programs, that determine their *weakest preconditions*. More concretely, the weakest precondition of a program C with respect to a postcondition $\mathcal{Q}$ is defined in terms of a function $\mathsf{wp}(C, \mathcal{Q})$, by structural recursion on C. A definition of wp can be found in the original work of [Dij76].

Determining weakest preconditions has shown to be an effective strategy for program verification, since proving whether $\vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ has shown to be equivalent to proving whether $\mathsf{wp}(C, \mathcal{Q})$ follows from $\mathcal{P}$:

$$\vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \qquad \textit{if and only if} \qquad \vdash \mathcal{P} \implies \mathsf{wp}(C, \mathcal{Q})$$

The latter strategy is effective, as it provides more automation than determining whether $\vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ can be derived, since, for example, application the HT-SEQ

rule requires one to find an intermediate assertion. The wp predicate transformer provides an automated technique for finding such intermediate assertions.

The theory of Hoare logic and wp-reasoning have inspired tremendous research, and form the basis of many modern tools and techniques for program verification [BH14a], including Viper and VerCors (see §2.2), which we use later, in Chapters 3–7.

2.1.2 Owicki–Gries Reasoning

The classic system of Hoare logic only allows reasoning about sequential programs. Reasoning about concurrent software is more challenging, as this requires reasoning about all possible interactions and interleaving of concurrent threads.

The first steps towards concurrency verification were made by Susan Owicki and David Gries in 1976 [OG75], by extending Hoare logic with the following inference rule, sometimes called the Owicki–Gries rule, for parallel composition:

$$\frac{\vdash \{\mathcal{P}_1\}\, C_1\, \{\mathcal{Q}_1\} \qquad \vdash \{\mathcal{P}_2\}\, C_2\, \{\mathcal{Q}_2\}}{\vdash \{\mathcal{P}_1 \wedge \mathcal{P}_2\}\, C_1 \parallel C_2\, \{\mathcal{Q}_1 \wedge \mathcal{Q}_2\}} \quad \textit{provided that the proof derivations of } C_1 \textit{ and } C_2 \textit{ are non-interfering.}$$

The exact definition of proof interference is quite technical. Intuitively, two proof derivations $\vdash \{\mathcal{P}_i\}\, C_i\, \{\mathcal{Q}_i\}$ for $i = 1, 2$ are interference-free if $\vdash \{\mathcal{P}_1\}\, C_1\, \{\mathcal{Q}_1\}$ is stable (remains derivable) under any assignments executed by C_2, and vice versa.

However, this extra condition of non-interference makes the Owicki–Gries rule non-compositional: the required information of interference is not present in the specifications of C_1 and C_2, and has to be derived separately from the proof derivations of C_1 and C_2. This non-compositionality limits the applicability of the Owicki–Gries approach. For example, from a software development perspective, it disallows developing C_1 and C_2 independently, as the Owicki–Gries proof rule requires global information of the whole composite system $C_1 \parallel C_2$ [XRH97]. This observation leads to the insight that, in order for the proof rule for parallel programs $C_1 \parallel C_2$ to be compositional, the information of thread interference should be present in the specifications of C_1 and C_2.

2.1.3 Concurrent Separation Logic

The above insight regarding compositionality motivated the development of *concurrent separation logic (CSL)* [O'H07, Bro07], an extension of Hoare logic for reasoning about concurrent heap manipulating programs. More specifically, CSL is able to reason about concurrent programs that interact with a shared heap, and can also reason about basic locking, by having a proof rule for atomic programs,

which are of the form **atomic** C. This thesis primarily focuses on CSL, or more specifically, on intuitionistic[3] CSL with support for permission accounting.

Ownership and disjointness. The two central concepts of CSL are *ownership* and *disjointness of ownerships*. CSL extends the assertion language of Hoare logic with predicates of the form $\ell \overset{\pi}{\hookrightarrow} v$, which express that the heap contains the value v at location ℓ. Moreover, $\pi \in (0,1]_{\mathbb{Q}}$ is a *fractional permission* [Boy03, BCOP05] that determines the amount of ownership that is available for ℓ: the predicate $\ell \overset{1}{\hookrightarrow} v$ provides *write access* to ℓ, whereas $\ell \overset{\pi}{\hookrightarrow} v$ with $\pi < 1$ provides *read-only access* to ℓ. We use the shorthand notation $\ell \overset{\pi}{\hookrightarrow} -$ to abbreviate $\exists v \,.\, \ell \overset{\pi}{\hookrightarrow} v$.

The notion of disjointness is implemented via the $\mathcal{P} * \mathcal{Q}$ connective, which is known as the *separating conjunction*. The assertion $\mathcal{P} * \mathcal{Q}$, which is read *"$\mathcal{P}$ and separately $\mathcal{Q}$"*, states that the ownerships expressed by $\mathcal{P}$ and $\mathcal{Q}$ are *disjoint*, e.g., disallowing them both to express write access to a common heap location. CSL comes with the following rule that allows heap ownership to be *split* and *merged*, where the notation $\dashv\vdash$ means that the entailment rule holds in both directions:

$$\frac{}{\ell \overset{\pi_1+\pi_2}{\hookrightarrow} v \dashv\vdash \ell \overset{\pi_1}{\hookrightarrow} v * \ell \overset{\pi_2}{\hookrightarrow} v} \hookrightarrow\text{-SPLIT-MERGE}$$

The key idea of CSL is that the $*$ connective can be used to express information regarding thread interference. If two programs C_1 and C_2 can be specified and proven to operate on disjoint parts of the heap, then their specifications can safely be combined into a specification of the composite program $C_1 \parallel C_2$, in a compositional manner. The $\hookrightarrow$-SPLIT-MERGE rule then allows splitting heap ownership (in the left-to-right direction), to be distributed over concurrent threads, and allows these ownerships to be merged (right-to-left) when the threads terminate and converge. Additionally, the soundness argument of CSL ensures that the total sum of permissions for any heap location does not exceed 1, which implies that any verified program is free of data races. A *data race* occurs when two threads access a common entry on the heap, where at least one is a write access.

Moreover, we present an *intuitionistic* version of CSL, which means that it includes the following entailment rule that allows "forgetting" about resources:

$$\frac{}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{P}} *\text{-WEAK}$$

Including the $*$-WEAK rule in the CSL proof system gives the advantage that one may write specifications that express properties over only parts of the heap, i.e.,

[3]Intuitionistic here means that the proofs of programs do not necessarily have full knowledge of the contents of the heap. The proofs are allowed to forget/lose knowledge of the heap, with fits naturally with the garbage collected nature of languages like Java or C#.

the exact contents of the heap does not have to be specified. Intuitionistic CSL therefore befits garbage collected languages, where the exact contents of the heap are never really known.

Program judgments. In CSL, *judgments of programs* are of the form:

$$\mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \tag{2.2}$$

where $\{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ is a standard Hoare triple, and $\mathcal{R}$ is a *resource invariant* that captures the ownerships that are protected by thread synchronisation mechanisms, such as locks. Resource invariants can only be acquired and used in the context of atomic programs, **atomic** C, as these gain exclusive access to the heap by the scheduler during program execution.

The operational meaning of a program judgment is that, starting from any state satisfying $\mathcal{P} * \mathcal{R}$, the invariant $\mathcal{R}$ is preserved throughout execution of C, and any final state upon termination of C satisfies $\mathcal{Q} * \mathcal{R}$.

The most important new proof rules of CSL w.r.t. Hoare logic are given below.

Heap allocation. The following proof rule HT-ALLOC covers $x :=$ **alloc** e instructions, with e an expression, which allocate a fresh location on the heap and write the value $[\![e]\!]$ to it (i.e., the result of evaluating e). The location of the allocated heap cell is then written to the variable x.

$$\text{HT-ALLOC}\ \frac{x \notin \mathsf{fv}(\mathcal{R}, e)}{\mathcal{R} \vdash \{\mathsf{true}\}\, x := \mathbf{alloc}\ e\, \{x \overset{1}{\hookrightarrow} e\}}$$

A heap ownership predicate $x \overset{1}{\hookrightarrow} e$ is produced by this rule, that (i) expresses write access to the allocated heap cell, and (ii) states that it holds the value represented by e. Write access is ensured, since the thread that executed the **alloc** instruction has exclusive knowledge of the existence of this new heap cell. The function fv is for obtaining the set of free program variables in its operands.

Heap reading. The instruction $x := [e]$ denotes reading from the heap, where $[e]$ denotes *heap dereferencing*, with e an expression whose evaluation determines the heap location to dereference. The proof rule HT-READ for heap reading is similar to HT-ASSIGN (page 32), but expresses that *at least* read access is needed for the dereferenced heap location.

$$\text{HT-READ}\ \frac{x \notin \mathsf{fv}(\mathcal{R}, e, e')}{\mathcal{R} \vdash \{P[x/e'] * e \overset{\pi}{\hookrightarrow} e'\}\, x := [e]\, \{\mathcal{P} * e \overset{\pi}{\hookrightarrow} e'\}}$$

Heap writing. The following proof rule, HT-WRITE, describes that heap writing $[e] := e'$ requires write access to the heap location described by e, and ensures that after its computation the heap holds the value e' at this location.

$$\frac{}{\mathcal{R} \vdash \{e \overset{1}{\hookrightarrow} -\}\, [e] := e' \,\{e \overset{1}{\hookrightarrow} e'\}}\ \text{HT-WRITE}$$

Heap deallocation. The instruction **dispose** e deallocates the heap cell at location e. Its proof rule, named HT-DEALLOC, requires exclusive (write) access to this heap location, and consumes the heap ownership predicate.

$$\frac{}{\mathcal{R} \vdash \{e \overset{1}{\hookrightarrow} -\}\, \mathbf{dispose}\ e \,\{\mathsf{true}\}}\ \text{HT-DEALLOC}$$

Note that, even though the proof system already includes the $*$-WEAK rule to forget about heap ownership, the above HT-DEALLOC rule is still needed, as it *forces* all ownership of a heap cell to be lost after its deallocation.

Parallel composition. The following proof rule states that, if proofs can be derived for the programs C_1 and C_2, stating that both operate on disjoint parts of the heap, then these can be composed into a proof for $C_1 \parallel C_2$.

$$\frac{\begin{array}{ll}\mathcal{R} \vdash \{\mathcal{P}_1\}\, C_1 \,\{\mathcal{Q}_1\} & \mathsf{fv}(\mathcal{R}, \mathcal{P}_1, C_1, \mathcal{Q}_1) \cap \mathsf{mod}(C_2) = \emptyset \\ \mathcal{R} \vdash \{\mathcal{P}_2\}\, C_2 \,\{\mathcal{Q}_2\} & \mathsf{fv}(\mathcal{R}, \mathcal{P}_2, C_2, \mathcal{Q}_2) \cap \mathsf{mod}(C_1) = \emptyset \end{array}}{\mathcal{R} \vdash \{\mathcal{P}_1 * \mathcal{P}_2\}\, C_1 \parallel C_2 \,\{\mathcal{Q}_1 * \mathcal{Q}_2\}}\ \text{HT-PAR}$$

The auxiliary $\mathsf{mod}(C)$ operation yields the set of local variables that are written to by C. The two extra premises on the right are needed for soundness, and exclude data races on the stack, i.e., data races with respect to local variables. If desired, one may get rid of these two extra conditions by treating "variables as resources" [BCY06], that is, by assigning and explicitly handling ownership of local variables, in the same style as heap ownership.

Atomics. The following proof rule for **atomic** C programs enables one to obtain the resource invariant for the proof of the inner program C.

$$\frac{\mathsf{true} \vdash \{\mathcal{P} * \mathcal{R}\}\, C \,\{\mathcal{Q} * \mathcal{R}\}}{\mathcal{R} \vdash \{\mathcal{P}\}\, \mathbf{atomic}\ C \,\{\mathcal{Q}\}}\ \text{HT-ATOM}$$

Resource invariant sharing. The proof system of CSL allows *sharing* resources, by extending the resource invariant with disjoint parts $\mathcal{R}'$ of the local state:

$$\frac{\mathcal{R} * \mathcal{R}' \vdash \{\mathcal{P}\}\, C \,\{\mathcal{Q}\}}{\mathcal{R} \vdash \{\mathcal{P} * \mathcal{R}'\}\, C \,\{\mathcal{Q} * \mathcal{R}'\}}\ \text{HT-SHARE}$$

Frame rule. Finally, the following important inference rule, known as the frame rule, makes (concurrent) separation logic a *scalable* technique for reasoning about concurrent programs: it enables local reasoning [ORY01]. More specifically, it allows localising the proof of a program C to only the resources that are needed by C, i.e., C's footprint:

$$\frac{\mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \qquad \mathsf{fv}(\mathcal{F}) \cap \mathsf{mod}(C) = \emptyset}{\mathcal{R} \vdash \{\mathcal{P} * \mathcal{F}\}\, C\, \{\mathcal{Q} * \mathcal{F}\}}\ \textsc{ht-frame}$$

The assertion $\mathcal{F}$ in HT-FRAME is often referred to as the *frame* (or the *frame axiom*) [MH81]. The frame is the part of the local state that is not accessed nor changed by C—it is in fact disjoint from the footprint of C, due to $*$. The extra premise is needed to ensure that also among local variables the frame $\mathcal{F}$ is independent of the footprint of C. Like with HT-PAR, one may get rid of this extra premise by treating local variables as resources [BCY06] and explicitly track their ownerships.

Example 2.1.1 (Application of the frame rule)**.** *The following proof tree shows the use of* HT-FRAME *to derive a proof for the program* $[x] := 3$*.*

$$\frac{\dfrac{}{\mathcal{R} \vdash \{x \overset{1}{\hookrightarrow} -\}\, [x] := 3\, \{x \overset{1}{\hookrightarrow} 3\}}\ \textsc{ht-write}}{\mathcal{R} \vdash \{x \overset{1}{\hookrightarrow} - * y \overset{\pi}{\hookrightarrow} 7\}\, [x] := 3\, \{x \overset{1}{\hookrightarrow} 3 * y \overset{\pi}{\hookrightarrow} 7\}}\ \textsc{ht-frame}$$

Also observe that the assertion $x \overset{1}{\hookrightarrow} - * y \overset{\pi}{\hookrightarrow} 7$ *disallows* x *and* y *to be aliases, since if they were aliases, the total amount of available permissions for this heap location would be (at least)* $1 + \pi$*, and would thus exceed* 1*, which is not possible.*

Example 2.1.2 (Handling concurrency and atomics)**.** *Figure 2.1 shows a proof for the specification* $\{x \overset{1}{\hookrightarrow} 1\}\, \textbf{atomic}\ [x] := 7 \parallel \textbf{atomic}\ y := [x]\, \{x \overset{1}{\hookrightarrow} - * y > 0\}$*, in which the* HT-PAR*,* HT-ATOM *and* HT-SHARE *rules are applied. Notice that atomics are required here, since otherwise there would be a data-race on the heap location* x*. The proof strategy is to share the ownership* $x \overset{1}{\hookrightarrow} 1$ *with the resource invariant, so that both threads can obtain permission to dereference* x *when needed.*

2.2 The VerCors Concurrency Verifier

This section gives an overview of concurrency verification using the VerCors verifier [BDHO17], which we use in various chapters of this thesis. VerCors uses CSL as its logical foundation, and allows verifying data-race freedom, memory safety, and functional correctness of concurrent programs written in high-level languages.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{true} \vdash \{\mathcal{R}\}\,[x] := 7\,\{\mathcal{R}\}}\,\text{HT-WRITE}}{\mathcal{R} \vdash \{\mathsf{true}\}\,C_1\,\{\mathsf{true}\}}\,\text{HT-ATOM} \qquad \dfrac{\dfrac{}{\mathsf{true} \vdash \{\mathcal{R}\}\,y := [x]\,\{\mathcal{R} * y > 0\}}\,\text{HT-ASSIGN}}{\mathcal{R} \vdash \{\mathsf{true}\}\,C_2\,\{y > 0\}}\,\text{HT-ATOM}}{\mathcal{R} \vdash \{\mathsf{true} * \mathsf{true}\}\,C_1 \parallel C_2\,\{\mathsf{true} * y > 0\}}\,\text{HT-PAR}}{\mathcal{R} \vdash \{\mathsf{true}\}\,C_1 \parallel C_2\,\{y > 0\}}\,\text{HT-CONSEQ}}{\exists v\,.\,x \overset{1}{\hookrightarrow} v * v > 0 \vdash \{\mathsf{true}\}\,C_1 \parallel C_2\,\{y > 0\}}\,\text{let } \mathcal{R} = \exists v\,.\,x \overset{1}{\hookrightarrow} v * v > 0$$

$$\dfrac{\dfrac{\mathsf{true} \vdash \{\exists v\,.\,x \overset{1}{\hookrightarrow} v * v > 0 * \mathsf{true}\}\,C_1 \parallel C_2\,\{\exists v\,.\,x \overset{1}{\hookrightarrow} v * v > 0 * y > 0\}}{\mathsf{true} \vdash \{x \overset{1}{\hookrightarrow} 1\}\,C_1 \parallel C_2\,\{x \overset{1}{\hookrightarrow} - * y > 0\}}\,\text{HT-CONSEQ}}{}\,\text{HT-SHARE}$$

Figure 2.1: Proof derivation for the specification $\{x \overset{1}{\hookrightarrow} 1\}\,C_1 \parallel C_2\,\{x \overset{1}{\hookrightarrow} - * y > 0\}$, where $C_1 = \textbf{atomic}\ [x] := 7$ and $C_2 = \textbf{atomic}\ y := [x]$. Here $\mathcal{R}$ is the resource invariant.

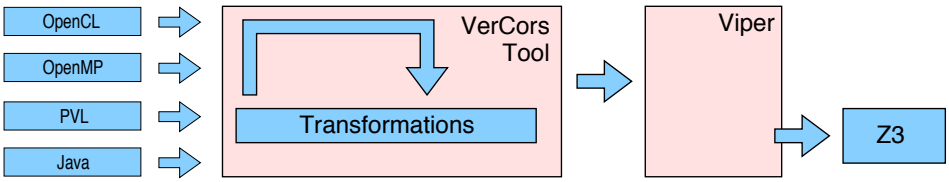


Figure 2.2: The architecture of the VerCors verifier.

Multiple widely-used languages with parallelism and concurrency features are targeted, such as Java, OpenCL, and OpenMP for C. VerCors essentially implements CSL for reasoning about various programming and concurrency models, including object-orientation (e.g., Java), homogeneous parallelism (e.g., GPGPU kernels of OpenCL), dynamically scoped parallelism (e.g., fork/join concurrency), and deterministic parallelism (e.g., OpenMP compiler directives for parallelisation).

VerCors is a tool of *static* verification in a design-by-contract fashion: it analyses the source code of an input program without actually executing it, to determine whether it satisfies a given specification. The specifications are given in the form of *program annotations*, as pre- and postconditions in the style of JML [LBR99].

Section 2.2.1 provides a quick description of the architecture of VerCors, after which Section 2.2.2 discusses several examples to illustrate its support for concurrency reasoning, and explains how this reasoning connects to CSL (i.e., §2.1.3).

2.2.1 Architecture of VerCors

The main aim of VerCors is to make existing program verification technology usable for high-level programming languages and advanced language features. This is reflected in the design of VerCors, which is implemented as a collection of compiler transformations and uses the existing Viper technology as back-end [MSS16]. Figure 2.2 gives an overview of the architecture of VerCors. Viper allows reasoning about programs with persistent mutable state, annotated with separation logic-style specifications. The compiler transformations are used to transform different high-level language/concurrency features into Viper code. The Viper technology provides two styles of reasoning: verification condition generation (via Boogie), and symbolic execution. The symbolic execution engine is the most powerful and provides support for, e.g., quantified permissions, which we heavily rely upon.

VerCors takes as input programs written in high-level programming languages, annotated with JML-style annotations, and transforms these into verification problems encoded in the Viper language. The current supported input languages are Java, PVL, OpenCL, and OpenMP for C. VerCors supports reasoning about the

main concurrency-related features of these languages. The support for OpenCL covers only the verification of kernels, including barrier synchronisation and atomic operations, but not host code (which would mostly require engineering). PVL, which abbreviates Prototypal Verification Language, is a Java-like procedural toy language used for quick prototyping of new verification features. Notably, it has support for kernels and host code. VerCors also supports a substantial subset of OpenMP, essentially characterising deterministic parallel programming.

VerCors is designed with modularity in mind. As a result, VerCors can easily be extended with new parallel or concurrent pointer languages, by providing a parser that transforms input programs and their specifications into the intermediate language of VerCors. All further program transformations are defined over this intermediate language, thereby automatically providing verification support for the features of the extended language.

2.2.2 Concurrency Reasoning with VerCors

To reason about concurrent software, VerCors uses (the principles of) intuitionistic CSL as its logical foundation. More precisely, VerCors builds on *Implicit Dynamic Frames (IDF)* [SJP09, SJP12], which is a variant of separation logic that is essentially equivalent to (intuitionistic) CSL [PS11], but that fits more naturally with object-oriented languages, and makes specification writing more convenient.

IDF uses the same principles of ownership and disjointness of shared memory as CSL, but uses a slightly different way to specify ownership. In VerCors, ownership of shared memory, e.g., a field f of an object o, is specified with a $\mathsf{Perm}(o.f, \pi)$ *permission predicate*, where π denotes the amount of ownership that is available for $o.f$, in the same style as the $\overset{\pi}{\hookrightarrow}$ predicates of CSL. These predicates may also be split and merged likewise, using the following rule.

PROC-SPLIT-MERGE
$$\mathsf{Perm}(o.f, \pi_1 + \pi_2) \dashv\vdash \mathsf{Perm}(o.f, \pi_1) * \mathsf{Perm}(o.f, \pi_2)$$

Moreover, IDF allows expressions to be heap-dependent (i.e., to refer to shared memory), but enforces that any reference to shared memory is *framed* by a permission predicate, that gives the required access rights. For example, if $o.f$ were an integer field, then $\mathsf{Perm}(o.f, \pi) * o.f > 0$ would be a properly framed assertion: the access to $o.f$ is preceded by an ownership predicate that provides the required access rights. However, the assertion $o.f > 0$ (without Perm predicate) would not verify, nor would $o.f > 0 * \mathsf{Perm}(o.f, \pi)$, since in these cases the access to $o.f$ is *not* framed (is not preceded by) permission specifications[4] to read from $o.f$.

[4]Notice that, due to these framing conditions, the separating conjunction is no longer commutative, i.e., $\mathcal{P} * \mathcal{Q}$ does not necessarily entail $\mathcal{Q} * \mathcal{P}$ in IDF, which is in contrast to CSL.

In the remainder of this thesis we sometimes switch between using the CSL and IDF formalisms. More specifically, verification examples and case studies that have been performed with VerCors (e.g., Chapters 3 and 6, and the remainder of this chapter) are presented in IDF style, whereas the more theoretical chapters (e.g., Chapters 4–5 and 7) build on CSL, and are therefore presented as such.

The remainder of this section discusses various examples of concurrency reasoning with VerCors. §2.2.2.1 discusses basic handling of ownership, after which §2.2.2.2 explains how loops are specified and verified. §2.2.2.3 and §2.2.2.4 discusses statically-scoped parallelism and locking, respectively, in the style of GPU kernels. Finally, §2.2.2.5 and §2.2.2.6 discuss dynamically-scoped (fork/join) concurrency and non-reentrant locking in VerCors, respectively.

2.2.2.1 Specifying Ownership

To better illustrate how ownership is specified in VerCors, consider the following simple example program, consisting of a class Counter with an integer field *counter*, and a procedure `incr(int n)` that increases *counter* by a specified amount n.

Example 2.2.1 (A simple counter)**.**

```
1 class Counter do
2 | int counter;
3 |
4 | void incr(int n)
5 | | counter := counter + n;
6 | end
7 end
```

Even though the above program does not have any specification annotations yet, if one would run VerCors on it, it will complain about insufficient permission for the assignment to *counter* on line 5. This is because VerCors by default checks for memory safety. Observe that *counter* is shared memory, and may potentially be accessed by other threads concurrently, and thereby cause undesirable memory access patterns like data races. To prevent these, VerCors requires to specify write permission to *counter*, as a precondition of `incr`, in the following way.

Example 2.2.2 (Specifying ownership to *this.counter*)**.**

```
1 requires Perm(this.counter, 1);
2 ensures Perm(this.counter, 1);
3 void incr(int n)
4 | counter := counter + n;
5 end
```

The Perm ownership predicates may be split and merged along their fractional permissions, and be separated by separating conjunction. To illustrate this, the annotations in the following example are equivalent to the ones of Example 2.2.2, where the purple lines $\{\cdots\}$ are internal, intermediate proof steps that are generated and proven automatically by VerCors.

Example 2.2.3 (Splitting and merging ownership)**.** *On line 5 in the following example, the* PROC-SPLIT-MERGE *rule is applied right-to-left, to merge the read ownerships to counter into write ownership, whereas on line 8 it is applied to split this write ownership back into separate read ownerships.*

```
1 requires Perm(counter, 1/2) * Perm(counter, 1/2);
2 ensures Perm(counter, 1/4) * Perm(counter, 3/4);
3 void incr(int n)
4 | {Perm(counter, 1/2) * Perm(counter, 1/2)}
5 | {Perm(counter, 1)}
6 | counter := counter + n;
7 | {Perm(counter, 1)}
8 | {Perm(counter, 1/4) * Perm(counter, 3/4)}
9 end
```

However, the total sum of fractional permissions for any shared location cannot exceed one. Any assertion that expresses ownership more than one for any shared location is equivalent to false, as is illustrated below.

Example 2.2.4 (Ownership exceeding write permission)**.** *In the following code snippet, the required permissions are first merged on line 5, which allows* false *to be deduced on line 6, using the rule of consequence,* HT-CONSEQ *(defined on page 33).*

```
1 requires Perm(counter, 1) * Perm(counter, 1/4);
2 ensures false;
3 void incr(int n)
4 | {Perm(counter, 1) * Perm(counter, 1/4)}
5 | {Perm(counter, 5/4)}
6 | {false}
7 | counter := counter + n;
8 | {false}
9 end
```

Besides heap ownership specifications, VerCors also allows writing functional specifications, for example that, after termination of `incr`, the value at *counter* has indeed been increased by n. This is shown by the following example.

Example 2.2.5 (Functional specification of `incr`).

```
1 requires Perm(counter, 1);
2 ensures Perm(counter, 1);
3 ensures counter = \old(counter) + n;
4 void incr(int n)
5 | counter := counter + n;
6 end
```

Note that, if one would remove the postcondition on line 2 of Example 2.2.5, the program would no longer verify, as in that case the postcondition on line 3 would no longer be framed by access rights to *counter*. This is expected: if no permission to *counter* were available, then $counter = \mathbf{old}(counter) + n$ cannot be guaranteed, as another thread might invalidate this postcondition right after termination of `incr`.

As discussed already, an important aspect of VerCors is that it builds on *intuitionistic* separation logic. This means that proof derivations do not have to be aware of the exact contents of the heap. In contrast, in classical separation logic the proof needs to state exactly what is in the heap. VerCors uses intuitionistic CSL, since it fits more naturally with the garbage-collected nature of modern programming languages like Java and C#, where the exact contents of the heap is never exactly known. As a consequence, VerCors may lose (forget about) resources if desired, as is indicated by the following code snippet.

Example 2.2.6 (Leaking ownership of *counter*). *The code snippet presented below shows an application of the* $*$-WEAK *rule (page 35) on line 9, after the ownership of counter has been splitted on line 8.*

```
1  requires Perm(counter, 1);
2  ensures Perm(counter, 1/2);
3  ensures counter = \old(counter) + n;
4  void incr(int n)
5  | {Perm(counter, 1)}
6  | counter := counter + n;
7  | {Perm(counter, 1) * counter = \old(counter) + n}
8  | {Perm(counter, 1/2) * Perm(counter, 1/2) * counter = \old(counter) + n}
9  | {Perm(counter, 1/2) * counter = \old(counter) + n}
10 end
```

Observe that the Perm ownership annotations (e.g., lines 1 and 2 in the above example) have a different meaning than annotations that express functional correctness (e.g., line 3). For example, the precondition on line 1 in Example 2.2.6

requires any thread invoking `incr` to give up the specified permission to *counter*, i.e., the predicate $\mathsf{Perm}(\mathit{counter}, 1)$ is *consumed*/*inhaled* in the proof system. Conversely, the ensures clause on line 3 expresses that, after termination of `incr`, the calling thread gets back (part of) the ownership to *counter*, i.e., the predicate $\mathsf{Perm}(\mathit{counter}, \frac{1}{2})$ is *produced*/*exhaled* in the proof system. The functional specification on line 3 is different, in the sense that it expresses a Boolean property that relates the output of `incr` to its input. In separation logic, such assertions are often referred to as *pure* assertions, whereas assertions that express ownership (e.g., lines 1 and 2) are referred to as *spatial* assertions. There are some logical differences between spatial and pure assertions. For example, the negation of spatial assertions is not defined, e.g., $\neg\mathsf{Perm}(\mathit{counter}, 1)$ has no meaning in VerCors.

Of course, leaking ownership as shown in Example 2.2.6 has consequences. For example, the following example program would fail to verify.

Example 2.2.7 (Insufficient ownership)**.** *Consider the following code snippet that extends Example 2.2.6. The second call to* `incr` *on line 6 will fail, as there is insufficient permission available to satisfy* `incr`*'s precondition, due to leakage.*

```
1 requires Perm(counter, 1);
2 void incr2(int n)
3 | {Perm(counter, 1)}
4 | incr(n);
5 | {Perm(counter, 1/2) * counter == \old(counter) + n}
6 | incr(n); // verification failure: precondition not satisfied!
7 end
```

Ownership leakage is sometimes used as a trick to specify that certain fields become read-only from some point on (e.g., after calling some function).

2.2.2.2 Loops and Loop Invariants

Reasoning about loops is slightly more involved than reasoning about other control-flow language constructs, since it is generally not known how many times a loop will iterate, or even whether it will terminate or not. To deal with this uncertainty it is common to maintain a *loop invariant*: an assertion that holds before entering the loop and after termination of the loop, and that can be reestablished after every iteration of the loop (see also the HT-WHILE rule on page 33).

Example 2.2.8 illustrates how loops are specified in VerCors. It shows a simple procedure, named `fill`, that "fills" a given integer array A in the sense that a value k is written to every entry of A. The example uses **context** $\mathcal{P}$ annotations, which are shorthands for **requires** $\mathcal{P}$; **ensures** $\mathcal{P}$ and are written to reduce duplication.

Example 2.2.8 (Specifying while loops)**.**

```
1  context A ≠ null;
2  context ∀* j . 0 ≤ j < A.length ⟹ Perm(A[j], 1);
3  ensures ∀j . 0 ≤ j < A.length ⟹ A[j] = k;
4  void fill(int[] A, int k)
5  |  int i := 0;
6  |
7  |  loop_invariant A ≠ null;
8  |  loop_invariant ∀* j . 0 ≤ j < A.length ⟹ Perm(A[j], 1);
9  |  loop_invariant 0 ≤ i ≤ A.length;
10 |  loop_invariant ∀j . 0 ≤ j < i ⟹ A[j] = k;
11 |  while i < A.length do
12 |  |  A[i] := k;
13 |  |  i := i + 1;
14 |  end
15 end
```

The **loop_invariant** annotations in Example 2.2.8 together form the assertion $\mathcal{P}$ in the Hoare logic rule HT-WHILE (page 33). In order to verify the property of functional correctness of `fill` on line 3, the loop body needs to have write access to $A[i]$ on every iteration i. To resolve this, lines 2 and 8 specify write access for every element in A, using a special $\forall*$ quantifier, which is known in the literature as the *iterated separating conjunction*. In particular, in the above example, the quantified expressions on lines 2 and 8 are a shorthand for expressing:

$$\mathsf{Perm}(A[0], 1) * \mathsf{Perm}(A[1], 1) * \cdots * \mathsf{Perm}(A[A.\mathit{length} - 1], 1) \tag{2.3}$$

More precisely, $\forall* x \,.\, \mathcal{P}(x)$ expresses the iteration $\mathcal{P}(v_0) * \mathcal{P}(v_1) * \cdots$ of assertions $\mathcal{P}(v_i)$ for all values v_i of the appropriate type, conjoined by $*$.

2.2.2.3 Statically-Scoped Concurrency

Notice that all iterations of the while-loop in Example 2.2.8 operate on disjoint memory, and are thus independent of one another, which gives easy opportunities for parallelism. VerCors has support to reason about statically-scoped parallel constructs, like GPGPU kernels, which spawn a number of threads that execute the same program uniformly, but with different data (which is known as the SIMD programming model). In the previous example, `fill` may be parallelised by spawning a thread for each element in A in the style of GPU kernels, so that each thread *tid* only writes to $A[\mathit{tid}]$ and performs one iteration of the loop, as follows.

The contract at lines 6–8 is called an *iteration contract*, and constitutes the pre-

Example 2.2.9 (Statically-scoped parallelism)**.**

```
1  context A ≠ null;
2  context ∀* j . 0 ≤ j < A.length ⟹ Perm(A[j], 1);
3  ensures ∀j . 0 ≤ j < A.length ⟹ A[j] = k;
4  void parfill(int[] A, int k)
5  |  par int tid := 0 to A.length
6  |    context A ≠ null;
7  |    context Perm(A[tid], 1);
8  |    ensures A[tid] = k;
9  |   do
10 |  |  A[tid] := k;
11 |  end
12 end
```

and postcondition of every parallel instance. From a separation logic point-of-view, the parallel block on lines 5–11 together with its iteration contract are an instance of a generalised version of the HT-PAR proof rule (page 37), namely:

$$\text{HT-PAR-N}\quad \frac{\forall tid \in [0,n]_{\mathbb{N}} \,.\, \vdash \{\mathcal{P}(tid)\}\, C_{tid}\, \{\mathcal{Q}(tid)\}}{\vdash \{\forall*\ tid \in [0,n]_{\mathbb{N}}\ \mathcal{P}(tid)\}\, C_0 \parallel \cdots \parallel C_n\, \{\forall*\ tid \in [0,n]_{\mathbb{N}}\ \mathcal{Q}(tid)\}}$$

The pre- and postconditions of the iteration contract on lines 6–8 correspond to the parameterised assertions $\mathcal{P}(tid)$ and $\mathcal{Q}(tid)$ in the above rule, respectively. In this case, the iteration contract specifies the ownership assigned to every thread tid, namely $\mathsf{Perm}(A[tid], 1)$, as well as their contribution, $A[tid] = k$. VerCors automatically combines these contributions (by solving the iterated separating conjunction in HT-PAR-N's conclusion) to ensure the property on line 3.

VerCors is able to reason about more elaborate applications of statically-scoped parallelism, like GPU kernels with atomics and barriers [ADBH15], essentially by using the same principles. For example, VerCors can also reason about reduction patterns in GPU kernels [BDHO17], e.g., kernels that have as input an array, and produce only a single scalar value as output (for example the sum of all elements of the input array). Furthermore, VerCors is able to reason about deterministic parallel programs in the context of OpenMP [DBH17]. Here, VerCors is able to verify whether sequential C programs annotated with OpenMP compiler directives can safely be parallelised, without changing the functional meaning of the program with respect to sequential execution. For more details we refer to [Dar18].

2.2.2.4 Statically-Scoped Locking

In the parallel program of Example 2.2.9, all threads work on disjoint parts of the heap. However, in many practical scenarios the shared-memory accesses of different threads overlap. Example 2.2.10 presents one such scenario: a concurrent program that determines the sum of all elements of a given input array A.

Example 2.2.10 (Statically-scoped locking)**.** *Verification of the following code snippet requires an application of the* HT-SHARE *rule (see page 37) on lines 8–17, as well as an application of the* HT-ATOM *rule (also on page 37) on lines 13–15.*

```
1  class Summation do
2  |  int sum;
3  |
4  |  context A ≠ null;
5  |  context ∀* j . 0 ≤ j < A.length ⟹ Perm(A[j], 1/2);
6  |  context Perm(sum, 1);
7  |  void summation(int[] A)
8  |  |  invariant Perm(sum, 1) do
9  |  |  |  par int tid := 0 to A.length
10 |  |  |  |  context A ≠ null;
11 |  |  |  |  context Perm(A[tid], 1/2);
12 |  |  |  do
13 |  |  |  |  atomic do
14 |  |  |  |  |  sum := sum + A[tid];
15 |  |  |  |  end
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

The program of Example 2.2.10 takes an integer array A as input, calculates the sum of all A's elements, and writes the result to the integer field *sum*. The calculation of the sum is done concurrently, by spawning a thread for each element of A, that contributes the value $A[tid]$ to the total sum. What makes this example difficult, is that all threads need to be able to write to *sum*. This is solved using atomics, on lines 13–15, using the language construct **atomic** C, which creates an implicit lock with a static scope that guarantees atomic execution of the program C. This in itself is not enough: the lock should protect enough resources so that each thread can do the assignment of *sum* on line 14. This is ensured by the **invariant** $\mathsf{Perm}(sum, 1)$ construct on line 8, that indicates that the lock should protect write access to *sum*, i.e., the predicate $\mathsf{Perm}(sum, 1)$.

Observe that the **invariant** $\mathcal{R}'$ **do** C **end** command is an implementation of the HT-SHARE proof rule (introduced on page 37), with $\mathcal{R}'$ the assertion that is being shared. For our example this means that, in the body program of **invariant** (lines 9–16) the write access to *sum* is no longer available, as it is protected by the resource invariant. Recall from HT-ATOM that this resource invariant can only be reacquired in the context of an **atomic** program, which we do on lines 13–15.

Note that this example only shows the distribution of ownership over threads and locks, and does not contain any functional specifications. Determining whether the correct sum has been calculated after all threads have terminated is slightly more complex: this would require to maintain some auxiliary state to explicitly track the contributions of each thread. More specifically, auxiliary state would be needed to express that, when thread *tid* has terminated, it has contributed $A[tid]$ to *sum*, and conversely, that *sum* is the summation of the contributions of all threads that have currently terminated. We do not present the auxiliary annotations required for verifying functional correctness here, but point out that a verified version can be found online at [Sup].

2.2.2.5 Dynamically-Scoped Concurrency

Besides statically-scoped (SIMD) concurrency, VerCors also supports reasoning about dynamically-scoped concurrency, e.g, fork/join threading for languages like Java. Example 2.2.11 illustrates how VerCors handles fork/join concurrency on a small example: recursively calculating the N^{th} Fibonacci number, where a new thread is forked for every recursive invocation.

The example consists of a class Fib that has two integer fields, *in* and *out*, that hold the input and output of the algorithm, respectively. The actual Fibonacci algorithm is performed by the `run` procedure. In the non-trivial case in which $2 \leq in$, `run` first instantiates two new objects of class Fib (on lines 16 and 18), which in the proof system generates new permission predicates (see lines 17 and 19). This matches with the HT-ALLOC proof rule for heap allocation given earlier, on page 36. Then, when forking a new thread for executing f_1.`run()` on line 20, all ownership required by f_1.`run`'s precondition is consumed; in this case half the ownership of $f_1.in$ and full ownership of $f_1.out$. Line 21 shows the remaining ownership of the current thread. Notice that the handling of **fork** requires an application of the frame rule, HT-FRAME, to frame-out half of the ownership of $f_1.in$; and likewise for the fork on line 22. Also notice that forking a thread generates a special Join predicate (see lines 21 and 23), which is a token that gives the right to join that thread, on lines 24 and 26. These tokens are needed to ensure soundness of the logic, to prevent generating inconsistencies by joining threads multiple times. The Join tokens are consumed when **join**ing the threads, and are exchanged for the postconditions of f_1.`run()` and f_2.`run()`, see lines 25 and 27; respectively.

Example 2.2.11 (Fork/join concurrency–calculating Fibonacci)**.** *The following code snippet shows a classical example of fork/join concurrency: calculating the Fibonacci number* $\mathsf{fib}(N)$ *of a given integer* N*. For presentational clarity the intermediate proof steps in purple only illustrate the distribution of ownership.*

```
1  class Fib do
2  | int in, out;
3  |
4  | requires 0 ≤ N;
5  | ensures Perm(in, 1) ∗ Perm(out, 1) ∗ in = N;
6  | Fib(int N)
7  | | in := N;
8  | end
9  |
10 | context Perm(in, 1/2) ∗ Perm(out, 1) ∗ 0 ≤ in;
11 | ensures out = fib(in); // the mathematical definition of Fibonacci
12 | void run()
13 | | if in < 2 then
14 | | | out := in;
15 | | else
16 | | | Fib f1 := new Fib(in − 1);
17 | | | {Perm(f1.in, 1) ∗ Perm(f1.out, 1) ∗ ···}
18 | | | Fib f2 := new Fib(in − 2);
19 | | | {Perm(f1.in, 1) ∗ Perm(f1.out, 1) ∗ Perm(f2.in, 1) ∗ Perm(f2.out, 1) ∗ ···}
20 | | | fork f1.run();
21 | | | {Join(f1) ∗ Perm(f1.in, 1/2) ∗ Perm(f2.in, 1) ∗ Perm(f2.out, 1) ∗ ···}
22 | | | fork f2.run();
23 | | | {Join(f1) ∗ Join(f2) ∗ Perm(f1.in, 1/2) ∗ Perm(f2.in, 1/2) ∗ ···}
24 | | | join f1;
25 | | | {Join(f2) ∗ Perm(f1.in, 1) ∗ Perm(f1.out, 1) ∗ Perm(f2.in, 1/2) ∗ ···}
26 | | | join f2;
27 | | | {Perm(f1.in, 1) ∗ Perm(f1.out, 1) ∗ Perm(f2.in, 1) ∗ Perm(f2.out, 1) ∗ ···}
28 | | | out := f1.out + f2.out;
29 | | end
30 | end
31 end
```

2.2.2.6 Dynamically-Scoped Locking

Finally, Example 2.2.12 gives an example of dynamically-scoped locking, i.e., locking using **lock** and **unlock** language constructs, which allow the scope of the critical region to dynamically be determined. VerCors natively supports reasoning

Example 2.2.12 (Locks with dynamic scopes)**.** *The following code snippet shows an example of dynamically-scoped non-reentrant locking, containing a procedure* `atomic_incr` *that atomically increments a shared class field val by one.*

```
1  class AtomicVal do
2  |  int val;
3  |
4  |  lock_invariant Perm(this.val, 1) * this.val > 0;
5  |
6  |  requires 0 < init;
7  |  AtomicVal(int init)
8  |  |  {Perm(val, 1)}
9  |  |  val := init;
10 |  |  {Perm(val, 1) * val = init}
11 |  end
12 |
13 |  void atomic_incr()
14 |  |  {true}
15 |  |  lock this;
16 |  |  {Perm(val, 1) * val > 0}
17 |  |  val := val + 1;
18 |  |  {Perm(val, 1) * val > 1}
19 |  |  unlock this;
20 |  |  {true}
21 |  end
22 end
```

about non-reentrant locking, meaning that acquiring the same lock twice leads to a deadlock in the program. However, reentrant locks have also been investigated. We refer to [HHH08] for the underlying theory, and [Ami18] for VerCors support.

One of the key ingredients is a *lock invariant*, declared on line 4, which corresponds to the resource invariants discussed earlier (e.g., on page 37). The lock invariant of any object o determines the resources that are protected by the lock of o. These resources must be transferred to the lock invariant at the end of the constructor; line 11 in the example. From that point on, these resources can only be obtained while holding the lock, like on line 15, and must be transferred back to the lock invariant when releasing the lock using **unlock** (see line 19).

```
1  void left()
2  |  if (l ≠ 0) then
3  |  |  l := l − 1;
4  |  |  r := r − 1;
5  |  |  a[r] := a[l];
6  |  end
7  end
8  void right()
9  |  if (r ≠ a.length − 1) then
10 |  |  a[l] := a[r];
11 |  |  l := l + 1;
12 |  |  r := r + 1;
13 |  end
14 end
15 void delete()
16 |  if (l ≠ 0) then
17 |  |  l := l − 1;
18 |  end
19 end
20 void insert(int c)
21 |  if (l = r) then
22 |  |  grow();
23 |  end
24 |  a[l] := c;
25 |  l := l + 1;
26 end
27 void grow()
28 |  int n := a.length;
29 |  int[] b := new int[n + K];
30 |  for (i := 0 to l) do
31 |  |  b[i] := a[i];
32 |  end
33 |  for (i := r to n) do
34 |  |  b[i + K] := a[i];
35 |  end
36 |  r := r + K;
37 |  a := b;
38 end
```

Figure 2.3: The basic operations on gap buffers.

2.3 Verifying a Gap Buffer Implementation

We now illustrate VerCors on a slightly bigger, yet sequential case study, namely the first challenge of the VerifyThis 2018 verification competition [HMM+19]. This challenge involves verifying four basic operations on a *gap buffer*, which is a data-structure commonly used in text editors to move the text cursor, and to add or delete characters at the cursor's current location.

A gap buffer is an integer array a, together with two indices $0 \leq l \leq r < a.length$, such that $a[l], \ldots, a[r]$ is a *gap*: a region of unused entries in a. The index l represents the current position of the cursor, and the contents of the gap buffer is represented as the section $a[0], \ldots, a[l-1], a[r], \ldots, a[a.length-1]$.

2.3.1 Problem Description

Figure 2.3 gives the implementation of four basic operations on gap buffers, namely: `left` and `right` for moving the text cursor to the left and right, respectively; `insert` for inserting a character at the position of the cursor; and `delete` for deleting the character at the cursor's position. These four operations assume the

array a to be global, as well as the indices l and r. Moreover, while inserting a character with `insert` it may happen that the gap is empty, i.e., that $l = r$. In that scenario, the procedure `grow` is called on line 22, which enlarges the array a by creating a gap of size K (which is assumed to be a positive integer).

The verification challenge is: verify in a modular way that the gap buffer behaves as intended with respect to the operations described in Figure 2.3. This intended behaviour should be specified in terms of a continuous representation of the buffers' content, for example as a sequence of characters.

2.3.2 Verification Approach

Our general approach is to represent the buffer's content as a sequence xs of integers and to verify the following:

- After calling `left`, `right` and `grow` the buffer's content is still represented by xs.
- After calling `delete` the buffer's content is represented by $xs[..l] + xs[l+1..]$, provided that a delete was possible, with l the cursor location after the call to `delete`[5].
- After calling `insert` the buffer's content is represented by $xs[..l-1] + \{c\} + xs[l-1..]$, with c the inserted character and l the cursor location after the call to `insert`.

The sequence slicing notations $xs[n..]$ and $xs[..n]$ are currently not natively supported by VerCors. Instead, we implemented these using two auxiliary operations over sequences, $\texttt{Skip}(xs, i)$ and $\texttt{Take}(xs, i)$, to skip and take the first i entries of the given sequence xs, respectively. These two operations are defined in Figure 2.4. The slicing shorthand notations are however used in the remainder of this section for presentational convenience, so that $xs[n..] \triangleq \texttt{Skip}(xs, n)$ and $xs[..n] \triangleq \texttt{Take}(xs, n)$.

2.3.3 Solution

Figure 2.5 shows the annotated version of the gap buffer implementation, with the annotated specifications displayed in blue. The presented annotations are somewhat simplified for the sake of brevity.

We define and use an auxiliary predicate $\mathsf{Represents}(xs)$ to specify the intended behaviour of the gap buffer using a sequence xs that represents the buffer's content.

[5]The + operator has been overloaded to represent sequence concatenation, and $\{c\}$ is the singleton sequence containing c as its value.

```
1  requires n ≤ |xs|;
2  ensures n < 0 ⟹ \result = xs;
3  ensures 0 ≤ n ⟹ |\result| = |xs| − n;
4  ensures 0 ≤ n ⟹ (∀i . 0 ≤ i ∧ i < |xs| − n ⟹ xs[n + i] = \result[i]);
5  seq⟨int⟩ Skip(seq⟨int⟩ xs, int n) ≜
6    0 < n ? Skip(tail(xs), n − 1) : xs;
7
8  requires n ≤ |xs|;
9  ensures n < 0 ⟹ \result = seq⟨int⟩ { };
10 ensures 0 ≤ n ⟹ |\result| = n;
11 ensures (∀i . 0 ≤ i ∧ i < n ⟹ xs[i] = \result[i]);
12 seq⟨int⟩ Take(seq⟨int⟩ xs, int n) ≜
13   0 < n ? seq⟨int⟩ { head(xs) } + Take(tail(xs), n − 1) : seq⟨int⟩ { };
```

Figure 2.4: Definitions of the `Skip` and `Take` operations, and their specifications.

Figure 2.6 shows the definition of Represents. Notably, line 2 asserts read access for l, r and a, so that the remaining definition of Represents may read from these locations. The fractions $\frac{2}{3}$ are somewhat arbitrary; the key point is that we did not express write permissions, so that we can be sure that a does not change and the contracts in Figure 2.5 may also still read from l and r (e.g., at lines 16, 17, and 28). Line 4 (Fig. 2.6) asserts write permission for every element of the array a using an iterated separating conjunction $\forall *$. Finally, lines 5–6 assert that a is properly abstracted by the integer sequence xs. More specifically, they express that the contents of xs coincides with the contents of a, minus the gap (i.e., minus the section $a[l], \ldots, a[r-1]$ of a).

The predicate Represents(xs) is used in Figure 2.5 to specify the functional behaviour of the gap buffer, and is folded and unfolded in every operation to provide access to its contents. The sequence xs is specified using a **given** annotation, stating that xs is a *ghost parameter*—an extra argument only for the sake of specification. Ghost parameters should be instantiated when calling the corresponding method, e.g., on line 38, using a **with** $\{\cdots\}$ annotation.

Also note that we slightly altered the implementations of `grow` and `insert` by making the gap size K a parameter, to simplify verification. The implementation of `grow` is omitted for brevity; the annotations mostly consist of loop invariants for the two for-loops asserting that a is correctly copied into the new array b.

The detailed, fully annotated PVL version of this example can be found in the online Git repository [Sup].

```
given seq⟨int⟩ xs;
context Perm(l, 1/3) ∗ Perm(r, 1/3);
context Represents(xs);
void left()
    if (l ≠ 0) then
        unfold Represents(xs);
        l := l − 1;
        r := r − 1;
        a[r] := a[l];
        fold Represents(xs);
    end
end

given seq⟨int⟩ xs;
context Perm(l, 1/3);
requires Represents(xs);
ensures 0 = \old(l) ⟹
    Represents(xs);
ensures 0 < \old(l) ⟹
    Represents(xs[.. l] + xs[l + 1 ..]);
void delete()
    if (l ≠ 0) then
        unfold Represents(xs);
        l := l − 1;
        fold Represents(
            xs[.. l] + xs[l + 1 ..]
        );
    end
end

given seq⟨int⟩ xs;
context Perm(l, 1/4) ∗ Perm(r, 1/3);
context K > 0;
context Represents(xs);
ensures l < r;
void grow(int K)
    // omitted for brevity
end

given seq⟨int⟩ xs;
context Perm(l, 1/3) ∗ Perm(r, 1/3);
context K > 0;
requires Represents(xs);
ensures Represents(
    xs[.. l − 1] + {c} + xs[l − 1 ..]
);
void insert(int c, int K)
    if (l = r) then
        grow(K) with { xs := xs };
        unfold Represents(xs);
    end
    else
        unfold Represents(xs);
    end
    a[l] := c;
    l := l + 1;
    fold Represents(
        xs[.. l − 1] + {c} + xs[l − 1 ..]
    );
end
```

Figure 2.5: The annotated operations of the gap buffer. The contract for `right` is the same as for `left` and is therefore omitted.

2.4 Conclusion

This chapter gives a brief background on Hoare logic, and explains how concurrent separation logic (CSL) extends on Hoare logic, to allow reasoning deductively about concurrent heap manipulating programs. All subsequent chapters in this thesis heavily build on, or extend, the CSL program logic.

```
1 resource Represents(seq⟨int⟩ xs) :=
2   | Perm(l, 2/3) ∗ Perm(r, 2/3) ∗ Perm(a, 2/3) ∗ a ≠ null ∗
3   | 0 ≤ l ≤ r < a.length ∗ |xs| = a.length − (r − l) ∗
4   | (∀∗ i . (0 ≤ i < a.length) ⟹ Perm(a[i], 1)) ∗
5   | (∀i . (0 ≤ i < l) ⟹ a[i] = xs[i]) ∗
6   | (∀i . (r ≤ i < a.length) ⟹ a[i] = xs[i − (r − l)]);
```

Figure 2.6: The definition of the Represents predicate, which relates an abstract representation of the gap buffer, xs, to the gap buffer implementation, a.

One of the central challenges of deductive concurrency verification is reasoning about thread interference, that is, reasoning about how different threads interact with each other and influence each others behaviour. CSL builds on two key mechanisms for specifying thread interference, namely: *ownership* and *disjointness*.

CSL's ownership mechanism is implemented via accessibility predicates, which are of the form $\ell \overset{\pi}{\hookrightarrow} -$ or $\mathsf{Perm}(\ell, \pi)$, that express that the current thread has access permission π to the shared heap location ℓ. Access permissions π are rational numbers in the range $(0, 1]_{\mathbb{Q}}$ that either express write access $\pi = 1$ or read access $(0 < \pi < 1)$. CSL only permits concurrent programs to access a shared heap location, if the program's specification expresses the required ownership to do so.

The mechanism of disjointness is implemented via CSL's separating conjunction connective, $\mathcal{P} * \mathcal{Q}$, which states that the ownerships expressed by the CSL assertions $\mathcal{P}$ and $\mathcal{Q}$ do not overlap (i.e., they cannot both express write ownership to the same heap location). CSL uses $*$ to enable thread-local reasoning, by enforcing that the specifications of concurrent programs are disjoint (i.e., interference-free). This implies that different threads work on disjoint parts of the heap, and are thus data-race free. Moreover, for scenarios that require thread synchronisation, e.g., two threads that access a common heap location, CSL supports basic atomics.

Furthermore, we introduced the deductive verifier VerCors, and explained how it builds on the principles of CSL to reason automatically about various sorts of concurrent programs. In particular, we illustrate how VerCors handles the principles of ownership and disjointness on various small examples, as well as a case study concerning the verification of a (sequential) gap buffer data structure.

In Chapter 3, we use VerCors in a bigger verification case study, concerning the formal verification of intricate parallel graph algorithms that are used in the context of multi-core model checking. Then, in Chapters 4–6 we extend CSL with a novel shared-memory abstraction technique, that allows specifying and verifying how the heap evolves over time. This is done by abstracting the concurrent interactions of threads with the heap as *abstract actions*, and to reason about sequences

of these actions, to reason indirectly about how the contents of the heap evolve over time. The motivation of this abstraction approach is that heap evolvement cannot so easily be specified or verified using the standard constructs of CSL, e.g., resource invariants. Finally, in Chapter 7, we investigate how this abstraction technique can be adapted for a distributed setting, by specifying the communication (send/receive) behaviour of distributed agents, instead of heap evolvement.

CHAPTER **3**

Automated Verification of Parallel Nested DFS

Abstract

Model checking algorithms are typically complex graph algorithms, whose correctness is crucial for the usability of a model checker. However, establishing the correctness of such algorithms can be challenging, and is often done manually. Mechanisation of the verification process is crucially important, because model checking algorithms are often parallelised for efficiency reasons, which makes them even more error-prone.

This chapter shows how the VerCors verifier is used to mechanically verify the parallel nested depth-first search (NDFS) algorithm of Laarman et al. [LLP+11]. We also show how having a mechanised proof supports the easy verification of various optimisations of the algorithm. As far as we are aware, this is the first automated deductive verification of a multi-core model checking algorithm.

3.1 Introduction

In Chapter 2 we introduced the VerCors concurrency verifier and demonstrated it on various verification examples. In this chapter, we use VerCors to mechanically verify parallel graph algorithms that are used in the context of model checking.

Model checking is an automated procedure to verify behavioural properties of reactive systems. To avoid a false sense of safety, it is essential that model checkers are correct themselves. However, to combat the large state space of critical industrial systems, model checkers use ever more ingenious algorithms [Sha18] and even parallel implementations [BBDL+18].

In this chapter, we focus on the mechanical verification of a multi-core algorithm to detect accepting cycles, called nested depth-first search (NDFS). This procedure

solves the model checking problem for Linear-time Temporal Logic (LTL), a widely used logic for specifying reactive systems. Developed by Laarman et al. [LLP+11] in 2011, multi-core NDFS is currently deployed in the high-performance model checker LTSmin [KLM+15]. This mechanical verification of the parallel algorithm is carried out in VerCors [BDHO17], a tool for the verification of concurrent and parallel programs based on separation logic. The verification extends a previous mechanical verification of *sequential* NDFS in Dafny [Lei10, Pol15].

This chapter shows the feasibility of mechanical program verification of parallel graph algorithms, like multi-core NDFS. The formalisation provides reusable components which can be used to verify variations of the parallel NDFS algorithm, as well as other parallel model checking algorithms.

3.1.1 Background on Model Checking

Amir Pnueli introduced the Linear-time Temporal Logic (LTL) [Pnu77] to specify properties of reactive systems. The model checking problem [Sha18] decides whether a transition system satisfies a given LTL property. The automata-based approach [VW86] reduces the model checking problem to the graph-theoretic problem of checking the reachability of an accepting cycle. Reachability of accepting cycles in a directed graph can be checked in linear-time, with the nested depth-first search algorithm (NDFS) [CVWY92, HPY96, SE05], which forms the basis of the Spin model checker.

Another line of research uses distributed and parallel computing to allocate more memory and processors to the problem [BBDL+18]. Several distributed and parallel model checking algorithms have been proposed, to allocate more memory and processors to the problem [BBDL+18]. Since linear-time algorithms for both SCC decomposition and NDFS is based on depth-first search, which is considered hard (impossible) to parallelise efficiently [Rei85]. For distributed approaches, the best strategy is to turn to Breadth-First Search algorithms [BC06], which are straightforward to parallelise, at the cost of increasing the amount of work beyond linear-time in a shared-memory setting, Swarm verification was proposed [HJG11], where each worker runs its own NDFS. However, for LTL liveness properties it was originally bounded to at most 2 processors (to handle the two nested searches). Various DFS-based multi-core algorithms for full LTL model checking have been devised [ELPP12, EPY11, LLP+11]. In this chapter, we consider the version by Laarman et al. [LLP+11], which is a parallel version of improved sequential NDFS [SE05].

Multi-core NDFS is quite subtle and hard to implement efficiently and correctly. In particular, parallel DFS does not fully respect a global depth-first ordering, since every worker maintains its own search stack, yet the correctness of NDFS depends

on the search order. Also, to realise practical speedups, the implementation avoids locking shared data structures by using atomic instructions. Finally, since graph algorithms tend to access memory in an irregular way, cache- and NUMA-aware hash-tables are required to store the set of visited states. This raises the question whether the implementation of a parallel model checker, meant to verify the correctness of safety-critical systems, is itself correct. For this reason the original paper [LLP+11] contains a detailed correctness proof, based on a number of invariants.

3.1.2 Related Work

To raise the level of confidence in model checkers, one approach is to certify each of its individual runs. Obviously, the counter-example returned by a model checker is itself a certificate, which can be easily verified independently. However, double-checking the absence of errors is harder. Namjoshi [Nam01] proposed to instrument a μ-calculus model checker, to generate a deductive proof. This proof can be checked independently, also in case the property holds. Recently, an IC3-style symbolic LTL model checker was extended with deductive proofs as well [GRT18]. However, these approaches do not prove correctness of the model checking algorithm, but only validate its outcome for each specific use.

Alternatively, one can formalise the model checking algorithm and its correctness proof in an interactive theorem prover. An early example of this approach was the verification of a model checker for the modal μ-calculus in Coq [Spr98]. A framework for verifying sequential depth-first search algorithms was developed in Isabelle [LN15, LW19], and applied to the verification of NDFS with partial-order reduction [BL18], and to a model checker for Timed Automata [WL18]. The recent formalisations of Tarjan's SCC algorithm [CCL+18] fit in the same line of research. These approaches require to model and verify the algorithm in an interactive theorem prover. The advantage is that the full power of the theorem prover can be used.

If one wishes to verify the code of the algorithm directly, yet another approach is to model the algorithm and its specification in an automated program verification tool, where the code is enriched with sufficient annotations to prove its correctness. This approach was followed for several standard sequential graph algorithms in Why3 [Why], and for sequential NDFS in Dafny [Pol15]. Wang et al. [WCMH19] verified several sequential graph manipulating (CompCert) C programs in the context of the Verified Software Toolchain [App11a], including a garbage collector for the CertiCoq project [AAM+17]. However, there is hardly any work on automated verification of parallel graph algorithms. Raad et al. [RHVG16] verified four concurrent graph algorithms in the context of CoLoSL, but the proofs have not been

automated. Sergey et al. [SNB15a] verified a concurrent spanning tree algorithm, but interactively, via a Coq embedding. Ter-Gabrielyan et al. [TGSM19] propose a separation logic-based verification technique for modularly verifying heap reachability properties, but restricts input graphs to be either DAGs or 0–1-path graphs (graphs with at most one path between every pair of nodes, modulo cycles).

To support the verification of shared-memory parallel software, program verifiers typically use concurrent separation logic. VeriFast [JSP+11] aims at sequential and multi-threaded C and Java programs. VerCors [BDHO17] verifies concurrent programs in Java and OpenCL. It applies a correctness-preserving translation into a sequential imperative language, delegating the generation of the verification conditions to Viper [MSS16] and their verification ultimately to Z3 [MB08]. Iris [BGKB19] formalises higher-order concurrent separation logic in Coq.

3.1.3 Chapter Outline

In this chapter[1], we mechanically verify the *parallel* NDFS algorithm [LLP+11] with the VerCors verifier [BDHO17]. This extends the verification of *sequential* NDFS in Dafny [Pol15]. To the best of our knowledge, we provide the first mechanical verification of a parallel graph algorithm. Section 3.2 introduces the preliminaries on the accepting cycle problem, the sequential NDFS algorithm, and the parallelised version of NDFS that we verify. The NDFS algorithm (§3.2.2) is based on various colour markings on the graph, administrating the status of the nested searches. Some of these colours are local to a single worker, while other colours are globally shared among the workers.

Section 3.3.1 presents the informal correctness proof of parallel NDFS, based on a number of global invariants on the possible colour configurations. Since workers can delegate the detection of accepting cycles to other workers, it is difficult to prove completeness of the algorithm. We contribute a new invariant (Invariant 2), which guarantees the preservation of a so-called *special path*. This allows us to circumvent using the complicated inductive argument from [LLP+11].

In Section 3.3.2, we detail how the algorithm is specified in VerCors. In particular, this requires the specification of permissions, to verify data race-free access to shared data structures. We encode the transition relation and colour maps as matrices, which greatly contributed to the feasibility of proof checking. We explain how atomic updates are specified, which was left implicit in the high-level pseudocode. Similarly, we implement asymmetric termination detection: if one worker finds a counter-example, all workers can terminate immediately; if, on the other hand, all workers have completely finished their computation, one may conclude that the model is correct.

[1] This chapter is based on the article [OHJP19].

Section 3.3.3 explains the techniques to formalise the full functional correctness proof in VerCors. In particular, this requires the distribution of permissions and invariants over threads and locks, and the introduction of auxiliary ghost state, to track the precise progress of the various search phases.

Finally, Section 3.4 demonstrates how the verification can be reused to verify optimisations to the algorithm. In particular, we check the optimisation "early cycle detection" that, for weak LTL properties, detects all cycles in the blue search. We also propose and verify a repair to the "all-red extension", inserting an extra check that was missing in [LLP+11]. This extension improves the speedup of the algorithm, by sharing more global information.

Section 3.5 concludes with a perspective on reusing our techniques for the verification of other parallel graph algorithms.

3.2 Preliminaries

Section 3.2.1 recalls the standard NDFS algorithm for finding reachable accepting cycles in automata. We verified a parallelised version of this algorithm, which is introduced in Section 3.2.2.

Before discussing the NDFS algorithms, we first recall the basic definitions of automata and accepting cycles. An *automaton* G is a quadruple $(\mathcal{S}, s_I, \mathsf{succ}, \mathcal{A})$ consisting of a finite set $\mathcal{S}$ of states, an initial state $s_I \in \mathcal{S}$, a next-state relation $\mathsf{succ} : \mathcal{S} \to 2^{\mathcal{S}}$, and a set of accepting states $\mathcal{A} \subseteq \mathcal{S}$. A *path* in G is a sequence $P = s_0, \ldots, s_{n+1}$ of $\mathcal{S}$-states so that $s_{i+1} \in \mathsf{succ}(s_i)$ for every $0 \leq i \leq n$. The notation $|P| = n + 2$ denotes the *length* of P, the indexing notation $P[i] = s_i$ the *ith state* on P, and $P[i\,..]$ the *subpath* $s_i, \ldots, s_{n+1}$. Any state s is defined to be *reachable* (from the initial state) if there exists an (s_I, s)-path. A path P is a *cycle* whenever $P[0] = P[|P| - 1]$ and $1 < |P|$. Finally, a cycle P is an *accepting cycle* if $P[i] \in \mathcal{A}$ for some $0 \leq i < |P|$.

3.2.1 Nested Depth-First Search

Figure 3.1 presents a standard, sequential implementation of NDFS, consisting of two nested DFS searches: `dfsblue` and `dfsred`. We sometimes refer to the execution of `dfsblue` as a *blue search* or an *outer search*, and an execution of `dfsred` as a *red search* or an *inner search*. The *blue search* processes successors recursively in DFS order, marking them blue when done (on line 11). The colour cyan (e.g., line 2) indicates a partially explored state, i.e., not all successors have been visited yet by the blue search. Just before backtracking from an accepting state, `dfsblue` calls the *red search* (on line 9) to report any accepting cycle. This

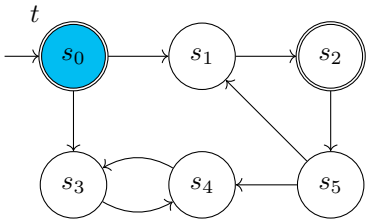
```
 1 void dfsblue(s)
 2   | s.color1 := cyan;
 3   | for t ∈ succ(s) do
 4   |   | if t.color1 = white then
 5   |   |   | dfsblue(t);
 6   |   | end
 7   | end
 8   | if s.acc then
 9   |   | dfsred(s);
10   | end
11   | s.color1 := blue;
12 end
13 void dfsred(s)
14   | s.color2 := pink;
15   | for t ∈ succ(s) do
16   |   | if t.color1 = cyan then
17   |   |   | report cycle; exit;
18   |   | end
19   |   | if t.color2 = white then
20   |   |   | dfsred(t);
21   |   | end
22   | end
23   | s.color2 := red;
24 end
```

Figure 3.1: A standard sequential implementation of nested DFS.

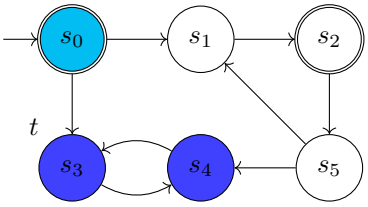
colours a state red after processing its successors recursively on line 23. The pink color denotes states that are only partially explored by `dfsred`. In the sequential algorithm, pink and red do not need to be distinguished, but having the distinction here makes the parallel version easier to explain.

Example 3.2.1 (An example run on NDFS)**.** *Figure 3.2 shows an example run of sequential NDFS on an automaton consisting of six states. The goal is to find the accepting cycle s_1, s_2, s_5 that is reachable from the initial state s_0.*

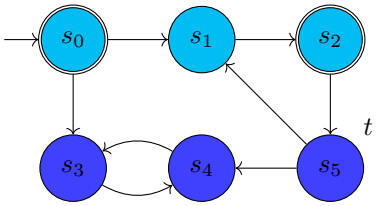
1. *The NDFS procedure starts by invoking* `dfsblue`(s_0)*, which colours s_0 cyan (Figure 3.2a) to indicate that the outer search has partially explored s_0. From this point* `dfsblue` *has the non-deterministic choice to explore either s_1 or s_3. Suppose that s_3 is chosen.*
2. *Then* `dfsblue` *fully explores s_3 and s_4 in DFS order and colours them blue (Figure 3.2b) during backtracking to administer their full exploration.*
3. *The next step is to explore s_1, s_2 and s_5 and marking them cyan while doing so, and then to colour s_5 blue (Figure 3.2c) as all its neighbours are already either cyan or blue.*
4. *During the backtrack* `dfsblue` *finds the accepting state s_2, and thus starts an inner search by invoking* `dfsred`(s_2)*, to search for a cycle that includes s_2. The inner search starts by marking s_2 pink to indicate its partial exploration by* `dfsred`*. So at this moment s_2 has two different colours, as shown in Figure 3.2d: one used by the outer search (blue) and the other (pink) by the inner search.*
5. *The inner search may proceed to explore s_5, s_4 and s_3, making them pink while doing so, and then mark s_3 and s_4 red to indicate their full exploration*



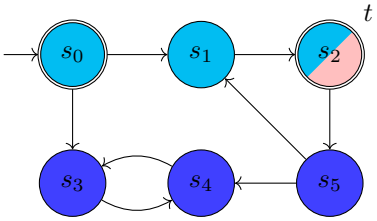
(a) The initial colouring: the outer search starts exploring from s_0 and marks it cyan.



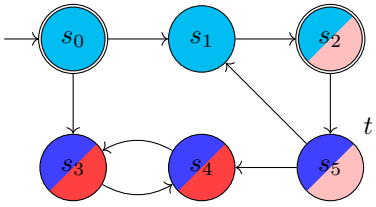
(b) s_3 and s_4 have been fully explored by the outer search.



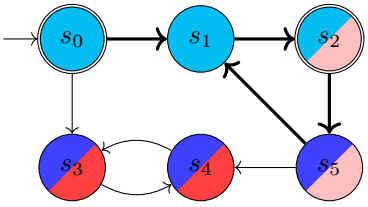
(c) The outer search partially explored s_1 and s_2, and fully explored s_5.



(d) An inner search is started from s_2 that attempts to close the accepting cycle.



(e) The inner search partially explored s_5 and fully explored s_3 and s_4.



(f) A reachable accepting cycle has been found and reported by the executing thread, as it encountered a cyan state, s_1, during its inner search.

Figure 3.2: An example run of sequential NDFS. The pointer t refers to the state that the thread is currently considering in its execution of NDFS.

by `dfsred` *(Figure 3.2e). Finally, the inner search explores* s_1 *from* s_5*, which is a cyan state.*

6. *It is a property of DFS that every cyan state has a path to* s_2*—the node from which the inner search was started (such a path resides on the recursive call stack of* `dfsblue`*). This means that by exploring* s_1 *the inner search has found a cycle that includes* s_2 *(Figure 3.2f), allowing the algorithm to terminate and yield a positive result.*

It is straightforward to see that NDFS is *sound*, i.e., it only reports true accepting cycles. To elaborate, whenever `dfsred(`s`)` executes, there is some accepting `cyan` state a on which it was originally invoked. Therefore, there exists an (a, s)-path. Moreover, from every `cyan` state there is a path to a (a standard property of `dfsblue`), thus there exists a cycle that includes a whenever one is reported on line 17. To see that NDFS is also *complete*, i.e., it will find an accepting cycle if one exists, note that `dfsred(`a`)` will indeed be started from every accepting state a. This in itself is not enough: the red search ignores states marked `red` in a previous call. It is essential that `dfsred` explores accepting states in the right order, to conclude that cycles will be found. The crucial insight is that `dfsred` only visits `cyan` and `blue` states, and that accepting states coloured `blue` cannot be part of any accepting cycle.

The correctness of NDFS has been verified with Dafny [Pol15]. We ported the correctness proof to VerCors as the basis for the verification of parallel NDFS.

3.2.2 Parallel Nested Depth-First Search

A naive strategy to parallelise NDFS is *swarming* [HJG11]: running several instances of NDFS in parallel, each working on a private set of colours. Swarmed NDFS relies on the non-determinism of `succ` for obtaining a potential speed-up, hoping that different workers explore different parts of the input graph. Swarmed NDFS tends to find accepting cycles faster, since its workers are expected to explore different parts of the input graph. However, swarming does not increase performance in the absence of accepting cycles. The correctness of swarmed NDFS with respect to sequential NDFS is almost immediate, except for termination handling: workers only share information about the exit condition. We also verified swarmed NDFS in VerCors, as a stepping stone to the verification of parallel NDFS.

Laarman et al. [LLP+11] improve on the swarming algorithm by sharing information in the backtrack phase of the red search. Figure 3.3 presents the improved algorithm. Here every line of code is supposed to be executed atomically. The entry point is `pndfs(`s_I, n`)`, which spawns n parallel instances of `dfsblue(`s_I, *tid*`)` in the fashion of swarming, so that each thread *tid* uses its own colour set. However, the `red` colourings are shared now, by which workers can guarantee that certain

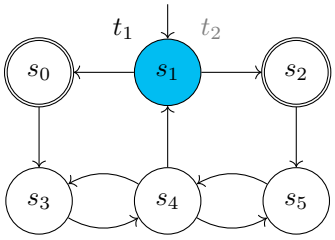
```
1  void dfsblue(s, tid)
2  |  s.color[tid] := cyan;
3  |  for t ∈ succ(s) do
4  |  |  if t.color[tid] = white ∧ ¬t.red
   |  |  then
5  |  |  |  dfsblue(t, tid);
6  |  |  end
7  |  end
8  |  if s.acc ∧ ¬s.red then
9  |  |  s.count := s.count + 1;
10 |  |  dfsred(s, tid);
11 |  end
12 |  s.color[tid] := blue;
13 end
14 void pndfs(s, nthreads)
15 |  par tid = 0 to nthreads do
16 |  |  dfsblue(s, tid);
17 |  end
18 |  report no cycle;
19 end
20 void dfsred(s, tid)
21 |  s.pink[tid] := true;
22 |  for t ∈ succ(s) do
23 |  |  if t.color[tid] = cyan then
24 |  |  |  report cycle; exit all;
25 |  |  end
26 |  |  if ¬t.pink[tid] ∧ ¬t.red then
27 |  |  |  dfsred(t, tid);
28 |  |  end
29 |  end
30 |  if s.acc then
31 |  |  s.count := s.count − 1;
32 |  |  await s.count = 0;
33 |  end
34 |  s.pink[tid] := false, s.red := true;
35 end
```

Figure 3.3: An implementation of parallel NDFS, where the red colours are shared.

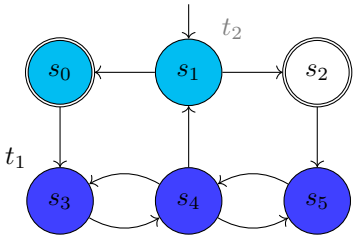
states are, or will be, sufficiently explored. So the red states can now be skipped in both the red search (line 26) *and* the blue search (line 4). PNDFS thus improves performance, since workers prune each other's search space. At the same time this significantly complicates the correctness argument, because workers might now prevent each other from finding accepting cycles. Moreover, if multiple workers initiated `dfsred` from the same accepting state, they must now finish simultaneously for the algorithm to be correct. This is ensured by the **await** s synchroniser on line 32, blocking thread execution until $s.count$, the number of workers in `dfsred`$(s, \cdot)$, reaches 0. The fields $s.count$ maintain the number of workers that are doing a red search from s.

The following example illustrates the difficulties that may arise during a run of PNDFS.

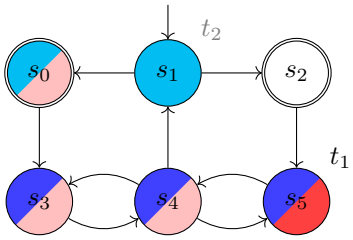
Example 3.2.2 (Difficulties of PNDFS)**.** *Figure 3.4 shows an example run of parallel NDFS performed by two threads, referred to as t_1 and t_2 in the figure. This example run shows that different workers can get in each others way, but preventing each other to find accepting cycles. This makes it significantly harder*



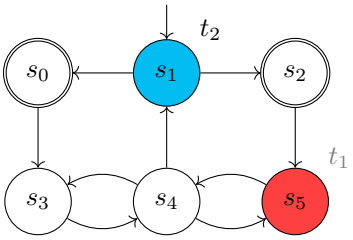
(a) Threads t_1 and t_2 both start an outer search from s_1.



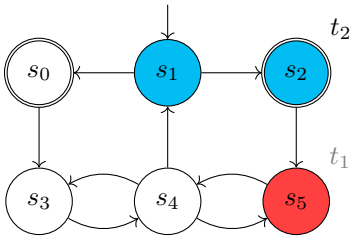
(b) Thread t_1 explored s_0, s_3, s_4, s_5 while thread t_2 remained idle/unscheduled.



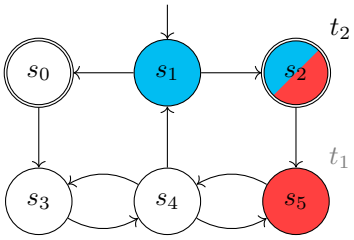
(c) Thread t_1 started an inner search from s_0 and explored s_0, s_3, s_4, s_5 while thread t_2 still has not been scheduled yet.



(d) Thread t_2 gets scheduled. These are the colours that are now visible to t_2.



(e) Thread t_2 explores s_2 while t_1 is idle.



(f) Thread t_2 starts and completes a red search from s_2.

Figure 3.4: An example run of parallel NDFS with two threads, t_1 and t_2, that shows the difficulties that may arise compared to sequential NDFS.

to establish correctness of PNDFS compared to the sequential version.

1. *The algorithm starts by spawning two threads, t_1 and t_2, that both start exploring from the initial state s_1 (see Figure 3.4a).*
2. *Suppose that t_1 gets scheduled first and chooses to explore s_0. Then the outer search of t_1 could explore s_0, s_3, s_4 and s_5 while thread t_2 remains idle—the situation of Figure 3.4b.*
3. *From this situation, t_1 may backtrack further and start an inner search from s_0, since s_0 is an accepting state. By doing so, t_1's inner search may explore s_0, s_3, s_4 and s_5 (still with t_2 inactive throughout), resulting in the situation of Figure 3.4c.*
4. *Now assume that thread t_2 finally gets scheduled. Figure 3.4d shows the colourings from t_2's perspective; it can only see its own initial cyan colouring of s_1 and the (now shared) red colouring of s_5.*
5. *Thread t_2 may choose to explore s_2 as sketched in Figure 3.4e. However, since s_5 is red, an inner search is invoked from s_2, which also immediately finishes without finding any cycle.*

Two interesting points can be highlighted in the resulting configuration of colours, as shown in Figure 3.4f. First, it shows that workers can indeed prune each others search spaces, allowing the parallel algorithm to gain potential speedup. Thread t_2 stopped exploring from s_2 because thread t_1 marked s_5 as red. Second, this pruning makes it significantly harder to see and establish that the algorithm is still correct: the accepting cycle $s_1 s_2 s_5 s_4$ is now missed and will never be found. However, we can show that if workers obstruct each other in such a way, there must always be another accepting cycle that can (and will) still be found. If the algorithm would continue from Figure 3.4f it would find the accepting cycle $s_1 s_0 s_3 s_4$.

The correctness argument of [LLP+11] relies on a complicated inductive invariant stating that not all accepting cycles can be missed due to pruning. However, this invariant is not suitable for use in an automated verifier. Section 3.3 discusses the verification of `pndfs` and provides a new invariant on the red colours that allows its correctness to be proven mechanically. We also discuss how concurrency and thread synchronisation is handled in the verification.

3.3 Automated Verification of Parallel NDFS

This section elaborates on the verification of `pndfs` in VerCors. As mentioned earlier, we actually verified three versions, viz. the sequential version from Figure 3.1, a naive parallel swarming version of it, and the full parallel algorithm from Figure 3.3 that shares the red colour. Section 3.3.1 gives a formal correctness argument for `pndfs`, that includes the new invariant on the red colours and a

proof of its correctness. Sections 3.3.2 and 3.3.3 discuss the mechanisation of this correctness proof in VerCors. The verified versions are available at [Sup].

3.3.1 Correctness of `pndfs`

The soundness proof of `pndfs` is not very different from the soundness proof of sequential NDFS: every time **report cycle** is executed, a witness cycle can be found. The main challenge lies in proving completeness: workers can *obstruct* each other's red searches and thereby prevent the detection of accepting cycles. We propose a new key invariant and completeness proof that is suitable for deductive verification. All proof steps have been mechanised in VerCors.

We start by introducing a number of low-level invariants on the local configurations of colours that can arise during a run of `pndfs`. Let $Cyan_{tid}$ be the set of cyan-coloured states $\{s \in \mathcal{S} \mid s.color[tid] = \mathsf{cyan}\}$ private to worker tid, and likewise for $White_{tid}$, $Blue_{tid}$ and $Pink_{tid}$. Moreover, let Red be the set of globally red states, and $\mathsf{succ}(X) \triangleq \bigcup_{s \in X} \mathsf{succ}(s)$ the *successor set* of a given set $X \subseteq \mathcal{S}$.

Invariant 1. `pndfs` *maintains the following global invariants during execution:*

1. $\forall tid \,.\, \mathsf{succ}(Blue_{tid} \cup Pink_{tid}) \subseteq Blue_{tid} \cup Cyan_{tid} \cup Red$
2. $\mathsf{succ}(Red) \subseteq Red \cup \bigcup_{tid}(Pink_{tid} \setminus Cyan_{tid})$
3. $\forall tid \,.\, \mathcal{A} \cap Blue_{tid} \subseteq Red$
4. $\forall tid \,.\, \mathcal{A} \cap Pink_{tid} \subseteq Cyan_{tid}$
5. $\forall tid \,.\, Pink_{tid} \subseteq Blue_{tid} \cup Cyan_{tid}$
6. $\forall tid \,.\, |\mathcal{A} \cap Pink_{tid}| \leq 1$

Proof. The proof basically checks that these invariants are preserved by each line of the program. □

Invariants 1.1–1.5 are reused from [LLP+11]. Invariant 1.6 is new and needed for the new completeness proof. Proving completeness amounts to proving that not all reachable accepting cycles can be missed due to obstruction. To help show this, we first identify a new class of paths, which we call *tid-special paths*.

Definition 3.3.1 (Special path)**.** *A path $P = s_0, \ldots, s_{n+1}$ is defined to be* tid-special *if $s_0 \in Pink_{tid}$, $s_{n+1} \in Cyan_{tid}$, and none of the states on P are red, i.e., $s_k \notin Red$ for every k such that $0 \leq k < n + 1$.*

Any path P is defined to be *special* if P is *tid*-special for some worker *tid*.

Intuitively, the existence of a *tid*-special path during execution of `pndfs` means that (i) worker *tid* is doing a red search, since it has pink states, and (ii) this worker

will eventually find an accepting cycle, unless other workers obstruct this path. Thus the above definition enables us to formally define obstruction: a worker *tid* is *obstructed* (will miss an accepting cycle) if some state on a *tid*-special path is coloured red.

Example 3.3.1 (Special paths)**.** *Figure 3.4c shows multiple t_1-special paths, including s_4, s_1 and s_0, s_3, s_4, s_1. The existence of a special path means that an accepting cycle will be found (for example s_0, s_3, s_4, s_1, s_0) unless another thread interferes.*

Our main strategy for proving completeness involves showing that every time a worker gets obstructed, a new special path can be found. A direct consequence of this is that not all accepting cycles can be missed. To help prove this, we use the following property (taken from [LLP+11], but rephrased to handle our special paths), that allows finding special paths by using the low-level colouring invariants of Invariant 1.

Lemma 3.3.1. *If Invariant 1 is satisfied, then every path $P = s_0, \ldots, s_{n+1}$ with $s_0 \in Red$ and $s_{n+1} \in \mathcal{A} \setminus Red$ contains a special subpath.*

Proof. The original handwritten proof in [LLP+11] shows that this lemma follows from Invariants 1.1–1.5 by using induction on P. □

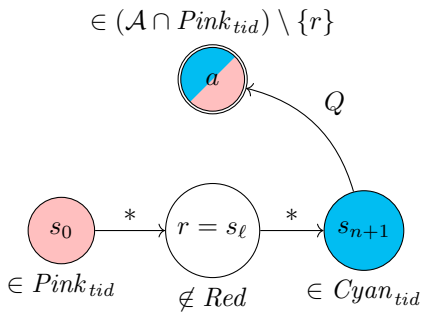
The original completeness proof of [LLP+11] performs induction on the number of obstructed accepting cycles, to show that such a cycle cannot exist as result of Theorem 3.3.1. However, such an argument cannot be encoded in an automated verifier. Instead, our new insight for a mechanised completeness proof is that, under certain colouring conditions, new special paths can always be found when workers get obstructed, as is shown by Lemma 3.3.2. In particular, `pndfs` guarantees that if there exists a special path before executing line 34, then there also exists a special path after executing it.

Lemma 3.3.2. *For any non-red state $r \in \mathcal{S} \setminus Red$ that is on a tid-special path, if:*

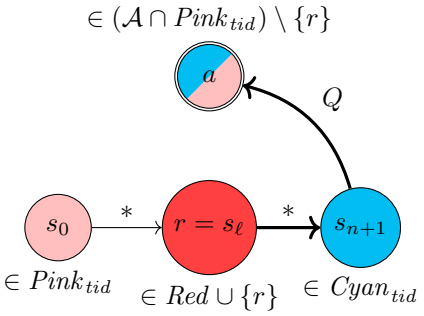
1. $r \in \mathcal{A} \implies \mathsf{succ}(r) \subseteq Red$, *and*
2. $r \in \mathcal{A} \cap Pink_{tid} \implies Pink_{tid} = \{r\}$,

then there still exists a special path after adding r to Red.

Proof. Let $P = s_0 \ldots s_{n+1}$ be a *tid*-special path and assume that r is on P, so that $r = s_\ell$, for some ℓ for which $0 \leq \ell \leq n+1$ holds. Since $Pink_{tid} \neq \emptyset$, worker *tid* is performing `dfsred` that was started from some accepting state $a \in \mathcal{A} \cap Pink_{tid}$. Then $a \neq r$, as otherwise $s_0 = r = a$ due to (*2.*), which by (*1.*) would contradict that P is special. Moreover, since $s_{n+1} \in Cyan_{tid}$ there exists a (s_{n+1}, a)-path Q



(a) Sketch of the graph that is described in the proof, with $P = s_0 \cdots r \cdots s_{n+1}$.



(b) Lemma 3.3.1 applies on the thick path when considering $Red \cup \{r\}$ as the new set of red states.

Figure 3.5: Sketch of the shape of the graph that is described in the proof of Lemma 3.3.2. Any edge $\xrightarrow{*}$ denotes a path of length ≥ 0.

(this is a standard property of `dfsblue`; the path Q must be on the recursive call stack of `dfsblue`). Figure 3.5a gives a sketch of the graph described so far. Then Lemma 3.3.1 applies on the path $s_\ell, \ldots, s_{n+1}, Q[1..]$ and gives a new special path when considering $Red \cup \{r\}$ as the new set of red states (see Figure 3.5b). □

Lemma 3.3.2 implies that every time an accepting cycle is missed due to pruning, there is always another accepting cycle that will eventually be reported. This is enough to establish completeness of `pndfs`, via the following key invariant.

Invariant 2. *The* `pndfs` *algorithm maintains the global invariant that either:*

1. *All reachable accepting cycles contain an accepting state that is not red; or*
2. *There exists a special path.*

Proof. The interesting case is showing that this invariant remains preserved after making a non-red state $s \in Pink_{tid} \setminus Red$ red (on line 34 of Figure 3.3) by some worker *tid*, which is doing a red search from some accepting state $a \in \mathcal{A} \cap Pink_{tid}$.

- Suppose $s \notin \mathcal{A}$. If s is on a special path, then Invariant 2.2 is reestablished due to Lemma 3.3.2, and otherwise the key invariant remains preserved.
- Suppose $s \in \mathcal{A}$. Then $s = a$ by Invariant 1.6. Since worker *tid* is about to finish its red exploration, we have that (*i.*) $Pink_{tid} = \{s\}$ (i.e., all other pink states have been fully explored), and consequently that (*ii.*) $\mathsf{succ}(s) \subseteq Red$. Furthermore, due to the **await** s instruction on line 32 we have that (*i.*) and (*ii.*) hold for *all* workers that are doing a red exploration that involves s. If s

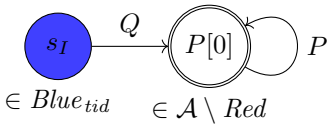


Figure 3.6: Sketch of the graph described in the proof of Theorem 3.3.3.

is on a special path, then Invariant 2.2 is reestablished due to Lemma 3.3.2. So now suppose that s is on an accepting cycle P. Without loss of generality, assume that $P[0] = s$. Then (‡) implies that $1 < |P|$ and $P[1] \in Red$. Thus Lemma 3.3.2 applies on the path $P[1..]$ to establish Invariant 2.2. □

The next theorem shows how completeness can be derived from Invariant 2. In particular, it shows that no accepting cycles can exist when all threads have terminated, in which case all the premises of the theorem are fulfilled.

Theorem 3.3.3. *If, for every worker tid, we have:*

1. $Pink_{tid} = \emptyset$, *and*
2. $Cyan_{tid} = \emptyset$, *and*
3. $s_I \in Blue_{tid}$,

then there does not exist a reachable accepting cycle.

Proof. Towards a contradiction, suppose that there exists an accepting cycle P that is reachable via an $(s_I, P[0])$-path Q. Due to the theorem's premises no special paths can exist, and therefore by Invariant 2 there is an accepting state on P that is not red. Without loss of generality, assume that (†) $P[0] \in \mathcal{A} \setminus Red$ (since otherwise Q can always be extended to meet this assumption). A sketch of the automaton structure described so far is given in Figure 3.6.

The proof proceeds by induction on Q to show that (‡) all states on Q are in $Blue_{tid} \cup Red$. The base case holds by assumption (*3.*), as $Q[0] = s_I \in Blue_{tid}$. The induction step amounts to showing that (‡) holds for the subpath $Q[..\ell+1]$, under the assumption that (‡) holds for $Q[..\ell]$. By the induction hypothesis we have that $Q[..\ell] \in Blue_{tid} \cup Red$. This gives the following two cases:

- Suppose $Q[..\ell] \in Blue_{tid}$. Then by Invariant 1.1 we have that $Q[..\ell+1] \in Blue_{tid} \cup Red$, since $Cyan_{tid} = \emptyset$ by assumption (*2.*).
- Suppose $Q[..\ell] \in Red$. Then by Invariant 1.2 we have that $Q[..\ell+1] \in Red$, since $Pink_{tid} = \emptyset$ and $Cyan_{tid} = \emptyset$ for any *tid* by assumptions (*1.*) and (*2.*).

Thus by the above induction argument it holds that $P[0] \in \mathit{Blue}_{tid} \cup \mathit{Red}$. However, since $P[0]$ is accepting, it holds that $P[0] \in \mathit{Red}$ as result of Invariant 1.3. This contradicts (†). □

We encoded and proved all the above definitions, invariants and proof steps in VerCors, which was highly non-trivial. While mechanising the proofs, many implicit proof steps had to be made explicit. Section 3.3.3 gives more details on the proof mechanisation and encoding.

3.3.2 Encoding of `pndfs` in VerCors

Graph structures are notoriously difficult to handle in separation logics, as they usually rely on pointer aliasing, which complicates ownership handling and prevents easy use of the frame rule [RHVG16]. However, since automata have a fixed and finite set of states, we can overcome this limitation by representing the input automata as an *adjacency matrix* of size $|\mathcal{S}| \times |\mathcal{S}|$. This does not impose serious restrictions: other automata encodings can be transformed at the specification level to an adjacency matrix, e.g., by using model fields in the style of JML [CLSE05, Lei98]. The suitability of adjacency matrices for deductive verification is confirmed by [Küb18].

Figure 3.7 shows the encoding of the input automaton G in VerCors, on lines 2–10. The thread-local colour sets are represented as matrices of dimension $\mathit{nthreads} \times |\mathcal{S}|$, so that each thread tid uses $\mathit{color}[\mathit{tid}][\cdot]$ (line 7) and $\mathit{pink}[\mathit{tid}][\cdot]$ (line 8) to administrate their status of exploration. The sets of red and accepting states (lines 9 and 5, respectively) are shared between threads and thus encoded as $|\mathcal{S}|$-sized Boolean *arrays* (instead of matrices). The function succ can now be defined such that $t \in \mathsf{succ}(s)$ whenever $G[s][t]$ is true, for every $0 \leq s, t < N$.

Lines 16–33 show the predicates that encode the definition of (special) paths. In particular, $\mathsf{Path}(s, t, P)$ encodes whether P is an (s, t)-path in P, i.e., a sequence of integers all within the range $[0, N)_{\mathbb{N}}$, such that $P[0] = s$, $P[|P| - 1] = t$, and there is an edge in G for every pair of adjacent elements in P. The predicate $\mathsf{ExPath}(s, t, n)$ encodes the existence of an (s, t)-path in G of length at least n. The latter argument, n, is used to establish the existence of cycles: given any state s, the predicate $\mathsf{ExPath}(s, s, 2)$ holds if there exists a cycle that includes s. The predicate $\mathsf{SpecialPath}(P, \mathit{tid})$ on line 28 expresses the conditions needed for P to be a tid-special path: it must be a path free of red states such that $P[0]$ is pink and $P[|P| - 1]$ is cyan. Finally, $\mathsf{ExSpecialPath}(\mathit{tid})$ encodes the existence of a tid-special path (on line 32). This predicate is needed to encode Invariant 2.2.

This encoding of automata, together with the encoding of the definition of paths is sufficient to express the main correctness property that is proven by VerCors.

```
1  /* Representation of the input automaton. */
2  int N; // the number of automata states (equal to |S|).
3  int nthreads; // the total number of participating workers.
4  boolean[N][N] G; // adjacency matrix representation of the input automaton.
5  boolean[N] acc; // encoding of the set of accepting states.
6  enum Color {white, cyan, blue}; // different types of colours for the outer search.
7  Color[nthreads][N] color; // the colour sets for dfsblue; one for each thread.
8  boolean[nthreads][N] pink; // the pink colour sets for dfsred; one per thread.
9  boolean[N] red; // the global set of red colourings.
10 boolean abort; // the global termination flag.
11
12 /* The resource invariant, which describes all ownerships protected by the lock. */
13 resource_invariant := ···; // further discussed in §3.3.3.1, on page 77.
14
15 /* The encoding of (s, t)-paths in G. */
16 boolean Path(int s, int t, seq⟨int⟩ P) :=
17   0 ≤ s, t < N ∧ 0 < |P| ∧ P[0] = s ∧ P[|P| − 1] = t ∧
18     (∀i . 0 ≤ i < |P| ⟹ 0 ≤ P[i] < N) ∧ (∀i . 0 ≤ i < |P| − 1 ⟹
19       G[P[i]][P[i + 1]]);
20
21 /* A predicate that encodes whether P is a path in G. */
22 boolean Path(seq⟨int⟩ P) := 0 < |P| ∧ Path(P[0], P[|P| − 1], P);
23
24 /* A predicate that encodes the existence of an (s, t)-path of length at least n. */
25 boolean ExPath(int s, int t, int n) := ∃P . n ≤ P ∧ Path(s, t, P);
26
27 /* The encoding of tid-special paths in G. */
28 boolean SpecialPath(seq⟨int⟩ P, int tid) := Path(P) ∧ pink[tid][P[0]] ∧
29   color[tid][P[|P| − 1]] = cyan ∧ (∀i . 0 ≤ i < |P| ⟹ ¬red[P[i]]);
30
31 /* A predicate that encodes the existence of tid-special paths. */
32 boolean ExSpecialPath(int tid) :=
33   ∃P . (1 < |P| ∧ Path(P)) ⟹ SpecialPath(P, tid);
34
35 /* An excerpt of the top-level contract of pndfs. */
36 context ···; // further discussed in §3.3.3.3, on page 81.
37 ensures \result ⟹
38   (∃a . 0 ≤ a < N ∧ acc[a] ∧ ExPath(s_I, a, 1) ∧ ExPath(a, a, 2)); // soundness.
39 ensures (∃a . 0 ≤ a < N ∧ acc[a] ∧ ExPath(s_I, a, 1) ∧ ExPath(a, a, 2)) ⟹
40   \result; // completeness.
41 boolean pndfs(int s_I)
42 | ··· // further discussed in §3.3.3.3, on page 81.
43 end
```

Figure 3.7: Automata representation and an excerpt of `pndfs`'s top-level contract.

Notably, lines 37–38 expresses *soundness*: if `pndfs` returns true, then there exists an accepting cycle. *Completeness* is expressed on lines 39–40: if there exists an accepting cycle, then `pndfs` yields a positive result.

3.3.2.1 Atomic Operations

The handwritten correctness argument of [LLP+11] for Figure 3.3 assumes that all program lines are executed atomically. This is reflected in the VerCors encoding: all updates to shared memory are made within atomic operations, which specification-wise all give access to the same shared resources. For example, the assignment $s.pink[tid] := \mathsf{true}$ on line 21 (Figure 3.3) is implemented as the following atomic set operation:

$$\textbf{atomic } \{\ pink[tid][s] := \mathsf{true};\ \}.$$

Note that this implementation of atomic operations can easily be lifted to an actual fine-grained atomic instruction in, for example, Java and C [ABH14].

On the specification level, the atomic sub-program receives all the missing access rights required for the assignment, which are otherwise protected by the resource invariant declared on line 13 (Figure 3.7). The exact definition of this resource invariant is deferred to Section 3.3.3.1. Furthermore, the **await** instruction on line 32 (Figure 3.3) is implemented as a busy while-loop that only stops when $s.count = 0$, which is checked atomically in every iteration.

3.3.2.2 Termination Handling

The pseudocode in Figure 3.3 uses an **exit all** command to terminate all threads when an accepting cycle has been found, but the mechanism was left implicit. Our formalisation in VerCors makes the termination mechanism explicit: it consists primarily of a global *abort* flag (that is declared on line 10, Figure 3.7). All workers regularly poll this flag to determine whether they continue or not. The *abort* flag is set to `true` by the main thread (the one that started `pndfs` and spawned all worker threads on line 15 of Figure 3.3) as soon as one of the workers returns with an accepting cycle.

3.3.3 Verification of `pndfs` in VerCors

One of the major challenges of concurrency verification is finding a proper distribution of shared-memory ownership that allows proving data-race freedom and verifying any functional properties of interest. This section starts by discussing how we distribute the ownership of the input automaton over the different threads and the resource invariant, in such a way that Invariant 1 and Invariant 2 can be

encoded. To prove the preservation of these invariants after every computation step, auxiliary bookkeeping is needed on the specification level. For example, to mechanise the proof of Invariant 2 (and also of Lemma 3.3.2) we need to make explicit that all workers *tid* with $Pink_{tid} \neq \emptyset$ are doing a red search that was started from some root state $a \in \mathcal{A} \cap Pink_{tid}$. This auxiliary bookkeeping is maintained in the resource invariant, via *auxiliary ghost state*, which is explained later. Finally, we give the fully annotated version of `pndfs` and explain how completeness is proven from Invariant 2, by applying the VerCors encoding of Theorem 3.3.3.

3.3.3.1 Distribution of Ownership

We start by explaining how the ownership of the automaton encoding (lines 2–10 in Figure 3.7) is distributed among the workers and the resource invariant. First recall that Invariants 1 and 2 both express *global properties* that span over (i) the shared *red* colourings, as well as over (ii) the local configurations $color[tid]$ and $pink[tid]$ of every worker *tid*. To define the ownership distribution for (i), observe that the only way to distribute the access rights to *red* to enable all threads to regain write access when needed, is to let the resource invariant have full ownership of *red*. This means that the resource invariant fully captures the properties about red states expressed in Invariants 1 and 2. However, to be able to specify that, the resource invariant also requires partial ownership of the thread-local colourings.

Figure 3.8 presents the full resource invariant, that includes: access rights to both global and thread-local colourings on lines 3–5; the encoding of Invariant 1 on lines 14–22 and 28; as well as the encoding of Invariant 2 on lines 38–40. In addition, the resource invariant holds partial ownership of the *abort* flag on line 11, to ensure that global termination can only be announced when an accepting cycle is found (by the condition on line 10).

The `dfsred_status()` resource on lines 25–35 is used by the resource invariant to maintain *auxiliary ghost state* (displayed in blue) on the progress of the inner search (`dfsred`) of all workers. More specifically, it maintains ghost fields to administer which workers are currently performing an inner search (using *exploringred*); from which node the inner search was started (using *redroot*); and whether the worker is currently waiting by executing an **await** instruction (using *waiting*). This auxiliary information is needed for proving and using Lemma 3.3.2 as well as Invariant 2. This use of auxiliary ghost state is further explained in §3.3.3.2.

Observe that the resource invariant holds a lot of quantified information. As a result, we experienced that proving the reestablishment of the resource invariant after finishing **atomic**s is expensive performance-wise. To make verification more efficient, we extracted all atomic operations (e.g., colour updates) into separate methods and prove their contracts in a function-modular way. This improves

```
1  /* Full definition of the resource invariant (declared earlier, in Fig.3.7; line 13). */
2  resource_invariant :=
3  | Perm(N) * Perm(nthreads) * Perm(G) * Perm(acc) *
4  | (∀tid, s . Perm(color[tid][s], 1/2) * Perm(pink[tid][s], 1/2)) *
5  | (∀s . Perm(red[s], 1)) *
6  | termination() * colourings() * dfsred_status() * keyinvariant();
7
8  /* Resources for termination handling, for proving soundness of pndfs. */
9  resource termination() :=
10 | Perm(abort, 1/2) *
11 | (abort ⟹ ∃s . acc[s] ∧ ExPath(s_I, s, 1) ∧ ExPath(s, s, 2));
12
13 /* The encoding of the low-level colouring invariants 1.1–1.5 (page 70). */
14 resource colourings() :=
15 | (∀tid, s . (color[tid][s] = blue ∨ pink[tid][s]) ⟹ ∀s' ∈ succ(s) .
16 |    color[tid][s'] = blue ∨ color[tid][s'] = cyan ∨ red[s']) * // Inv. 1.1.
17 | (∀s . red[s] ⟹ ∀s' ∈ succ(s) .
18 |    red[s'] ∨ ∃tid . pink[tid][s'] ∧ color[tid][s'] ≠ cyan) * // Inv. 1.2.
19 | (∀tid, s . (acc[s] ∧ color[tid][s] = blue) ⟹ red[s]) * // Inv. 1.3.
20 | (∀tid, s . (acc[s] ∧ pink[tid][s]) ⟹ color[tid][s] = cyan) * // Inv. 1.4.
21 | (∀tid, s . pink[tid][s] ⟹
22 |    (color[tid][s] = cyan ∨ color[tid][s] = blue)); // Inv. 1.5.
23
24 /* Auxiliary state for proving Lemma 3.3.2 and the preservation of Inv. 2. */
25 resource dfsred_status() :=
26 | ∀tid . (Perm(exploringred[tid], 1/2) * Perm(redroot[tid], 1/2) * Perm(waiting[tid], 1/2) *
27 | ∀s . (pink[tid][s] ⟹
28 |    (exploringred[tid] ∧ (acc[s] ⟹ s = redroot[tid])) * // Inv. 1.6.
29 |    exploringred[tid] ⟹ (acc[redroot[tid]] ∧
30 |       (∀s . pink[tid][s] ⟹ ExPath(redroot[tid], s, 1)) ∧
31 |       (∀s . color[tid][s] = cyan ⟹ ExPath(s, redroot[tid], 1)) ∧
32 |       (¬waiting[tid] ⟹ ¬red[redroot[tid]]) ∧
33 |       (waiting[tid] ⟹ ∀s . pink[tid][s] ⟺ s = redroot[tid])
34 |    )
35 | ));
36
37 /* The encoding of Invariant 2 (page 72), from which completeness follows. */
38 resource keyinvariant() :=
39 | (∀s . (acc[s] ∧ ExPath(s_I, s, 1) ∧ ExPath(s, s, 2)) ⟹ ¬red[s]) ∨ // Inv. 2.1.
40 | (∃tid . ExSpecialPath(tid)); // Inv. 2.2.
```

Figure 3.8: The global resource invariant. For brevity we omitted several bound checks.

```
1  context Perm(N) ∗ Perm(nthreads) ∗ Perm(G) ∗ Perm(acc);
2  context 0 ≤ s < N;
3  context 0 ≤ tid < nthreads;
4  context ∀t . (0 ≤ t < N) ⟹ Perm(color[tid][t], 1/2) ∗ Perm(pink[tid][t], 1/2);
5  requires color[tid][s] = white;
6  requires ∀t . (0 ≤ t < N ∧ color[tid][t] = cyan) ⟹ ExPath(t, s, 1);
7  ensures \result ⟹ ∃a . 0 ≤ a < N ∧ acc[a] ∧ ExPath(s_I, a, 1) ∧ ExPath(a, a, 2);
8  ensures ¬\result ⟹ ∀t . (color[tid][t] = cyan ⟺ \old(color[tid][t]) = cyan);
9  ensures ¬\result ⟹ (pink[tid] = \old(pink[tid]) ∧ color[tid][s] = blue);
10 boolean dfsblue(int s, int tid)
11   | ···
12 end
```

Figure 3.9: The ownership specification in the contract `dfsblue` for thread *tid*.

performance, as it cuts the problem of verifying `dfsred` and `dfsblue` into smaller bits that are individually more manageable for the underlying SMT solver.

High-level contract of `pndfs`. Figure 3.9 presents an excerpt of the contract of `dfsblue`, showing the ownership pattern of all threads. Annotations of the form **context** P abbreviate **requires** P; **ensures** P, and are used to reduce duplication.

Notably, every thread *tid* receives the *remaining ownership* of $color[tid]$ and $pink[tid]$ on line 4. With remaining ownership we mean the access rights to *color* and *pink* that are not held by the resource invariant. The implication of this distribution of access rights, is that threads can always read from their thread-local colour fields, and may write to them while doing so atomically. This distribution of ownership matches with the encoding of atomic operations discussed earlier.

Line 7 expresses soundness of `dfsblue`, captured in the resource invariant (line 10 of Figure 3.8) when global termination has been announced. This allows soundness of `pndfs` to be deduced from the resource invariant, after all threads have terminated as result of the detection of an accepting cycle.

3.3.3.2 Auxiliary Ghost State

As mentioned earlier, to prove that `pndfs` also *preserves* the (encodings of) Invariants 1 and 2 after every computation step, additional ghost state needs to be maintained. In particular, we need to make explicit that every worker *tid* with $Pink_{tid} \neq \emptyset$ is doing a `dfsred` search that was started from some root state $a \in \mathcal{A} \cap Pink_{tid}$. In addition, the proof of Lemma 3.3.2 needs that there exists an (s, a)-path for every $s \in Cyan_{tid}$. To prove the preservation of Invariant 2 we

additionally need that, if worker *tid* is not yet executing the **await** instruction it holds that $a \notin \mathit{Red}$, or otherwise it holds that $\mathit{Pink}_{tid} = \{a\}$.

This extra information is encoded in the resource invariant on lines 25–35 (Figure 3.8 on page 78), via three *ghost arrays*, named *exploringred*, *redroot*, and *waiting*. Firstly, the *exploringred* ghost array is used to administer which workers are doing a red search. For verification purposes we also added ghost code to the program, to set $\mathit{exploringred}[\mathit{tid}]$ to `true` whenever `dfsred(`a, tid`)` is invoked by worker *tid* from a blue search, and back to `false` whenever `dfsred(`a, tid`)` returns. Secondly, *redroot* stores the root state on which `dfsred` was invoked. Finally, *waiting* administrates which workers are executing an **await** instruction.

These three ghost arrays together are closely related to the $s.\mathit{count}$ fields in the program of Figure 3.3, via the following invariant:

$$\forall s \,.\, s.\mathit{count} = \left|\left\{\ \mathit{tid} \mid \mathit{exploringred}[\mathit{tid}] \wedge \mathit{redroot}[\mathit{tid}] = s \wedge \neg \mathit{waiting}[\mathit{tid}]\ \right\}\right|$$

Establishing that `pndfs` adheres to Invariants 1 and 2 was highly non-trivial and required multiple complex auxiliary lemmas to be encoded and proven. These are encoded in VerCors as *ghost methods*: side-effect-free helper methods on which the lemma is encoded in the method's contract. Induction proofs, for example, are encoded using either resource invariants or recursion. The following example illustrates such an induction proof, and is one of the auxiliary lemmas needed by VerCors to prove correctness of `pndfs`.

Example 3.3.2 (Encoding induction proofs as functions in VerCors)**.** *The following function encodes a simple auxiliary lemma stating that, if P is an (s,t)-path such that s is blue, and if all successors of blue states are also blue, then it must be that all states on P are blue. This is proven by induction on the length of P.*

```
1 requires Path(s, t, P);
2 requires color[tid][s] = blue;
3 requires ∀v . color[tid][v] = blue ⟹ ∀v' ∈ succ(v) . color[tid][v'] = blue;
4 ensures \result ∧ ∀i . (0 ≤ i < |P| ⟹ color[tid][P[i]] = blue);
5 pure boolean lemma_reachblue(int tid, int s, int t, seq⟨int⟩ P) :=
6     1 < |P| ⟹ lemma_reachblue(tid, P[1], t, P[1..]);
```

The VerCors encoding of this lemma actually includes bounds checks as well as Perm *annotations for the shared arrays. These are omitted for ease of presentation.*

VerCors can automatically prove this lemma: the base case of the induction proof amounts to $1 \geq |P|$, in which case $1 = |P|$ due to the Path *predicate on line 1, and so it trivially follows that $s = t$ and thus that P is entirely blue.*

The induction case is encoded via the recursive call to `lemma_reachblue` *on line 6, by which the contract of* `lemma_reachblue` *is used as the induction hypothesis, with respect to the subpath* $P[1..]$. *In particular, since* $s = P[0]$ *is blue (line 2), by line 3 it must be that* $P[1]$ *is also blue, and therewith all preconditions for the recursive call to* `lemma_reachblue` *are satisfied. Moreover, due to the induction hypothesis (i.e., the function contract), the path* $P[1..]$ *is entirely blue. VerCors can therefore automatically conclude that* P *must also be entirely blue.*

Application of a lemma in VerCors then amounts to a function call on the specification level. For example, the function invocation `lemma_reachblue`(tid, s, t, P) gives the proof that P is entirely blue, via its postcondition, provided that all preconditions of `lemma_reachblue` can be proven at the point of invocation.

The proofs in Section 3.3.1 are all encoded and applied in this way.

3.3.3.3 Correctness of `pndfs`

Figure 3.10 gives the annotated version of `pndfs`[2] that extends the excerpt given earlier, in Figure 3.7 (on page 75). The main thread requires partial ownership of all thread-local colour fields on line 2, and distributes these over the appropriate threads on line 12. The contract associated to the parallel block (lines 11–17) is called an *iteration contract* and assigns pre- and postconditions to every parallel instance. For more details on iteration contracts we refer to [BDH15]. Most importantly, the iteration contract of each thread holds enough resources to satisfy all preconditions for the invocation of `dfsblue`, on line 19.

Now suppose that all threads have terminated and *abort* has been set to `true`. In that case, the resource invariant states that an accepting cycle has been found. This information can be retrieved by briefly obtaining the resource invariant in *ghost code* on line 24, which directly allows deducing soundness (see line 7). This retrieval could also have been implemented on the level of program code, e.g., by using an atomic set operation, as explained in [ABH14]. Note that the retrieved information is not lost upon releasing the resource invariant, as it is a pure Boolean property and thus duplicable.

Conversely, when *abort* is still false when all workers have terminated, it holds that $Pink_{tid} = \emptyset$ and $Cyan_{tid} = \emptyset$ for every worker *tid* (line 16). Furthermore, since all workers started their blue exploration from s_I, we also have that $s_I \in Blue_{tid}$ (line 17). By briefly obtaining the resource invariant on line 24, the main thread gets access to the encoding of Invariant 2. As a result, the executing thread is

[2]Note that every thread reads *abort* in their contract on lines 16–17, even though they do not have the required access rights to do so. This is resolved by adding some auxiliary ghost state, but this is omitted for presentational clarity.

```
1  context Perm(N) ∗ Perm(nthreads) ∗ Perm(G) ∗ Perm(acc) ∗ Perm(abort, 1/2);
2  context ∀tid, s . Perm(color[tid][s], 1/2) ∗ Perm(pink[tid][s], 1/2);
3  context ∀tid . Perm(exploringred[tid], 1/2) ∗ Perm(redroot[tid], 1/2);
4  context ∀tid . Perm(waiting[tid], 1/2);
5  context 0 ≤ s_I < N;
6  requires ∀tid, s . ¬exploringred[tid] ∧ color[tid][s] = white ∧ ¬pink[tid][s];
7  ensures \result ⟹ (∃a . acc[a] ∧ ExPath(s_I, a, 1) ∧ ExPath(a, a, 2));
8  ensures (∃a . acc[a] ∧ ExPath(s_I, a, 1) ∧ ExPath(a, a, 2)) ⟹ \result;
9  boolean pndfs(s_I)
10 | par tid = 0 to nthreads
11 |   context Perm(N) ∗ Perm(nthreads) ∗ Perm(G) ∗ Perm(acc);
12 |   context ∀s . Perm(color[tid][s], 1/2) ∗ Perm(pink[tid][s], 1/2);
13 |   context Perm(exploringred[tid], 1/2);
14 |   context Perm(redroot[tid], 1/2) ∗ Perm(waiting[tid], 1/2);
15 |   requires ¬exploringred[tid] ∧ ∀s . color[tid][s] = white ∧ ¬pink[tid][s];
16 |   ensures ¬abort ⟹ ∀s . color[tid][s] ≠ cyan ∧ ¬pink[tid][s];
17 |   ensures ¬abort ⟹ color[tid][s_I] = blue;
18 |  do
19 |  | boolean found := dfsblue(s_I, tid);
20 |  | if found then
21 |  |  | atomic { abort := true; } // initiate global termination.
22 |  | end
23 |  // apply the encoding of Theorem 3.3.3 as an atomic ghost function call.
24 |  atomic { if ¬abort then theorem_completeness() }
25 |  return abort;
26 end
```

Figure 3.10: The annotated version of parallel NDFS, that extends the excerpt given in Figure 3.7.

able to prove all the premises of Theorem 3.3.3. Thus the ghost method encoding of Theorem 3.3.3 can be applied on line 24, from which completeness of `pndfs` (line 8) is derived.

The encoding of parallel NDFS in VerCors comprises roughly 2500 lines of code, which includes the mechanisation of all proof steps described in Section 3.3.1. The verification time is about 140 seconds, measured on a Macbook with an Intel Core i5 CPU with 2,9 GHz, and 8Gb internal memory.

3.4 Optimisations

One major benefit of mechanically verified code is that optimisations can be applied with full confidence. Without verification, changes to critical code are often

```
1  void dfsblue(s, tid)
2    s.color[tid] := cyan;
3    for t ∈ succ(s) do
4      if t.color[tid] = cyan then
5        if s.acc ∨ t.acc then
6          report cycle; exit all;
7        end
8      end
9      if t.color[tid] = white then
10       if ¬t.red then
11         dfsblue(t, tid);
12       end
13     end
14   end
15   if s.acc ∧ ¬s.red then
16     s.count := s.count + 1;
17     dfsred(s, tid);
18   end
19   s.color[tid] := blue;
20 end
```

(a) The "early cycle detection" extension.

```
1  void dfsblue(s, tid)
2    allred := true;
3    s.color[tid] := cyan;
4    for t ∈ succ(s) do
5      if t.color[tid] = white then
6        if ¬t.red then
7          dfsblue(t, tid);
8        end
9      end
10     if ¬t.red then
11       allred := false;
12     end
13   end
14   if allred then
15     await s.count = 0;
16     s.red := true;
17   end
18   else if s.acc ∧ ¬s.red then
19     s.count := s.count + 1;
20     dfsred(s, tid);
21   end
22   s.color[tid] := blue;
23 end
```

(b) The "all-red" extension.

Figure 3.11: Two extensions to `dfsblue` that improve work sharing, and thereby performance of the algorithm. The code extensions are highlighted grey.

avoided, to ensure that no errors are introduced. A verified algorithm allows optimisations to be applied easily, as these often do not change the outer contract, at most requiring only minor adaptions to the invariants. We illustrate this with two optimisations, for which [LLP+11] experimentally demonstrated improved speedup.

The "early cycle detection" extension checks already in the blue search if an accepting cycle is closed, cf. lines 4–6 in Figure 3.11a. It is known that for weak LTL properties, all accepting cycles will now be found in the blue search. When checking weak LTL properties, the automata are always such that t is accepting if there is any accepting node on the path from s to t, so this special case is not uncommon. If all nodes in a strongly connected component are accepting (which can be achieved for the large class of weak LTL properties), this optimisation will

detect all accepting cycles in the blue search. To show that this optimisation indeed preserves the invariant, we simply inserted these 3 lines in the VerCors specification. The proof introduces a case distinction on whether s or t is accepting and constructs a witness path. This adds another 10 lines: two for the case distinction and four in each branch to show that a witness accepting cycle exists. Collectively, these extra 13 lines constitute indeed very little effort to prove this particular optimisation correct.

The second optimisation, named "all-red", checks if all successors of s became red (lines 2 and 7 in Figure 3.11b). If so, we can mark $s.red$ (lines 8–10). This optimisation is important, since it allows spreading the global red colour even in portions of the graph that are not under an accepting state, thereby allowing more pruning. However, this optimisation only preserves the invariants if we wait until $s.count = 0$ (on line 9). This test was erroneously omitted in [LLP+11].[3] Fortunately, the version in Figure 3.11b is correct, which has now been checked in VerCors in a straightforward manner.

3.5 Conclusion

This chapter presents the first *automated* deductive verification of a parallel graph algorithm, nested depth-first search, which is a major contribution to the field of computer-aided verification. We verified both soundness and completeness of parallel NDFS using the automated verifier VerCors. The correctness proof is inspired by an earlier handwritten proof [LLP+11], but the completeness argument has been rephrased in order to mechanise the proof. In addition, many details that were left implicit in the handwritten proof are made explicit. Furthermore, we show that this mechanisation is helpful in quickly discovering whether optimisations of the algorithm preserve its correctness. Many of the presented verification techniques, e.g., the use of separate contracts for single statements, the way we handle termination of the algorithm, and the construction of explicit witnesses through ghost-variables, will be useful for the verification of other similar algorithms. Moreover, our encoding of parallel NDFS closely resembles the implementation of such an algorithm in main-stream programming languages like C++ and Java. The way we handle graphs provides a first step towards a library for the verification of graph-based model checking algorithms.

We noticed that, since VerCors is not interactive like, e.g., Coq or Isabelle, there is less control on how quantifiers are instantiated, which makes mechanising the proof steps described in Section 3.3.1 more work-intensive. An interactive mode

[3] In 2012, Wan Fokkink and his students already found that the all-red extension in [LLP+11] required an extra check "**await** $s.count = 0$", and wondered whether "**await** $s.count \leq 1$" would be sufficient. Independently, Akos Hajdu reported this omission in 2015.

would be a useful addition to VerCors for doing proofs that are not immediately related to program execution. On the other hand, the automation that VerCors provides, together with its ability to directly analyse program code, were crucial in tackling a concurrency verification problem of this scale.

Finally, notice that, for this `pndfs` verification case study, we managed to encode all the properties required for proving correctness into a single global invariant (presented in Figure 3.8 on page 78). However, it is in general not always possible nor convenient during verification to encode the property of interest into a global, static invariant. For example, protocol-like properties of control-flow (e.g., a lock always first needs to be unlocked before it can be acquired) are not conveniently expressed into a static, first-order logic invariant. Chapter 4 introduces an abstraction technique that allows specifying such protocol-like properties about heap evolvement conveniently, in a shared-memory CSL setting. This abstraction approach is demonstrated on various verification examples, before Chapter 5 gives a full machine-checked formalisation and soundness proof of the approach.

3.5.1 Future Work

There are many possibilities to extend the line of research on the verification of parallel model checking algorithms initiated in this thesis. First, one may consider to extend the scope of this verification closer towards the actual efficient C-implementation in LTSmin. This would involve the verification of the underlying concurrent hash-table to store visited states (a simplified version of which has been verified before with VerCors [ABH14]), the encoding of the colours as "bits" in the hash table buckets, and the use of CAS to manipulate these bits.

One might consider alternative parallel NDFS versions, notably [EPY11], which shares the blue colour, invoking a repair procedure when the depth-first order is violated. Both algorithms have been reconciled in [ELPP12], sharing both blue and red. This work could be extended to a wealth of other optimisations (like partial-order reduction) or other parallel model checking algorithms, e.g., based on parallel SCC algorithms [RDKP17, BLP16]. Another challenge would be to verify the extension of parallel NDFS with subsumption for timed automata [LOD+13].

As mentioned, our work can be considered as a first step towards a library for the verification of graph-based (multi-core) model checking algorithms. It will be an interesting line of future work to continue this, and to develop a full-fledged verification library for common subtasks, like graph manipulations, generic DFS procedures, and termination detection.

Part II

Advances in Concurrent Program Verification

CHAPTER 4

An Abstraction Technique for Describing Concurrent Program Behaviour

Abstract

This chapter presents a technique to reason about functional properties of shared-memory concurrent software by means of abstraction. The abstract behaviour of the program is described using process algebras. In the program we indicate which concrete atomic steps correspond to the actions that are used in the process algebra term. Each action comes with a specification that describes its effect on the shared state. Program logics are used to show that the concrete program steps adhere to this specification. Separately, we also use program logics to prove that the program behaves as described by the process algebra term. Finally, via process algebraic reasoning we derive properties that hold for the program from its abstraction. This technique allows reasoning about the behaviour of highly concurrent, non-deterministic and possibly non-terminating programs. This chapter discusses various verification examples to illustrate our approach. The verification technique is implemented as part of the VerCors concurrency verifier. We demonstrate that our technique is capable of verifying data- and control-flow properties that are hard to verify with alternative approaches, especially with mechanised tool support.

4.1 Introduction

In Chapters 2 and 3 we exclusively considered and verified global behavioural properties that could be specified as global, first-order logic invariants. Nevertheless, in practice not all properties can conveniently be expressed in this way. For example, properties of control-flow, or properties relying on intricate thread interaction, are difficult to specify as a global invariant, but are instead more easily specified as a transition system, or a state machine. This chapter introduces an abstraction

technique for specifying such properties as a process algebra, and verifying them in a sound manner using the VerCors concurrency verifier.

The major challenge when reasoning about concurrent or distributed software is to come up with an appropriate abstraction that provides sufficient detail to capture the intended properties, while at the same time making verification manageable. This chapter presents a new powerful abstraction approach that enables reasoning about the intended properties of the program in a purely non-deterministic setting, and can abstract code at different levels of granularity. The presentation of the abstraction technique in this chapter focuses on shared-memory concurrent programs and safety properties, but many extensions may be explored. For example for distributed programs (see Chapter 7), or progress properties, as sketched in the paragraph on future work (in §4.8.1). This chapter illustrates our approach by discussing multiple verification examples in which we verify various data- and control-flow properties. We demonstrate that the proposed technique can be used to verify program properties that are hard to verify with alternative approaches, especially in a practical manner via mechanised tools.

To motivate our approach, consider the program shown in Figure 4.1. The figure shows a parallel version of the classical Euclidean algorithm for finding a greatest common divisor, $\texttt{gcd}(x, y)$, of two given positive integers x and y. This is done by forking two concurrent threads: one thread to decrement the value of x whenever possible, and one thread to decrement the value of y.

We are interested in verifying deductively that this program indeed computes the greatest common divisor of x and y. To accomplish this in a scalable fashion requires that our technique be *modular*, or more precisely procedure-modular and thread-modular, to allow the individual functions and threads to be analysed independently of one another. The main challenge in achieving this lies in finding a suitable way of capturing the *effect* of function calls and threads on the shared memory in a way that is independent of the other functions and threads. Our proposal is to capture these effects as *sequences of exclusive accesses* (in this example increments and decrements) to shared memory (in this example the heap locations ℓ_x and ℓ_y). We abstract such accesses into so-called *actions*, and their sequences into process algebraic terms.

In our example above we abstract the assignments $[\ell_x] := [\ell_x] - [\ell_y]$ and $[\ell_y] := [\ell_y] - [\ell_x]$ needed to decrease the values at heap locations ℓ_x and ℓ_y into actions decrx and decry, respectively. Action behaviour is specified by means of *contracts* consisting of a guard and an effect. The explanation of the details of this are deferred to Section 4.3. Using these actions, we can specify the effects of the two threads by means of the process algebra terms Tx and Ty, respectively, which are

```
1  /* Two references to integers stored on the heap. */
2  int ref ℓ_x, ℓ_y;
3
4  /* Startup procedure for the concurrent GCD algorithm. */
5  int startgcd(int x, int y)
6  |  [ℓ_x] := x; // writing x to the heap at location ℓ_x.
7  |  [ℓ_y] := y; // writing y to the heap at location ℓ_y.
8  |  ref t_1 := fork thread_x();
9  |  ref t_2 := fork thread_y();
10 |  join t_1;
11 |  join t_2;
12 |  return [ℓ_x]; // returning the value at heap location ℓ_x.
13 end
```

(a) The starting procedure for the concurrent GCD algorithm, which spawns two two threads that each do a part of the computation.

```
10 void thread_x()
11 |  boolean stop := false;
12 |  while (¬stop) do
13 |  |  atomic lock do
14 |  |  |  if ([ℓ_x] > [ℓ_y]) then
15 |  |  |  |  [ℓ_x] := [ℓ_x] − [ℓ_y];
16 |  |  |  end
17 |  |  |  stop := [ℓ_x] = [ℓ_y];
18 |  |  end
19 |  end
20 end
```

```
21 void thread_y()
22 |  boolean stop := false;
23 |  while (¬stop) do
24 |  |  atomic lock do
25 |  |  |  if ([ℓ_y] > [ℓ_x]) then
26 |  |  |  |  [ℓ_y] := [ℓ_y] − [ℓ_x];
27 |  |  |  end
28 |  |  |  stop := [ℓ_y] = [ℓ_x];
29 |  |  end
30 |  end
31 end
```

(b) The (symmetric) implementations of `thread_a` and `thread_b`, where the $[\cdot]$ notation denotes *heap dereferencing*, used to read from, or write to, the heap.

Figure 4.1: A parallel implementation of the Euclidean algorithm for finding the greatest common divisor of two (positive) integers x and y.

defined as follows:

$$\textbf{process } \mathsf{Tx} \triangleq \mathsf{decrx} \cdot \mathsf{Tx} + \mathsf{done}; \qquad \textbf{process } \mathsf{Ty} \triangleq \mathsf{decry} \cdot \mathsf{Ty} + \mathsf{done};$$

Here the action done indicates termination of a process. The functional behaviour of the program can then be specified by the process GCD defined as the term Tx ‖ Ty. Standard process algebraic reasoning can be applied to show that executing GCD results in calculating the correct gcd.

Therefore, by proving that the implementation executes as prescribed by GCD, we simultaneously establish its functional property of producing the correct result. The GCD process thus describes the program behaviour.

Once the program has been specified, the access exclusiveness of the actions is verified by a suitable extension of separation logic with permission accounting [O'H07, BCOP05]. On top of it, we develop rules that allow to prove, in a thread-local fashion, that the program indeed follows its prescribed process. The details of our technique applied to the above program are presented in Section 4.3.

In previous work [BHZS15, ZS15] we developed an approach that records the actions of a concurrent program as the program executes. Reasoning with this approach is only suitable for terminating programs, and occurs at the end of its execution, requiring the identification of repeating patterns. In contrast, the current approach requires a process algebra term upfront that describes the patterns of atomic concurrent actions, which allows specifying of functional behaviour of reactive, non-terminating programs. For instance, we can verify properties such as "the values of the shared locations a and b will be equal infinitely often", expressed in LTL by the formula $\Box\Diamond([\ell_x] = [\ell_y])$, of a program that forks separate threads to modify ℓ_x and ℓ_y, similarly to the above parallel GCD program.

Compared to many of the other modern separation logics to reason about concurrent programs, such as CAP [DYDG$^+$10], CaReSL [TDB13], Iris [JSS$^+$15], and TaDA [RPDYG14], our approach does the abstraction at a different level. Our abstraction connects program code with individual actions, while these other approaches essentially encode an abstract state machine, describing how program steps evolve from one abstract program state to the next abstract program state, and explicitly consider the changes that could be made by the thread environment. As a result, in our approach the global properties are specified in a way that is independent of the program implementation. This makes it easier for non-experts to understand the program specification.

4.1.1 Contributions

The main contributions of this chapter are[1]:

- An abstraction technique to specify and verify the behaviour of possibly non-terminating, shared-memory concurrent programs, where the abstractions are implementation-independent and may be non-deterministic;
- A number of verification examples that illustrate our approach and can mechanically be verified via the VerCors verifier; and thus
- Tool support for our model-based reasoning approach.

4.1.2 Chapter Outline

The remainder of this chapter is organised as follows. Section 4.2 gives a brief background on process algebras, and introduces the process algebra language that is used throughout the remainder of this chapter. Section 4.3 then illustrates in more detail how our abstraction technique is used in the verification of the parallel GCD example (Figure 4.1). Section 4.4 elaborates on the proof rules of the approach, after which Section 4.5 revisits the parallel GCD example once more, to explain how the proof rules are used and to show all the intermediate proof steps. After that, Section 4.6 discusses three more verification examples to demonstrate our approach: (i) verifying a concurrent counter, (ii) verifying a locking protocol, and (iii) verifying a classical leader election protocol. Finally, Section 4.7 discusses related work, after which Section 4.8 concludes.

4.2 Background on Process Algebra

The abstract models we use to reason about shared-memory program behaviour are represented as process algebra terms. We use a subset of the μCRL [GP95, GM14] language for this, as a suitably expressive process algebra with data. The basic primitives of the language are *actions*, which represent basic, indivisible process behaviours. These actions can be composed in different ways to construct *processes*. Actions and processes may both may be parameterised by data.

Our process algebra language has the following structure:

$$P, Q \ ::= \ \varepsilon \mid \delta \mid a(\overline{E}) \mid p(\overline{E}) \mid P \cdot Q \mid P + Q \mid P \parallel Q \mid \textbf{if } B \textbf{ then } P \textbf{ else } P$$

In the above definition, E is an arithmetic expression, B a Boolean expression, a an action label, and p a process label. With $\overline{E}$ we mean a sequence of expressions.

[1] This chapter is based on the article [OBG+17].

The empty process is denoted ε and the deadlock process by δ. The process $a(\overline{E})$ is an action call and $p(\overline{E})$ a recursive process invocation, with $\overline{E}$ the argument sequence. Two process terms P and Q may compose either sequentially $P \cdot Q$ or alternatively $P+Q$. The parallel composition $P \parallel Q$ allows the actions of P and Q to be interleaved during execution. The conditional construct **if** B **then** P **else** Q resembles the classical "if-then-else"; it yields either P or Q, depending on the result of evaluating the expression B. The else branch is sometimes omitted, in which case the abbreviation **if** B **then** $P \triangleq$ **if** B **then** P **else** δ is used.

4.3 Motivating Example

This section demonstrates our approach by verifying functional correctness of the parallel GCD verification example that was discussed in the introduction. With *functional correctness* we mean verifying that the correct value has been calculated after termination of the program. In this example, the correct value is the mathematical GCD of the two (positive) values given as input to the algorithm.

Our approach consists of the following steps:

1. *Actions* and their associated *guards* and *effects* are defined that describe in what ways the program is allowed to make updates to shared memory.
2. The actions are composed into *processes* by using the process algebraic connectives discussed in Section 4.2. These processes determine the desired behaviour of (parts of) the concrete program. Notably, processes that are composed in parallel correspond to forked threads in the program.
3. All defined processes that have a *contract* (pre- and postconditions) are verified. Concretely, we automatically verify with VerCors whether the postconditions of processes can be ensured by all traces that start from a state in which the precondition is satisfied.
4. Finally we verify that every thread forked by the program *behaves as specified* by the process algebraic specification. If this is the case, the verification results that are established from the previous step can be obtained and used in the program logic.

Tool support for model-based reasoning is provided as part of the VerCors deductive verifier [BH14c, BDHO17]. Preliminaries on the use of VerCors can be found in Chapter 2. The VerCors verifier aims to verify programs under various concurrency models, notably heterogeneous and homogeneous concurrency, written in high-level programming languages such as Java and C. Although most of the examples presented in this chapter have been worked out and verified in PVL, the Prototypal Verification Language that we use to prototype new verification features, tool support is also provided for both Java and C.

All verification examples presented in this chapter have been verified with the VerCors verifier, and are furthermore accessible via the online Git repository that comes with this thesis [Sup].

4.3.1 Parallel GCD

We demonstrate our model-based reasoning approach by capturing the functional behaviour of a parallel GCD algorithm. The parallel GCD verification problem is taken from the VerifyThis verification competition held at ETAPS 2015[2] and considers a parallel version of the classical Euclidean algorithm.

Definition 4.3.1 (Euclidean algorithm)**.** *The classical Euclidean algorithm is defined as a function* $\mathsf{gcd} : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ *in the following way.*

$$\mathsf{gcd}(x, y) \triangleq \begin{cases} \mathsf{gcd}(x - y, y) & \textit{if } x > y \\ \mathsf{gcd}(x, y - x) & \textit{if } y > x \\ x & \textit{if } x = y \end{cases}$$

The parallel version of this algorithm uses two concurrent threads: the first thread continuously decrements the value of x when $x > y$, while the second thread continuously decrements the value of y whenever $y > x$, and this process continues until x and y converge to the gcd of the two original input values. Model-based reasoning is used here to describe the interleaving of the concurrent threads and to prove functional correctness of the parallel algorithm in an elegant way.

Figure 4.2 presents the setup of the GCD process, which models the behaviour of a parallel GCD algorithm with respect to the two global variables x and y. The GCD process uses three different actions, named: decrx, decry, and done. Performing the action decry captures the effect of decreasing x by an amount y, provided that $x > y$ before the action is performed. Likewise, performing decry captures the effect of decreasing y. Finally, the done action may be performed when $x = y$ and is used to indicate termination of the algorithm.

The GCD process is defined as the parallel composition of two processes; the process Tx describes the behaviour of the thread that decreases x, and Ty describes the behaviour of the thread that decreases y. The GCD process requires that the shared variables x and y are both positive, and ensures that both x and y contain the gcd of the original values of x and y. Proving that GCD satisfies its contract is done via standard process algebraic reasoning: first GCD is converted to a linear process (i.e., a process without parallel constructs), which is then analysed (e.g., via model checking) to show that every thread interleaving leads to a correct answer, in this case $\mathsf{gcd}(\backslash\mathbf{old}(x), \backslash\mathbf{old}(y))$.

[2] see also `http://etaps2015.verifythis.org`.

```
1  /* Process-algebraic state */
2  shared int x, y;
3
4  /* Abstract description of the decrement of x */
5  guard x > 0 ∧ y > x;
6  effect x = \old(x);
7  effect y = \old(y) − \old(x);
8  action decrx;
9
10 /* Abstract description of the decrement of y */
11 guard y > 0 ∧ x > y;
12 effect x = \old(x) − \old(y);
13 effect y = \old(y);
14 action decry;
15
16 /* Abstract description of the termination condition */
17 guard x = y;
18 action done;
19
20 /* Process-algebraic behavioural specification of the two threads */
21 process Tx := decrx · Tx + done;
22 process Ty := decry · Ty + done;
23
24 /* Process-algebraic specification of the parallel GCD algorithm */
25 requires x > 0 ∧ y > 0;
26 ensures x = y = gcd(\old(x), \old(y));
27 process GCD := Tx ‖ Ty;
```

Figure 4.2: The processes used for the parallel GCD verification example. Three actions are used: decrx, decry, and done. The first two actions capture modifications made to the (shared) variables x and y, and done indicates termination of the algorithm.

4.3.2 Verifying the Correctness of Parallel GCD

Figure 4.3 shows the `startgcd` function, which is the entry point of the implementation of the parallel GCD algorithm[3]. This implementation spawns two threads

[3]It should be noted that the presentation is slightly different from the version that is verified by VerCors, to better connect to the theory discussed in the earlier sections to the case study. Notably, as discussed earlier, VerCors uses Implicit Dynamic Frames [LMS09] as the underlying logical framework, which is equivalent to Separation Logic [PS11] but handles ownership slightly

```
1  /* Two references to integers stored on the heap. */
2  int ref ℓ_x, ℓ_y;
3
4  /* The resources (ownerships) and knowledge protected by the lock. */
5  lock_invariant lock := ∃v'_x, v'_y . ℓ_x ↪[1]_proc v'_x * ℓ_y ↪[1]_proc v'_y * v'_x > 0 * v'_y > 0;
6
7  /* Startup procedure for the concurrent GCD algorithm. */
8  given v_x, v_y;
9  requires ℓ_x ↪[1] v_x * ℓ_y ↪[1] v_y * v_x > 0 * v_y > 0;
10 ensures ∃v'_x, v'_y . ℓ_x ↪[1] v'_x * ℓ_y ↪[1] v'_y * v'_x = v'_y = gcd(v_x, v_y);
11 void startgcd()
12 |  ref m := process GCD over ⟨x ↦ ℓ_x, y ↦ ℓ_y⟩;
13 |  invariant lock do
14 |  |  ref t_1 := fork thread_x(m);
15 |  |  ref t_2 := fork thread_y(m);
16 |  |  join t_1;
17 |  |  join t_2;
18 |  end
19 |  finish m;
20 end
```

Figure 4.3: The entry point of the parallel GCD algorithm. Two threads are forked and continuously decrement the values at heap locations ℓ_x and ℓ_y until these become equal, which is when the threads converge. The functional property of actually producing a gcd is proven by analysing the process (externally).

to compute the GCD concurrently, and uses shared-memory as the communication model between these threads. More specifically, the threads communicate via two shared locations on the heap, named ℓ_x and ℓ_y in the figure, which are globally declared on line 2. Moreover, we use the notation $[\ell_x]$ to denote *heap dereferencing*, i.e., obtaining the value at location ℓ_x from the heap, and $[\ell_x] := v$ to denote *heap writing*, i.e., writing the value v to the heap at location ℓ_x.

The `startgcd` method has been annotated with *code annotations*, displayed in blue. These annotations are specified manually, by the user of VerCors.

According to `startgcd`'s contract (lines 8–10), the references ℓ_x and ℓ_y must indeed be allocated on the heap, and must point to some *given* positive integer values v_x and v_y, respectively. This is expressed by the precondition on line 9.

differently. The details of this are deferred to [BDHO17, JOSH18].

On line 12 a new process-algebraic model is initialised, which is given the reference m. This model describes that all further updates to the shared-memory locations ℓ_x and ℓ_y must behave as specified by the GCD process. Here the process-algebraic variable x is linked/connected to the heap location ℓ_x (i.e., protecting its modifications), while y is linked to heap location ℓ_y. This means that, after the initialisation of GCD, the program can no longer "freely" write to locations ℓ_x or ℓ_y: modifications to the heap at these locations must happen in the context of a process-algebraic action, for example decrx or decry.

Since GCD is defined as the parallel composition of the processes Tx and Ty, its definition may be *matched* in the program code by forking two concurrent threads and giving each thread one of the components of Tx ∥ Ty. In this case, the thread that executes `thread_x` continues from the process Tx, whereas the thread executing `thread_y` continues from Ty. By later joining the two threads and finishing the model by using the ghost statement **finish** (which is only possible if GCD has been fully executed), we may establish that `startgcd` satisfies its contract. However, we still have to show that the threads executing `thread_x` and `thread_y` behave as described by the model m.

Moreover, lines 14–17 are executed in the context of an **invariant**, which enables locking and unlocking. In particular, when one of the threads obtains a lock, it gets access to the resources that are protected by the *lock invariant*, which is named *lock* and specified on line 5. In this case, the lock invariant *lock* protects write access to the heap at locations ℓ_x and ℓ_y, and states that these locations point to positive integer values. These write accesses are transferred from the thread to the lock at line 13, and are transferred back to the thread on line 18.

Thread implementation. Figure 4.4 shows the implementation of `thread_x`. The implementation of `thread_y` is symmetric to `thread_x` and therefore omitted. The precondition of `thread_x` states that a predicate of the form $\mathsf{Proc}_{\frac{1}{2}}(m, \mathsf{Tx})$ is required. This predicate is used to indicate that there exists an initialised program abstraction named m, that is able to execute the Tx (sub)process. Moreover, the fractional permission $\frac{1}{2}$ indicates that this predicate describes only a fraction of the entire process-algebraic model: in this case, `thread_y` will have the other half, and will execute according to Ty. We will later see that these Proc predicates can be split and merged, and be distributed over different threads.

The connection between the process and program code is made via the **action** specification statements. To illustrate, in the function `thread_x` the decrement of $[\ell_x]$ on line 13 is performed in the context of an *action block*, thereby forcing the Tx process in the $\mathsf{Proc}_{\frac{1}{2}}$ predicate to perform the decrx action. The **guard** of decrx specifies the condition under which decrx can be executed, and the **effect** clause describes the effect on the (shared) state as result of executing decrx. More

```
1  /* Specification and implementation for the thread decrementing at ℓ_x. */
2  requires Proc_{1/2}(m, Tx);
3  ensures Proc_{1/2}(m, ε);
4  void thread_x(ref m)
5  |  boolean stop := false;
6  |
7  |  loop_invariant ¬stop ⟹ Proc_{1/2}(m, Tx);
8  |  loop_invariant stop ⟹ Proc_{1/2}(m, ε);
9  |  while (¬stop) do
10 |  |  atomic lock do
11 |  |  |  if ([ℓ_x] > [ℓ_y]) then
12 |  |  |  |  action m.decrx do
13 |  |  |  |  |  [ℓ_x] := [ℓ_x] − [ℓ_y];
14 |  |  |  |  end
15 |  |  |  end
16 |  |  |  else if ([ℓ_x] = [ℓ_y]) then
17 |  |  |  |  action m.done do
18 |  |  |  |  |  stop := true;
19 |  |  |  |  end
20 |  |  |  end
21 |  |  end
22 |  end
23 end
```

Figure 4.4: The implementation of the procedure used by `thread_x` for calculating the GCD, by decrementing $[\ell_x]$. The implementation of `thread_y` is symmetric.

specifically, the proof system will ensure that the modification of $[\ell_x]$ on line 13 complies with the guard and effect of the decrx action, where the process-algebraic variable x is linked to ℓ_x, while y is linked to ℓ_y. Eventually, both threads execute the done action to indicate their termination.

The VerCors verifier can automatically verify the parallel GCD verification example discussed above, including the analysis of the GCD process. Moreover, observe that the contract of `thread_x` does not express any functional properties on the behaviour of its implementation, other than the new Proc predicates. The functional postcondition of calculating the correct result is established entirely on the process-algebraic level.

The next section gives more details on the assertion language, notably the new Proc_π and $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ assertions, as well as the proof system. After that, we come back

to the GCD example in Section 4.5, and also discuss all intermediate verification steps, before Section 4.6 discusses some more applications of our approach.

4.4 Program Logic

This section shortly elaborates on the assertion language and the proof rules of the verification approach, as used internally by VerCors to reason about abstractions. We do not present a full formalisation just yet, for full details we refer to Chapter 5. We now only sketch and discuss the proof rules related to model-based reasoning.

4.4.1 Assertions

The program logic builds on standard CSL with permission accounting [Boy03]. The following grammar defines its assertion language:

$$
\begin{aligned}
\mathcal{P}, \mathcal{Q} \ &::= B \mid \forall x.\mathcal{P} \mid \exists x.\mathcal{P} \mid \mathcal{P} * \mathcal{Q} \mid && \text{(standard CSL connectives)} \\
&\quad \ell \overset{\pi}{\hookrightarrow}_t E \mid \mathsf{Proc}_\pi(x, p, P, \Pi) && \text{(heap ownership, abstraction ownership)} \\
t \ &::= \mathsf{std} \mid \mathsf{proc} \mid \mathsf{act} && \text{(heap ownership types)} \\
\Pi \ &::= \langle \ell_0 \mapsto E_0, \ldots, \ell_n \mapsto E_n \rangle && \text{(abstraction binders)}
\end{aligned}
$$

where E are arithmetic expressions, B are Boolean expressions, x are variables, π are fractional permissions, ℓ are references to heap locations, and p are process labels. Note that the specification language implemented in VerCors supports more assertion constructs; we only highlight a subset to elaborate on our approach.

Heap ownership. Instead of using a single points-to ownership predicate, like in standard CSL, our extensions require three different points-to predicates:

- The $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E$ predicate is the standard points-to predicate from CSL. It gives write permission to the heap location expressed by ℓ in case $\pi = 1$, and gives read access in case $\pi \in (0, 1]$. This predicate also represents the knowledge that the heap contains the value expressed by E at location ℓ.
- The *process points-to predicate* $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E$ is similar to $\overset{\pi}{\hookrightarrow}_{\mathsf{std}}$, but indicates that the heap location at ℓ is *bound* by an abstract model. Since all changes to this heap location must be captured by the model, the $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicate *only* gives read permission to ℓ, even when $\pi = 1$.
- The *action points-to predicate* $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E$ gives read- or write access to the heap location ℓ in the context of an **action** block. As a precondition, **action** blocks require $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates for all heap locations that are accessed in

$$\frac{t \in \{\mathsf{std}, \mathsf{proc}, \mathsf{act}\}}{\ell \hookrightarrow^{\pi_1+\pi_2}_t E \iff \ell \hookrightarrow^{\pi_1}_t E * \ell \hookrightarrow^{\pi_2}_t E} \hookrightarrow\text{-SPLIT-MERGE}$$

$$\mathsf{Proc}_{\pi_1+\pi_2}(E, p, P_1 \parallel P_2, \Pi) \iff \mathsf{Proc}_{\pi_1}(E, p, P_1, \Pi) * \mathsf{Proc}_{\pi_2}(E, p, P_2, \Pi) \quad \text{PROC-SPLIT-MERGE}$$

Figure 4.5: Simplified proof rules for splitting and merging ownerships.

their body. These predicates are then converted to $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ predicates, which give reading permission if $\pi \in (0, 1]$, and writing permission if $\pi = 1$.

The notation $\ell \overset{\pi}{\hookrightarrow} E \triangleq \ell \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E$ (without the std subscript) is sometimes used to abbreviate standard heap ownership predicates.

All three points-to ownership predicates can be split and merged along the associated fractional permission, to be distributed among concurrent threads, as shown by the $\hookrightarrow$-SPLIT-MERGE proof rule in Figure 4.5.

Essentially, three different predicates are needed to ensure soundness of the verification approach. When a heap location ℓ becomes bound by an abstract model, its $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E$ predicate is converted to an $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E$ predicate in the program logic. As an effect, the value at ℓ cannot just be changed, since the $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicate does not permit writing to ℓ (even when $\pi = 1$). However, the value at ℓ *can* be changed in the context of an action block, as the rule for action blocks in our program logic converts all affected $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates to $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ predicates, and $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ again allows writing to ℓ. The intuition is that, by converting $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E$ predicates to $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E$ predicates, all changes to ℓ are *forced* to occur in the context of action blocks, and this allows describing all changes to ℓ as processes. Consequently, by reasoning over these process algebra terms, we may reason about all possible changes to ℓ, and our verification approach allows the result of this reasoning to be used in the proof system.

Process ownership. The second main extension our program logic makes to standard CSL is the $\mathsf{Proc}_\pi(x, p, P, \Pi)$ predicate, which represents the knowledge of the existence of an abstract process-algebraic model that:

1. Is identified by the variable/reference x;
2. Was initialised by invoking the process labelled as p;

3. Is described by the process term P; and
4. Connects process-algebraic state (variables) to shared program state (heap locations) via the $\Pi = \langle x_0 \mapsto \ell_0, \ldots, x_n \mapsto \ell_n \rangle$ mappings, which we refer to as *binders*.

For brevity we omit the p and Π components from the annotations in all example programs, and simply write $\mathsf{Proc}_\pi(x, P)$ instead. This is because these two components are constant, and can never be changed in the proof system. The third component P is the remaining process that is to be "executed" by the program (i.e., "reduced" in its proof derivation).

The Proc_π predicates may be split and merged along the fractional permission and parallel compositions in its process component, likewise to the heap ownership predicates, as shown by the PROC-SPLIT-MERGE rule in Figure 4.5.

4.4.2 Proof System

Before discussing the proof rules of our model-based verification approach, let us first define the following auxiliary operation, $\Pi{\downarrow_X}$, for removing all mappings in Π with a variable that is not in the set X of process-algebraic variables.

Definition 4.4.1 (Binder restriction)**.** *Let* Π *be a binder and* X *be a set of process-algebraic variables. The* restriction of Π to X*, written* $\Pi{\downarrow_X}$*, gives a new binder containing only the elements from* Π *that have a process-algebraic variable in* X*:*

$$\mathsf{nil}{\downarrow_X} \triangleq \mathsf{nil} \qquad (x \mapsto E :: \langle xs \rangle){\downarrow_X} \triangleq \begin{cases} x \mapsto E :: (\langle xs \rangle{\downarrow_X}) & \text{if } x \in X \\ \langle xs \rangle{\downarrow_X} & \text{if } x \notin X \end{cases}$$

To give some intuition on how the underlying proof system works, Figure 4.6 shows the *simplified* proof rules. The actual proof rules are more involved, and are discussed in detail in Chapter 5, together with their soundness proof.

The HT-READ rule allows reading from the heap, which can be done with any heap ownership predicate (that is, $\overset{\pi}{\hookrightarrow}_t$ for any permission type t). Writing to shared memory is only allowed by HT-WRITE with a *full* heap ownership predicate that is *not* of type proc; if the targeted heap location is bound by an abstract model, then all changes must be done in an action block (see the HT-ACTION rule). HT-PROC-INIT handles the initialisation of a model, which on the specification level converts all affected $\overset{1}{\hookrightarrow}_{\mathsf{std}}$ predicates to $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$, and produces a *full* Proc_1 predicate. HT-PROC-FINISH handles model finalisation: it requires a fully executed Proc_1 predicate (holding the process ε) and converts all affected $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$ predicates back to $\overset{1}{\hookrightarrow}_{\mathsf{std}}$. Finally, HT-ACTION handles action blocks. If a proof can be derived for the

HT-READ

$$\frac{x \notin \mathsf{fv}(E)}{\vdash \{\mathcal{P}[x/E] * \ell \overset{\pi}{\hookrightarrow}_t E\}\, x := [\ell]\, \{\mathcal{P} * \ell \overset{\pi}{\hookrightarrow}_t E\}}$$

HT-WRITE

$$\frac{t \neq \mathsf{proc}}{\vdash \{\ell \overset{1}{\hookrightarrow}_t -\}\, [\ell] := E\, \{\ell \overset{1}{\hookrightarrow}_t E\}}$$

HT-PROC-INIT

$$\frac{\Pi = \langle x_0 \mapsto \ell_0, \ldots, x_n \mapsto \ell_n \rangle}{\vdash \begin{array}{c} \left\{ (*_{i\in\{0,\ldots,n\}} \ell_i \overset{1}{\hookrightarrow}_{\mathsf{std}} E_i) * \mathsf{precondition}(p)[x_i/E_i]_{\forall i\in\{0,\ldots,n\}} \right\} \\ \textbf{ref } m := \textbf{process } p \textbf{ over } \Pi \\ \left\{ (*_{i\in\{0,\ldots,n\}} \ell_i \overset{1}{\hookrightarrow}_{\mathsf{proc}} E_i) * \mathsf{Proc}_1(m, \mathsf{def}(p), P, \Pi) \right\} \end{array}}$$

HT-PROC-FINISH

$$\frac{\Pi = \langle x_0 \mapsto \ell_0, \ldots, x_n \mapsto \ell_n \rangle}{\vdash \begin{array}{c} \left\{ (*_{i\in\{0,\ldots,n\}} \ell_i \overset{1}{\hookrightarrow}_{\mathsf{proc}} E_i) * \mathsf{Proc}_1(m, p, \varepsilon, \Pi) \right\} \\ \textbf{finish } m \\ \left\{ (*_{i\in\{0,\ldots,n\}} \ell_i \overset{1}{\hookrightarrow}_{\mathsf{std}} E_i) * \mathsf{postcondition}(p)[x_i/E_i]_{\forall i\in\{0,\ldots,n\}} \right\} \end{array}}$$

HT-ACTION

$$\frac{\begin{array}{c} FV = \mathsf{fv}(\mathsf{guard}(a)) \cup \mathsf{fv}(\mathsf{effect}(a)) \qquad \Pi{\downarrow_{FV}} = \langle x_0 \mapsto \ell_0, \ldots, x_n \mapsto \ell_n \rangle \\ B_{guard} = \mathsf{guard}(a)[x_i/E_i]_{\forall i\in\{0,\ldots,n\}} \qquad B_{effect} = \mathsf{effect}(a)[x_i/E'_i]_{\forall i\in\{0,\ldots,n\}} \\ \vdash \left\{ (*_{i\in\{0,\ldots,n\}} \ell_i \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E_i) * B_{guard} \right\} S \left\{ (*_{i\in\{0,\ldots,n\}} \ell_i \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E'_i) * B_{effect} \right\} \end{array}}{\vdash \begin{array}{c} \left\{ (*_{i\in\{0,\ldots,n\}} \ell_i \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E_i) * \mathsf{Proc}_\pi(m, p, a(\overline{E}) \cdot P + Q, \Pi) * B_{guard} \right\} \\ \textbf{action } m.a(\overline{E}) \textbf{ do } S \\ \left\{ (*_{i\in\{0,\ldots,n\}} \ell_i \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E'_i) * \mathsf{Proc}_\pi(m, p, P, \Pi) * B_{effect} \right\} \end{array}}$$

Figure 4.6: Selected simplified proof rules that deal with handling heaps and process-algebraic models. The semantic details and soundness of these proof rules are deferred to Chapter 5.

body S of the action block that (i) respects the guard and effect of the action, and (ii) has the $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$ predicates of all heap locations accessed in S converted to $\overset{1}{\hookrightarrow}_{\mathsf{act}}$; then a similar proof can be established for the entire action block. Observe that HT-ACTION requires and consumes the matching action call $a(\overline{E})$ in the process component $a(\overline{E}) \cdot P + Q$ of the Proc_π predicate, so that Proc_π is left with P and Q is discarded, as by performing $a(\overline{E})$ the choice is made not to follow execution as prescribed by Q.

4.5 Parallel GCD—Intermediate Proof Steps

Let us now come back to the parallel GCD example that we annotated in Section 4.3, and discuss the intermediate proof steps. Figure 4.7 shows the algorithm's

```
1  int ref ℓ_x, ℓ_y;
2
3  lock_invariant lock := ∃v'_x, v'_y . ℓ_x ↪¹_proc v'_x ∗ ℓ_y ↪¹_proc v'_y ∗ v'_x > 0 ∗ v'_y > 0;
4
5  given v_x, v_y;
6  requires ℓ_x ↪¹ v_x ∗ ℓ_y ↪¹ v_y ∗ v_x > 0 ∗ v_y > 0;
7  ensures ∃v'_x, v'_y . ℓ_x ↪¹ v'_x ∗ ℓ_y ↪¹ v'_y ∗ v'_x = v'_y = gcd(v_x, v_y);
8  void startgcd()
9      {ℓ_x ↪¹ v_x ∗ ℓ_y ↪¹ v_y ∗ v_x > 0 ∗ v_y > 0}
10     ref m := process GCD over ⟨x ↦ ℓ_x, y ↦ ℓ_y⟩;
11     {Proc_1(m, GCD) ∗ ℓ_x ↪¹_proc v_x ∗ ℓ_y ↪¹_proc v_y ∗ v_x > 0 ∗ v_y > 0}
12     {Proc_1(m, GCD) ∗ ∃v'_x, v'_y . ℓ_x ↪¹_proc v'_x ∗ ℓ_y ↪¹_proc v'_y ∗ v'_x > 0 ∗ v'_y > 0}
13     invariant lock do
14         {Proc_1(m, GCD)}
15         {Proc_1(m, Tx ‖ Ty)}
16         {Proc_1/2(m, Tx) ∗ Proc_1/2(m, Ty)}
17         ref t_1 := fork thread_x(m);
18         {Proc_1/2(m, Ty)}
19         ref t_2 := fork thread_y(m);
20         { }
21         join t_1;
22         {Proc_1/2(m, ε)}
23         join t_2;
24         {Proc_1/2(m, ε) ∗ Proc_1/2(m, ε)}
25         {Proc_1(m, ε ‖ ε)}
26         {Proc_1(m, ε)}
27     end
28     {Proc_1(m, ε) ∗ ∃v'_x, v'_y . ℓ_x ↪¹_proc v'_x ∗ ℓ_y ↪¹_proc v'_y ∗ v'_x > 0 ∗ v'_y > 0}
29     finish m;
30     {∃v'_x, v'_y . ℓ_x ↪¹ v'_x ∗ ℓ_y ↪¹ v'_y ∗ v'_x > 0 ∗ v'_y > 0 ∗ v'_x = v'_y = gcd(v_x, v_y)}
31 end
```

Figure 4.7: The entry point of the parallel GCD algorithm. All assertions $\{\mathcal{P}\}$ that are displayed in purple are intermediate verification steps that are automatically generated and proven by VerCors.

```
1  requires Proc_{1/2}(m, Tx);
2  ensures Proc_{1/2}(m, ε);
3  void thread_x(ref m)
4  | boolean stop := false;
5  |
6  | loop_invariant ¬stop ⟹ Proc_{1/2}(m, Tx);
7  | loop_invariant stop ⟹ Proc_{1/2}(m, ε);
8  | while (¬stop) do
9  | | {Proc_{1/2}(m, Tx)}
10 | | atomic lock do
11 | | | {Proc_{1/2}(m, Tx) ∗ ∃v'_x, v'_y . ℓ_x ↪¹_proc v'_x ∗ ℓ_y ↪¹_proc v'_y ∗ v'_x > 0 ∗ v'_y > 0}
12 | | | int w_x, w_y := [ℓ_x], [ℓ_y];
13 | | | {Proc_{1/2}(m, decrx · Tx + done) ∗ ℓ_x ↪¹_proc w_x ∗ ℓ_y ↪¹_proc w_y ∗ ···}
14 | | | if (w_x > w_y) then
15 | | | | {Proc_{1/2}(m, decrx · Tx + done) ∗ ℓ_x ↪¹_proc w_x ∗ ℓ_y ↪¹_proc w_y ∗ w_x > w_y ∗ ···}
16 | | | | action m.decrx do
17 | | | | | {ℓ_x ↪¹_act w_x ∗ ℓ_y ↪¹_act w_y ∗ ···}
18 | | | | | [ℓ_x] := w_x − w_y;
19 | | | | | {ℓ_x ↪¹_act w_x − w_y ∗ ℓ_y ↪¹_act w_y ∗ ···}
20 | | | | end
21 | | | | {Proc_{1/2}(m, Tx) ∗ ℓ_x ↪¹_proc w_x − w_y ∗ b ↪¹_proc w_y ∗ ···}
22 | | | end
23 | | | else if (w_x = w_y) then
24 | | | | {Proc_{1/2}(m, decrx · Tx + done) ∗ ℓ_x ↪¹_proc w_x ∗ ℓ_y ↪¹_proc w_y ∗ ···}
25 | | | | action m.done do
26 | | | | | stop := true;
27 | | | | end
28 | | | | {Proc_{1/2}(m, ε) ∗ ℓ_x ↪¹_proc w_x ∗ ℓ_y ↪¹_proc w_y ∗ ···}
29 | | | end
30 | | end
31 | | {¬stop ∧ Proc_{1/2}(m, Tx) ∨ stop ∧ Proc_{1/2}(m, ε)}
32 | end
33 end
```

Figure 4.8: The implementation of the procedures used by the two threads for calculating the GCD: `thread_x` decrements $[\ell_x]$, while `thread_y` decrements $[\ell_y]$.

entry point, `startgcd`, but with all intermediate proof steps displayed in purple.

On line 10, the process-algebraic model GCD is initialised, which by HT-PROC-INIT produces a $\mathsf{Proc}_1(m, \mathsf{GCD})$ predicate in the logic (see line 11), and converts the $\ell_x \overset{1}{\hookrightarrow}_{\mathsf{std}}$ and $\ell_y \overset{1}{\hookrightarrow}_{\mathsf{std}}$ predicates to process points-to predicates, $\ell_x \overset{1}{\hookrightarrow}_{\mathsf{proc}}$ and $\ell_y \overset{1}{\hookrightarrow}_{\mathsf{proc}}$, respectively. This conversion ensures that ℓ_x and ℓ_y can only be written to in the context of action blocks, since the $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$ predicates never provide write access, unless they are first upgraded to $\overset{1}{\hookrightarrow}_{\mathsf{act}}$.

On line 13, the executing thread is required to hand-in all ownerships that are to be protected by the lock, as specified by the lock invariant on line 3, named *lock*. From that point on these resources can only be obtained in the context of an **atomic** program, until after the invariant stops protecting them on line 27.

Line 15 shows that the GCD process can be replaced by its definition, $\mathsf{Tx} \parallel \mathsf{Ty}$, after which it can be split along $\parallel$ using PROC-SPLIT-MERGE; see line 16. The two resulting predicates, $\mathsf{Proc}_{\frac{1}{2}}(m, \mathsf{Tx})$ and $\mathsf{Proc}_{\frac{1}{2}}(m, \mathsf{Ty})$, can then be distributed over `thread_x` (line 17) and `thread_y` (line 19). When both these threads have been joined (on line 23), the two resulting $\mathsf{Proc}_{\frac{1}{2}}(m, \varepsilon)$ predicates can be merged back into a single $\mathsf{Proc}_1(m, \varepsilon \parallel \varepsilon)$ predicate using PROC-SPLIT-MERGE (see line 25), which is (bisimulation) equivalent to $\mathsf{Proc}_1(m, \varepsilon)$ (see line 26). We now have a full Proc_1 predicate (i.e., with a fractional permission of 1), with a fully executed process component (i.e., ε). This means that the HT-PROC-FINISH proof rule can be applied to finalise the abstraction, on line 29, to obtain the desired postcondition.

Figure 4.8 shows the intermediate steps of the proof of `thread_x` (the steps of `thread_y` are similar). As a precondition (on line 1), an abstract model of the form Tx is required, with a fractional permission of $\frac{1}{2}$. This model is maintained throughout execution of `thread_x` until the Boolean flag *stop* is set to false.

Inside the while loop, on line 13, the process Tx is unfolded into $\mathsf{decrx} \cdot \mathsf{Tx} + \mathsf{done}$. This allows the program to do the decrx action in case $[\ell_x] > [\ell_y]$ (on line 16), by using the HT-ACTION rule, so that afterwards, on line 20, the program can continue to execute as prescribed by Tx, and enter another loop iteration. However, in case $[\ell_x] = [\ell_y]$, the program can perform the done action on line 25, to indicate termination of the algorithm, and thereby reduce the process-algebraic model to ε. This causes the loop to terminate, and by line 7 allows ensuring the postcondition of having $\mathsf{Proc}_{\frac{1}{2}}(m, \varepsilon)$; see line 2.

4.6 Applications

In this section we apply our approach on three more verification challenges, namely:

(i) Verifying a concurrent program in which multiple threads increase a shared counter by one, in the style of the classical Owicki–Gries example (§4.6.1);

(ii) Verifying whether a fine-grained, non-reentrant lock implementation follows the intended locking protocol (§4.6.4); and

(iii) Verifying functional correctness of a classical leader election protocol, that is implemented on shared memory (§4.6.6)

In addition to these examples, some interesting variants on them are also discussed.

For (i) we verify the functional property that, after termination, the program has calculated the correct value. For (ii) we verify that clients of the lock adhere to the intended locking protocol, and thereby avoid misusing the lock. Finally, for (iii) we verify that, after termination of the leader election algorithm, the rightful leader had been announced, which is the participant with the highest initial value.

4.6.1 Concurrent Counting

Our second example considers a *concurrent counter*: a program where two threads concurrently increment a common shared integer stored on the heap. The basic algorithm is given in Figure 4.9. The goal is to verify that `program` increments the value at heap location $\ell_{counter}$ by two. However, providing a specification for `worker` can be difficult, since no guarantees to the value of $\ell_{counter}$ can be given after termination of `worker`, as it is used in a concurrent environment.

Existing verification approaches for this particular example [RPDYG15] mostly require auxiliary ghost state, some form of rely/guarantee reasoning, or, more recently, concurrent abstract predicates, which may blow-up the amount of required specifications and are not always easy to use. We show how to verify the program of Figure 4.9 via our model-based abstraction approach. Later, we show how our techniques may be used on the same program but generalised to n threads.

Our approach is to protect all changes to the heap at location $\ell_{counter}$ by a process that we name parincr, using a process-algebraic variable named *counter*. The parincr process is defined as the parallel composition incr $\|$ incr of two processes that both execute the incr action once. Performing incr has the effect of incrementing *counter* by one. From a process algebraic point of view it is easy to see that parincr satisfies its contract: every possible trace of parincr indeed has the effect of increasing *counter* by two, and this can automatically be verified.

```
1  /* The location to the heap entry that stores the counter. */
2  int ref ℓ_counter;
3
4  /* Implementation of workers, which atomically increment the counter. */
5  void worker()
6  |  atomic lock do
7  |  |  [ℓ_counter] := [ℓ_counter] + 1;
8  |  end
9  end
10
11 /* Startup procedure of the concurrent counting example program. */
12 void program(int n)
13 |  [ℓ_counter] := n;
14 |  ref t1 := fork worker();
15 |  ref t2 := fork worker();
16 |  join t1;
17 |  join t2;
18 end
```

Figure 4.9: The concurrent counting example program, where two threads forked by `program` increment the shared integer *counter*.

We use this result in the verification of `program` by using model-based reasoning. In particular, we may instantiate parincr as a model m, split along its parallel composition, and give each forked thread a fraction of the splitted Proc predicate. The interface specification of the `worker` procedure thus becomes (for any π):

$$\{\, \mathsf{Proc}_\pi(m, \mathsf{incr}) \,\}\ \texttt{worker}(m)\ \{\, \mathsf{Proc}_\pi(m, \varepsilon) \,\}$$

An annotated version of the concurrent counting program is presented in Figure 4.10. Lines 2–7 define the process-algebraic model parincr, and line 12 defines the lock invariant that is needed by the **atomic** block on lines 17–21, for obtaining the right to be able to write to $\ell_{counter}$ in the context of an action block.

Indeed, by deductively showing that both threads execute the incr action, the established result of incrementing $\ell_{counter}$ by 2 can be concluded, on line 25.

4.6.2 Generalised Concurrent Counting

The interface specification of worker is generic enough to allow a generalisation to n threads. Instead of the parincr process as presented in Figure 4.10, one could

```
1  /* Action that models the atomic increment of the counter. */
2  effect counter = \old(counter) + 1;
3  action incr;
4
5  /* Abstract behavioural specification of the concurrent counter. */
6  ensures counter = \old(counter) + 2;
7  process parincr := incr ‖ incr;
8
9  /* Location to the heap entry that stores the counter. */
10 int ref ℓ_counter;
11
12 lock_invariant lock := ℓ_counter ↪¹_proc −;
13
14 requires Proc_π(m, incr);
15 ensures Proc_π(m, ε);
16 void worker(ref m)
17   atomic lock do
18     action m.incr do
19       [ℓ_counter] := [ℓ_counter] + 1;
20     end
21   end
22 end
23
24 requires ℓ_counter ↪¹ −;
25 ensures ℓ_counter ↪¹ n + 2;
26 void program(int n)
27   [ℓ_counter] := n;
28   ref m := process parincr over ⟨counter ↦ ℓ_counter⟩;
29   invariant lock do
30     ref t1 := fork worker(m) with π = 1/2;
31     ref t2 := fork worker(m) with π = 1/2;
32     join t1;
33     join t2;
34   end
35   finish m;
36 end
```

Figure 4.10: Definition of the parincr process that models two concurrent threads performing an incr action, and the required annotations for `worker` and `program`.

consider the following process, which essentially encodes the process "incr ‖ ⋯ ‖ incr" (n times) via recursion:

```
1 requires n ≥ 0;
2 ensures counter = \old(counter) + n;
3 process parincr(int n) := if n > 0 then incr ‖ parincr(n − 1) else ε;
```

Figure 4.11 shows the generalised version of the concurrent counting program, including all intermediate proof steps, in which we reuse the incr action and the `worker` procedure from Figure 4.10. Here `program` takes an extra parameter n that determines the number of threads to be spawned. The `spawn` procedure has been added to spawn the n threads This procedure is recursive to match the recursive definition of the parincr(n) process. Again, each thread executes the `worker` procedure. We verify that after running `program` the value of *counter* has increased by n.

On the level of processes we may automatically verify that each trace of the process parincr(n) is a sequence of n consecutive incr actions. As a consequence, from the effects of incr we can verify that parincr(n) increases *counter* by n. On the program level we may verify that `spawn`(m, n) fully executes according to the parincr(n) process. To clarify, before the fork on line 10 the definition of parincr(n) can be unfolded to incr ‖ parincr($n-1$) (see line 8) and can then be split along its parallel composition (see line 9). Then the forked thread receives incr and the recursive call to `spawn` receives parincr($n-1$). After calling **join** on line 14, both the call to `worker` and the recursive call to `spawn` have ensured completing the process they received (see line 15), thereby leaving the process $\varepsilon \parallel \varepsilon$ (see line 16), which can be rewritten to ε (see line 17) to satisfy `spawn`'s postcondition. As a result, after calling **finish** on line 33 we can successfully verify that the value stored on the heap at location $\ell_{counter}$ has indeed been increased by n.

4.6.3 Unequal Concurrent Counting

One could consider an interesting variant on the two-threaded concurrent counting problem: one thread performing the assignment "$[\ell_{counter}] = [\ell_{counter}] + v$" for some integer value v, and the other thread concurrently performing "$[\ell_{counter}] = [\ell_{counter}] * v$". Starting from a state where $\ell_{counter} \overset{1}{\hookrightarrow} c$ holds for some integer c, the challenge is to verify whether $\ell_{counter} \overset{1}{\hookrightarrow} (c + v) * v \vee (c * v) + v$ holds after termination of the program, in a thread-modular manner.

This program can be verified using our model-based approach (without requiring for example auxiliary state) by defining corresponding actions for the two different assignments. Figure 4.12 shows how the global model is described. The action

```
1  requires n ≥ 0;
2  requires Proc_π(m, parincr(n));
3  ensures Proc_π(m, ε);
4  void spawn(ref m, int n)
5  |  if (n > 0) then
6  |  |  {Proc_π(m, parincr(n))}
7  |  |  {Proc_π(m, if n > 0 then incr ∥ parincr(n − 1) else ε)}
8  |  |  {Proc_π(m, incr ∥ parincr(n − 1))}
9  |  |  {Proc_{π/2}(m, incr) ∗ Proc_{π/2}(m, parincr(n − 1))}
10 |  |  ref t := fork worker(m) with π = π/2;
11 |  |  {Proc_{π/2}(m, parincr(n − 1))}
12 |  |  spawn(m, n − 1) with π = π/2;
13 |  |  {Proc_{π/2}(m, ε)}
14 |  |  join t;
15 |  |  {Proc_{π/2}(m, ε) ∗ Proc_{π/2}(m, ε)}
16 |  |  {Proc_π(m, ε ∥ ε)}
17 |  |  {Proc_π(m, ε)}
18 |  end
19 end
20
21 requires ℓ_counter ↪[1] − ∗ n ≥ 0;
22 ensures ℓ_counter ↪[1] c + n;
23 void program(int c, int n)
24 |  [ℓ_counter] := c;
25 |  ref m := process parincr(n) over ⟨counter ↦ ℓ_counter⟩;
26 |  {Proc_1(m, parincr(n)) ∗ ℓ_counter ↪[1]_proc c}
27 |  invariant lock do
28 |  |  {Proc_1(m, parincr(n))}
29 |  |  spawn(m, n) with π = 1;
30 |  |  {Proc_1(m, ε)}
31 |  end
32 |  {Proc_1(m, ε) ∗ ℓ_counter ↪[1]_proc −}
33 |  finish m;
34 |  {ℓ_counter ↪[1] c + n}
35 end
```

Figure 4.11: Generalisation of the concurrent counting verification problem, where `program` forks n threads using the recursive `spawn` procedure. Each thread executes the `worker` procedure, therewith incrementing the value at $\ell_{counter}$ by one.

```
1  /* Action that represents the increment of ℓ_counter in the program. */
2  effect counter = \old(counter) + n;
3  action incr(int n);
4
5  /* Action that represents the multiplication of ℓ_counter in the program. */
6  effect counter = \old(counter) * n;
7  action mult(int n);
8
9  /* Abstract behavioural description of unequal concurrent counting. */
10 ensures counter = \old(counter) * n + n ∨
11         counter = (\old(counter) + n) * n;
12 process count(int n) := plus(n) ‖ mult(n);
```

Figure 4.12: The abstract model for the unequal concurrent counting problem.

plus(n) has the effect of incrementing *counter* by n, while mult(n) has the effect of multiplying *counter* by n. The required program annotations are then similar to the ones used in Figure 4.11.

All three variants on the concurrent counting problem have mechanically been verified using VerCors.

4.6.4 Lock Specification

The third example demonstrates how our approach can be used to verify control-flow properties of programs, in this case the *compare-and-swap lock implementation* that is presented in the Concurrent Abstract Predicates (CAP) paper [DYDG+10]. The implementation is given in Figure 4.13. The cas(ℓ, c, v) operation is the *compare-and-swap* instruction, which atomically updates the value at location ℓ on the heap by v if the old value at ℓ is equal to c, otherwise $[\ell]$ is not changed. A Boolean result is returned that indicates whether the update to ℓ was successful.

In particular, model-based reasoning is used to verify that the clients of this lock adhere to the intended locking protocol: clients may only successfully acquire the lock when the lock was unlocked and vice versa. Stated differently, we verify that clients may not acquire (nor release) the same lock successively.

The process algebraic description of the locking protocol is a composition of two actions, named acq and rel, that model the process of acquiring and releasing the lock, respectively. A third action named done is used to indicate that the lock is no longer used and can thus be destroyed. We use this process as a model to protect changes to a shared Boolean flag that is stored on the heap, at location

```
1  /* Reference to a Boolean flag on the heap */
2  boolean ref ℓ_flag;
3
4  /* Simple implementation of lock acquiring */
5  void acquire()
6  |  boolean b := false;
7  |  while (¬b) do
8  |  |  b := cas(ℓ_flag, false, true);
9  |  end
10 end
11
12 /* Simple implementation of lock releasing */
13 void release()
14 |  atomic lock do
15 |  |  [ℓ_flag] := false;
16 |  end
17 end
```

Figure 4.13: Implementation of a simple locking system.

ℓ_{flag}, so that all changes to ℓ_{flag} must either happen as an acq or as a rel action. The acq action may be performed only if $[\ell_{flag}]$ is currently false and has the effect of setting the flag to true. The rel action simply has the effect of setting $[\ell_{flag}]$ to false, whatever the current value at ℓ_{flag} (therefore rel does not need a guard).

The locking protocol is given in Figure 4.14, and is defined by the processes $\mathsf{Locked} := \mathsf{rel} \cdot \mathsf{Unlocked}$ and $\mathsf{Unlocked} := \mathsf{acq} \cdot \mathsf{Locked} + \mathsf{done}$. This allows using the following interface specifications for the `acquire` and `release` procedures (with any π, and with m a global identifier of an initialised model):

$$\{\, \mathsf{Proc}_\pi(m, \mathsf{Unlocked}) \,\}\ \texttt{acquire}(m)\ \{\, \mathsf{Proc}_\pi(m, \mathsf{Locked}) \,\}$$
$$\{\, \mathsf{Proc}_\pi(m, \mathsf{Locked}) \,\}\ \texttt{release}(m)\ \{\, \mathsf{Proc}_\pi(m, \mathsf{Unlocked}) \,\}$$

Specification-wise, clients of the lock may only perform `acquire` when they have a corresponding process predicate that is in an "Unlocked" state (and the same holds for `release` and "Locked"), thereby enforcing the locking protocol (i.e., the process only allows traces of the form: $\mathsf{acq}, \mathsf{rel}, \mathsf{acq}, \mathsf{rel}, \cdots$). The `acquire` procedure performs the acq action via the `cas` operation: one may define `cas` to update ℓ_{flag} as an acq action. Moreover, since `cas` is an atomic operation, it can get all necessary ownership predicates from the resource invariant inv. Furthermore, calling `destroy`() corresponds to performing the done action on the process algebra level,

```
1  /* Process-algebraic state */
2  shared int flag;
3
4  /* Action that models lock acquiring. */
5  guard ¬flag;
6  effect flag;
7  action acq;
8
9  /* Action that models lock releasing. */
10 effect ¬flag;
11 action rel;
12
13 /* Action that indicates termination of the locking protocol. */
14 action done;
15
16 /* Processes that model the locking protocol. */
17 process Unlocked := acq · Locked + done;
18 process Locked := rel · Unlocked;
```

Figure 4.14: Process-algebraic specification of the locking protocol.

which may only be done in the "Unlocked" state.

The full annotated lock implementation is presented in Figure 4.15 and Figure 4.16. The `init` and `destroy` procedures have been added to initialise and finalise the lock and thereby to create and destroy the corresponding model. The `init` consumes write permission to ℓ_{flag} (line 38), initialises the model (line 44), and transfers the converted write permission into the resource invariant *lock* (on line 46). Both the atomic block (lines 25–34) and the `cas` operation (on line 15) make use of *lock* to get permission to change the value at ℓ_{flag} in an action block. The `cas` operation on line 15 performs the acq action internally, depending on the success of the compare-and-swap (indicated by its return value). This is reflected upon in the loop invariant. The `destroy` procedure has the opposite effect of `init`: it consumes the (full) Proc predicate (in state "Unlocked"), destroys the model and the resource invariant, and gives back the converted write permission to ℓ_{flag}.

In the current presentation, `init` returns a single Proc predicate in state Unlocked, thereby allowing only a single client. This is however not a limitation: to support two clients, `init` could alternatively initialise and ensure the Unlocked ‖ Unlocked process. Furthermore, to support n clients (or a dynamic number of clients), `init` could apply a construction similar to the one used in the generalised concurrent counting example (see Section 4.6.2).

```
1  ref boolean ℓ_flag;
2
3  lock_invariant lock := ℓ_flag ↪¹_proc −;
4
5  /* Simple (annotated) implementation of lock acquiring. */
6  requires Proc_π(m, Unlocked);
7  ensures Proc_π(m, Locked);
8  void acquire(ref m)
9  |  boolean b := false;
10 |
11 |  loop_invariant ¬b ⟹ Proc_π(m, acq · Locked);
12 |  loop_invariant b ⟹ Proc_π(m, Locked);
13 |  while (¬b) do
14 |  |  {Proc_π(m, acq · Locked)}
15 |  |  b := cas_acq(ℓ_flag, false, true);
16 |  |  {(b ∧ Proc_π(m, Locked)) ∨ (¬b ∧ Proc_π(m, acq · Locked))}
17 |  end
18 end
19
20 /* Simple (annotated) implementation of lock releasing. */
21 requires Proc_π(m, Locked);
22 ensures Proc_π(m, Unlocked);
23 void release(ref m)
24 |  {Proc_π(m, Locked)}
25 |  atomic lock do
26 |  |  {Proc_π(m, Locked) ∗ ℓ_flag ↪¹_proc −}
27 |  |  {Proc_π(m, rel · Unlocked) ∗ ℓ_flag ↪¹_proc −}
28 |  |  action m.rel do
29 |  |  |  {ℓ_flag ↪¹_act −}
30 |  |  |  [ℓ_flag] := false;
31 |  |  |  {ℓ_flag ↪¹_act false}
32 |  |  end
33 |  |  {Proc_π(m, Unlocked) ∗ ℓ_flag ↪¹_proc false}
34 |  end
35 |  {Proc_π(m, Unlocked)}
36 end
```

Figure 4.15: Annotated implementations of the `acquire` and `release` procedures.

```
/* Simple (annotated) implementation for lock initialisation. */
requires ℓ_flag ↪[1] −;
ensures Proc_1(\result, Unlocked);
ref init()
  {ℓ_flag ↪[1] −}
  [ℓ_flag] := false;
  {ℓ_flag ↪[1]_std false}
  ref m := process Unlocked over ⟨flag ↦ ℓ_flag⟩;
  {ℓ_flag ↪[1]_proc false ∗ Proc_1(m, Unlocked)}
  init lock lock;
  {Proc_1(m, Unlocked)}
  return m;
end

/* Simple (annotated) implementation for lock finalisation. */
requires Proc_1(m, Unlocked);
ensures ℓ_flag ↪[1] −;
void destroy(ref m)
  {Proc_1(m, Unlocked)}
  {Proc_1(m, acq · Locked + done)}
  action do m.done end
  {Proc_1(m, ε)}
  destroy lock lock;
  {ℓ_flag ↪[1]_proc − ∗ Proc_1(m, ε)}
  finish m;
  {ℓ_flag ↪[1] −}
end
```

Figure 4.16: Procedures for initialising (`init`) and finalising (`destroy`) the lock.

4.6.5 Reentrant Locking

The process algebraic description of the locking protocol can be upgraded to describe a *reentrant lock*: a locking system where clients may `acquire` and `release` multiple times in succession. A reentrant lock that is acquired n times by a client must also be released n times before it is available to other clients. Instead of using the Locked and Unlocked processes, the reentrant locking protocol is described by the following process (with $n \geq 0$):

$$\textbf{process}\ \mathsf{Lock}(\textbf{int}\ n) := \mathsf{acq} \cdot \mathsf{Lock}(n+1) + \textbf{if}\ n > 0\ \textbf{then}\ \mathsf{rel} \cdot \mathsf{Lock}(n-1)$$

Rather than describing the lock state as a Boolean flag, like we have done in the single-entrant locking example, the state of the reentrant lock can be described as a *multiset* (*bag*) containing thread identifiers. In that case, acq and rel protect all changes made to the multiset in order to enforce the locking protocol described by Lock. The interface specifications of `acquire` and `release` then become:

$$\{\, \mathsf{Proc}_\pi(m, \mathsf{Lock}(n)) \,\}\ \texttt{acquire}(m)\ \{\, \mathsf{Proc}_\pi(m, \mathsf{Lock}(n+1)) \,\}$$
$$\{\, 0 < n * \mathsf{Proc}_\pi(m, \mathsf{Lock}(n)) \,\}\ \texttt{release}(m)\ \{\, 0 \leq n * \mathsf{Proc}_\pi(m, \mathsf{Lock}(n-1)) \,\}$$

Moreover, the Lock(n) process could be extended with a done action to allow the reentrant lock to be destroyed, like shown in the previous example. The done action should then only be allowed when $n = 0$. Both the simple locking implementation and the reentrant locking implementation have been automatically verified using the VerCors verifier.

4.6.6 Verifying a Leader Election Protocol

This section demonstrates our verification approach on a bigger case, that involves verifying the correctness of a classical distributed algorithm, namely a leader election protocol [OBH16]. The algorithm is performed by N distributed workers that are organised in a ring, so that worker i only sends to worker $i + 1$ and only receives from worker $i - 1$, modulo N. The goal is to determine a leader among those workers. To find a leader, the election procedure assumes that each worker i receives a unique integer value v_i to start with, i.e., $v_i = v_j$ implies $i = j$, and then operates in N rounds. In every round, each worker sends the highest value it encountered so far to its right neighbour and in turn receives a value from its left neighbour, and remembers the highest of the two. The result after N rounds is that all workers know the highest unique value $\mathsf{max}\{v_0, \ldots, v_N\}$ in the network, allowing its original owner to announce itself as leader.

The case study has been verified with VerCors. Since VerCors does not yet have native support for message passing we first implemented basic asynchronous message passing functionality. This implementation consists of two non-blocking operations for standard communication: `mp_send(`r`,`msg`)` for sending a message msg to the worker with rank r[4], and $msg :=$ `mp_recv`(r) for receiving a message msg from worker r. The election protocol is implemented on top of this message passing system.

The main challenge of this case study is to define a system of message passing on the abstraction level that matches the implementation, using the techniques that have been presented so far. To design such a system we follow the ideas of [OBH16].

[4]The identifiers of workers are typically called *ranks* in message passing terminology.

First we define two process-algebraic actions, $\mathsf{send}(r, msg)$ and $\mathsf{recv}(r, msg)$, that abstractly describe the behaviour of the concrete implementations in `mp_send` and `mp_recv`, respectively. Secondly, to properly handle message receival on the abstraction level we also need to define *process-algebraic summation*. A summation operator $\Sigma_{x \in D} P$ allows quantifying over a (finite) set of data $D = \{d_0, \ldots, d_n\}$ and has the behaviour of $P[x/d_0] + \cdots + P[x/d_n]$. We use process-algebraic summation to quantify over the possible messages that `mp_recv` might receive.

The following two rules illustrate how the abstract send and recv actions are connected to `mp_send` and `mp_recv`.

$$\{\mathsf{Proc}_\pi(m, \mathsf{send}(r, msg) \cdot P + Q)\}\texttt{mp\_send}(m, r, msg)\{\mathsf{Proc}_{\pi'}(m, P)\}$$
$$\{\mathsf{Proc}_\pi(m, \Sigma_{x \in Msg}\mathsf{recv}(r, x) \cdot P + Q)\}msg := \texttt{mp\_recv}(m, r)\{\mathsf{Proc}_\pi(m, P[x/msg])\}$$

And last, we use send and recv to construct a process-algebraic model of the leader election protocol and verify that the implementation adheres to this model. This model has been analysed with mCRL2 to establish the global property that the correct leader is announced—the one with the highest initial value. From this it follows that the implementation determines the correct leader.

4.6.6.1 Behavioural Specification

Our main goal is proving that the implementation determines the correct leader upon termination. To prove this, we first define a *behavioural specification* of the election protocol that hides all irrelevant implementation details, and prove the correctness property on this specification. Process algebra provides the right abstraction level, as the behaviour of leader election can concisely be specified in terms of sequences of sends and receives. Figure 4.17 presents the process-algebraic specification. In particular, ParElect specifies the *global* behaviour of the protocol, whereas Elect specifies the *thread-local* behaviour. Ideally the send and recv actions would be part of a native message passing library. This is planned as future work.

The ParElect process encodes the parallel composition of the eligible participants. ParElect takes a sequence *vs* of initial values as argument, whose length equals the total number of workers due to its precondition. ParElect's postcondition states that *lead* must be a valid rank after termination and that $vs[lead]$ be the highest initial worker value. It follows that worker *lead* is the correctly chosen leader. We used mCRL2 to verify that ParElect is a correct abstract specification of the election protocol, with respect to its contract. The mCRL2 encoding can be found in the online Git repository that comes with this thesis.

The Elect process takes four arguments, namely:

1. The rank *rank* of the worker;

```
1  /* Process-algebraic state. */
2  shared seq⟨seq⟨Msg⟩⟩ chan; // communication channels between workers
3  shared int lead; // rank of the worker that is announced as leader
4
5  /* Action for message sending. */
6  guard 0 ≤ rank < |chan|;
7  effect chan[rank] = \old(chan[rank]) + {msg};
8  effect ∀r' : int . (0 ≤ r' < |chan| ∧ r' ≠ rank) ⟹
9           chan[r'] = \old(chan[r']);
10 action send(int rank, Msg msg);
11
12 /* Action for message receival. */
13 guard 0 ≤ rank < |chan|;
14 effect {msg} + chan[rank] = \old(chan[rank]);
15 effect ∀r' : int . (0 ≤ r' < |chan| ∧ r' ≠ rank) ⟹
16          chan[r'] = \old(chan[r']);
17 action recv(int rank, Msg msg);
18
19 /* Action for announcing a leader. */
20 guard 0 ≤ rank < |chan|;
21 effect lead = rank;
22 action announce(int rank);
23
24 /* Local behavioural specification of the election protocol. */
25 requires 0 ≤ n ≤ |chan|;
26 requires 0 ≤ rank < |chan|;
27 process Elect(int rank, Msg v0, Msg v, int n) :=
28   if 0 < n then send((rank + 1) % |chan|, v) ·
29     Σ_{v'∈Msg} recv(rank, v') · Elect(rank, v0, max(v, v'), n − 1)
30   else (if v = v0 then announce(rank) else ε);
31
32 /* Global behavioural specification of the election protocol. */
33 requires |vs| = |chan|;
34 requires ∀i, j : int . (0 ≤ i < |vs| ∧ 0 ≤ j < |vs| ∧ vs[i] = vs[j]) ⟹ i = j;
35 ensures |vs| = |chan| ∧ 0 ≤ lead < |vs|;
36 ensures ∀i : int . (0 ≤ i < |vs|) ⟹ vs[i] ≤ vs[lead];
37 process ParElect(seq⟨Msg⟩ vs) :=
38   Elect(0, vs[0], vs[0], |vs|) ‖ ··· ‖ Elect(|vs|−1, vs[|vs|−1], vs[|vs|−1], |vs|);
```

Figure 4.17: Behavioural specification of the leader election protocol.

2. The initial unique value v_0 of that worker;
3. The current highest value v encountered by that worker; and finally
4. The number n of remaining rounds.

The rounds are implemented via general recursion. In each round all workers send their current highest value v to their right neighbour (line 28), receive a value v' in return from their left neighbour (line 29), and continue with the highest of the two. The extra announce action is declared and used to announce the leader after n rounds. The postcondition of announce is that *lead* stores the leader's rank.

The contracts of send and recv describe the behaviour of standard non-blocking message passing. Communication on the specification level is implemented via *message queues*. Message queues are defined as sequences of messages, where messages are taken from a finite domain *Msg*. Since workers are organised in a ring in this case, every worker can do with only a single queue and the global communication channel architecture can be defined as a sequence of message queues: *chan* in the figure. The action contract of $\mathsf{send}(r, msg)$ expresses enqueuing the message *msg* onto the message queue $chan[r]$ of the worker with rank r. In more detail, the precondition of $\mathsf{send}(r, msg)$ expresses that r must be a valid rank in the network. Note that, since every worker receives one message queue we have that $|chan|$ is equal to the total number of workers. The postcondition of send is that *msg* has been enqueued onto $chan[r]$ and that the other queues, $chan[r']$ for any $r' \neq r$, have not been altered. Likewise, the contract of $\mathsf{recv}(r, msg)$ expresses dequeuing *msg* from $chan[r]$. Recall that the expression $\backslash\mathbf{old}(e)$ indicates that e is to be evaluated with respect to the pre-state of computation.

4.6.6.2 Protocol Implementation

Figure 4.18 presents the annotated implementation of the election protocol The `elect` method contains the code that is executed by every worker. The contract of `elect`$(m, rank, v_0, v, n)$ states that the method body adheres to the behavioural description $\mathsf{Elect}(rank, v_0, v, n)$ of the election protocol. The annotation **context** $\mathcal{P}$ is shorthand for **requires** $\mathcal{P}$; **ensures** $\mathcal{P}$. Each worker performing `elect` enters a for-loop that iterates n times, whose loop invariant states that, at iteration i, the remaining program behaves as prescribed by the process $\mathsf{Elect}(rank, v_0, v, n-i)$. The invocations to `mp_send` and `mp_recv` on lines 19 and 23 are annotated with **with** clauses that instantiate the free variables that occur in the contracts of `mp_send` and `mp_recv`. After n rounds, all workers with $v = v_0$ announce themselves as leader. However, since the initial values are chosen to be unique there can only be one such worker. Finally, we can verify that at the post-state of `elect` the abstract model has been fully executed (reduced to ε).

```
1  /* Shared state of the program (as references to the heap). */
2  seq⟨seq⟨Msg⟩⟩ ref ℓ_chan; // implementation of communication channels
3  int ref ℓ_nworkers; // total number of workers
4  int ref ℓ_lead; // rank of the leader to be announced
5
6  /* The information and ownership that is protected by the global lock. */
7  lock_invariant lock := ℓ_lead ↪¹_proc −∗
8     ∃xs : seq⟨seq⟨Msg⟩⟩ . ℓ_chan ↪¹_proc xs ∗ ℓ_nworkers ↪½_proc |xs|;
9
10 /* Implementation of the election protocol, for worker rank. */
11 context ℓ_nworkers ↪^π_proc n ∗ 0 ≤ rank < n;
12 requires Proc_π'(m, Elect(rank, v_0, v, n));
13 ensures Proc_π'(m, ε);
14 void elect(ref m, int rank, Msg v_0, Msg v, int n)
15    loop_invariant ℓ_nworkers ↪^π_proc n;
16    loop_invariant 0 ≤ i ≤ n;
17    loop_invariant Proc_π'(m, Elect(rank, v_0, v, n − i));
18    for (int i := 0 to n) do
19       mp_send(m, (rank + 1) % n, v) with {
20          P := Σ_{x∈Msg} recv(rank, x) · Elect(rank, v_0, max(v, x), n − i − 1),
21          Q := ε, π := π, π' := π'
22       };
23       Msg v' := mp_recv(m, rank) with {
24          P := Elect(rank, v_0, max(v, v'), n − i − 1), Q := ε, π := π, π' := π'
25       };
26       v := max(v, v');
27    end
28    if (v = v_0) then
29       action m.announce(rank) do
30          [ℓ_lead] := rank;
31       end
32    end
33 end
```

Figure 4.18: Annotated implementation of the leader election protocol.

Figure 4.19 presents bootstrapping code for the message passing implementation. The `main` function initialises the communication channels whereas `parelect` spawns all worker threads. `main(`vs`)` additionally initialises and finalises the abstraction ParElect(vs) on the specification level (line 24 and 28, respectively), whose

```
1  /* Spawns and joins |vs| parallel workers that all execute elect. */
2  context ℓ_nworkers ↪[1/2]_proc |vs| * 0 < |vs|;
3  requires Proc_1(m, ParElect(vs));
4  ensures Proc_1(m, ε);
5  void parelect(ref m, seq⟨Msg⟩ vs)
6  |  context 0 ≤ rank < |vs|;
7  |  requires Proc_{1/|vs|}(m, Elect(rank, vs[rank], vs[rank], |vs|));
8  |  ensures Proc_{1/|vs|}(m, ε);
9  |  par (int rank := 0 to |vs|) do
10 |  |  elect(m, vs[rank], vs[rank], |vs|) with {
11 |  |    π := 1/(4|vs|), π' := 1/|vs|
12 |  |  };
13 |  end
14 end
15
16 /* Startup procedure for the leader election protocol. */
17 context ℓ_nworkers ↪[1] − * ℓ_chan ↪[1] − * ℓ_lead ↪[1] −;
18 requires ∀i, j : int . (0 ≤ i < |vs| ∧ 0 ≤ j < |vs| ∧ vs[i] = vs[j]) ⟹ i = j;
19 ensures 0 ≤ \result < |vs|;
20 ensures ∀i : int . (0 ≤ i < |vs|) ⟹ vs[i] ≤ vs[\result];
21 int main(seq⟨Msg⟩ vs)
22 |  [ℓ_nworkers] := |vs|;
23 |  [ℓ_chan] := initialiseChannels(|vs|);
24 |  ref m := process ParElect(vs) over ⟨chan ↦ ℓ_chan, lead ↦ ℓ_nworkers⟩;
25 |  invariant lock do
26 |  |  parelect(m, vs);
27 |  end
28 |  finish m;
29 |  return [ℓ_lead];
30 end
```

Figure 4.19: Bootstrap procedures of the leader election protocol.

analysis allows establishing the postconditions of `main`. The function `parelect`(m, vs) implements the abstract model ParElect(vs) by spawning $|vs|$ workers that all execute the `elect` program. The contract associated to the parallel block (lines 6–8) is called an *iteration contract* and assigns pre- and postconditions to every parallel instance. For more details on iteration contracts we refer to [BDH15]. Most importantly, the iteration contract of each parallel worker states (on line 7) that the worker behaves as specified by Elect. Thus, we deductively verify in a *com-*

positional and *thread-modular* way that the program implements its behavioural specification. Lastly, all the required ownership for the global fields and the Proc_1 predicate is split and distributed among the individual workers via the iteration contract and the **with** clause on lines 10–11.

4.6.6.3 Communication Primitives

Finally, Figure 4.20 presents the implementation of the message passing system. The `mp_send`(m, *rank*, *msg*) method implements the operation of enqueuing *msg* onto the message queue of worker *rank*. The contract of `mp_send` expresses that the enqueuing operation is encapsulated as a send(*rank*, *msg*) action that is prescribed by an abstract model identified by m. The program performs the send action by means of an action block that updates ℓ_{chan} by enqueuing *msg*. The result is that send has been performed in the post-state of `mp_send` (see line 4). In order for m to be able to perform the send action, all send's preconditions have to be satisfied. For this purpose line 2 requires that $\ell_{nworkers}$ is a reference to some integer n that represents the total number of workers, and that *rank* is between 0 and n.

The `mp_recv`(m, *rank*) function implements the operation of dequeuing and returns the first message in the message queue of worker *rank*. The receive is prescribed as the recv action on the abstraction level, where the potential received message is ranged over by the summation on line 15.

The implementation of `mp_recv`(m, *rank*) simply checks in a busy-loop whether $\ell_{chan}[rank]$ contains a message, and if so, pops the first available message of $\ell_{chan}[rank]$ as a recv action. This message will eventually be returned on line 34. The resulting abstraction after termination of `mp_recv`, as by line 4 is the trailing process P with the quantified variable x substituted for the returned message.

4.6.7 Other Verification Examples

This section demonstrated the use of process algebraic models in multiple different verification examples, as well as some interesting variants on them. We showed how model-based reasoning can be used as a practical tool to verify different types of properties that would otherwise be hard to verify, especially with an automated tool. We considered *data properties* in the parallel GCD and the concurrent counting examples, and considered *control-flow* properties in the locking examples. Moreover, we showed how to use the model-based reasoning approach in environments with a dynamic number of concurrent threads.

Our approach can also be used to reason about non-terminating programs. Notably, a *no-send-after-read* verification example is available that addresses a commonly used security property: if confidential data is received by a secure device, it

```
1  /* Operation for sending a message msg to worker rank. */
2  context ∃n . ℓ_nworkers ↪^π_proc n * 0 ≤ rank < n;
3  requires Proc_π'(m, send(rank, msg) · P + Q);
4  ensures Proc_π'(m, P);
5  void mp_send(ref m, int rank, Msg msg)
6    atomic lock do
7      action m.send(rank, msg) do
8        ℓ_chan[rank] := ℓ_chan[rank] + {msg};
9      end
10   end
11 end
12
13 /* Operation for receiving a message from worker rank. */
14 context ∃n . ℓ_nworkers ↪^π_proc n * 0 ≤ rank < n;
15 requires Proc_π'(m, Σ_{x∈Msg} recv(rank, x) · P + Q);
16 ensures Proc_π'(m, P[x/\result]);
17 Msg mp_recv(ref m, int rank)
18   boolean stop := false;
19   Msg msg;
20
21   loop_invariant ¬stop ⟹ Proc_π'(m, Σ_{x∈Msg} recv(rank, x) · P + Q);
22   loop_invariant stop ⟹ Proc_π'(m, P[x/msg]);
23   while (¬stop) do
24     atomic lock do
25       if (0 < |ℓ_chan[rank]|) then
26         msg := head(ℓ_chan[rank]);
27         action m.recv(rank, msg) do
28           ℓ_chan[rank] := tail(ℓ_chan[rank]);
29           stop := true;
30         end
31       end
32     end
33   end
34   return msg;
35 end
```

Figure 4.20: A basic implementation of message passing, whose behaviour is specified in terms of the send and recv actions that were defined on page 118.

will not be passed on. The concrete send- and receive behaviour of the device can be abstracted by send and recv actions, respectively. Receiving confidential information is modelled as the clear action. Essentially, we show that after performing a clear action the device can no longer perform send's.

4.7 Related Work

The abstraction technique proposed in this chapter allows reasoning about functional behaviour of concurrent, possibly non-terminating programs. A related approach is (impredicative) Concurrent Abstract Predicates (CAP) [DYDG$^+$10, SB14], which also builds on CSL with permissions. In the program logic of CAP, *regions* of memory can be specified as being *shared*. Threads must have a consistent view of all shared regions: all changes must be specified as *actions* and all shared regions are equipped with a set of possible actions over their memory. Our approach uses process algebraic abstractions over shared memory in contrast to the shared regions of CAP, so that all changes to the shared memory must be captured as process algebraic actions. We mainly distinguish in the use of *process algebraic reasoning* to verify properties that could otherwise be hard to verify, and in the capability of doing this mechanically by providing tool support.

Other related approaches include TaDA [RPDYG14], a program logic that builds on CAP by adding a notion of *abstract atomicity* via Hoare triples for atomic operations. CaReSL [TDB13] uses a notion of shared regions similar to CAP, but uses *tokens* to denote ownership. These tokens are used to transfer ownership over resources between threads. Iris [JSS$^+$15, KJB$^+$17] is a reasoning framework that aims to provide a comprehensive and simplified solution for recent (higher-order) concurrency logics. Sergey et al. [SNB15b] propose *time-stamped histories* to capture modifications to the shared state. Our approach may both capture and model program behaviour and benefits from extensive research on process algebraic reasoning [GM14]. Moreover, the authors provide a *mechanised* approach to interactively verify full functional correctness of concurrent programs by building on CSL [SNB15a]. Popeea and Rybalchenko [PR12] combine abstraction refinement with rely-guarantee reasoning to verify termination of multi-threaded programs.

In the context of verifying distributed systems, Session Types [HYC08] describe communication protocols between processes [HMM$^+$12]. However, our approach is more general as it allows describing any kind of behaviour, including communication behaviour between different system components.

4.8 Conclusion

This chapter addresses thread-modular verification of possibly non-terminating concurrent programs by proposing a technique to abstract program behaviour using process algebras. A key characteristic of our approach is that properties about programs can be proven by analysing process algebraic abstractions and by verifying that programs do not deviate from these abstractions. The verification is done in a thread-modular way, using an abstraction-aware extension of CSL. This chapter demonstrates how the proposed technique provides an elegant solution to various verification problems that may be challenging for alternative verification approaches. In addition, we contribute tool support and thereby allow mechanised verification of the presented examples.

In Chapter 5, we present and discuss a full formalisation and soundness proof of the abstraction approach introduced in this chapter, which have been encoded in the Coq proof assistant, and mechanically been proven.

Later, in Chapter 7, we extend our abstraction approach for the distributed case, by having the actions abstract message passing behaviour, instead of shared-memory behaviour via action contracts.

4.8.1 Future Directions

At the moment, verification at the process algebra level is non-modular: in order to analyse process-algebraic abstractions, they first have to be linearised before they can be analysed by, for example, VerCors or mCRL2. It would, however, be desirable to be able to analyse the parallel components in process-algebraic models in isolation, and combine their analyses into a analysis of the global process. We plan to achieve this by investigating how our approach combines with rely-guarantee reasoning [Jon83] and deny-guarantee reasoning [DFPV09]. We might, for example, be able to extend the contracts of processes and actions with *rely* and *guarantee* clauses, that express the assumptions and contributions of the process/action to the global property of interest, respectively.

Moreover, currently the proof system allows the results of process-algebraic analysis to be used only when the process has been fully executed, using the **finish** command. We plan to investigate how (partial) results can be obtained and used at *intermediate* points of program execution. This would, for example, be beneficial for non-terminating programs, e.g., programs that execute an event/message loop. (We have already implemented this for message passing programs and abstractions, in Chapter 7, but not yet for shared-memory abstractions.)

Finally, we plan to investigate how and to what extent process-algebraic models

can be extracted from annotated source code. The current approach requires to annotate program code with **action** annotations, to indicate the relation between abstract actions and concrete program instructions. However, these annotations may give an opportunity to extract models from program code, and thereby to perform some automated analysis of the program's control-flow.

CHAPTER 5

Soundness of Shared-Memory Program Abstractions

Abstract

This chapter presents and discusses a full formalisation of the abstraction-based verification approach introduced in Chapter 4. The formalisation consists primarily of a program logic that extends CSL, as used in Chapter 4, and a soundness proof for this logic. Moreover, the formalisation and soundness proof have been fully encoded in the Coq proof assistant, and are machine-checked, to increase the confidence of their correctness.

5.1 Introduction

In the previous chapter, we introduced an abstraction technique for verifying behavioural properties of concurrent programs in a practical manner, and demonstrated it on various examples. The key idea of this approach is that program behaviour is abstractly specified as a *process-algebraic model*. It is known that process algebra provides a language for modelling and reasoning about the behaviour of concurrent programs at a suitable level of abstraction [ABC10]. Process algebra offers an abstract, mathematically elegant way of expressing program behaviour. In contrast, the behaviour of a *real* concurrent programming language with shared memory, threads and locks, has far less algebraic behaviour. This makes process algebra a suitable language for *specifying* program behaviour. Such a specification can be seen as a model, the properties of which can additionally be checked (say, by model checking against temporal logic formulas, which can be seen as even more abstract behavioural specifications). The main difficulty is presented by the typical abstraction gap between program implementations and their models. The unique contribution of the approach is that it bridges this gap by providing a deductive technique for formally linking programs with their process-algebraic

models. These formal links preserve *safety* properties; we leave the preservation of liveness properties for future work.

The uniqueness of the approach rests in the use of concurrent separation logic (CSL) to reason not only about data races and memory safety, which is standard, but also about process-algebraic models (i.e., specified program behaviours), viewing the latter as *resources* that can be split and consumed. This results in a modular and compositional approach to establish that a program behaves as specified by its abstract model.

In this chapter, we give a formal justification of the abstraction approach introduced in Chapter 4. In particular, we define the program logic sketched in Section 4.4 in more detail, and prove soundness of this logic, with respect to an operational semantics of programs and process-algebraic models. In addition, this formalisation has fully been encoded in the Coq proof assistant. This Coq encoding includes a machine-checked soundness proof of the program logic, stating that any verified program adheres to its behavioural specification—its abstract model.

5.1.1 Contributions

This chapter makes the following two main contributions[1]:

1. *Theoretical justification* of the verification approach that we introduced in Chapter 4, presented as a program logic that extends CSL [Vaf11]. This chapter complements the previous one (i.e., [OBG+17]), which essentially contributes tool support and gives a more practical overview of the verification technique.
2. A *Coq formalisation* of the theory, containing machine-checked proofs of all theorems presented in this chapter, including soundness of the program logic.

5.1.2 Chapter Outline

The remainder of this chapter is organised as follows. First, Section 5.2 defines the syntax and semantics of the process algebra language for program abstractions, after which Section 5.3 defines the syntax and semantics of programs. Then Section 5.4 defines the assertion language and its semantics. Section 5.5 discusses the proof rules of the program logic and Section 5.6 discusses their soundness. Then, Section 5.7 elaborates on the implementation of the verification approach in VerCors and on the Coq development of the meta-theory. Finally, Section 5.8 gives related work and Section 5.9 concludes, and gives directions for future work.

[1] This chapter is based on the article [OGH20].

5.2 Process-Algebraic Models

This section defines the syntax and semantics of process-algebraic abstractions of program behaviour.

5.2.1 Syntax

In this work, program abstractions are defined using the following ACP-style [BK84] process-algebraic specification language, where $a, b, \cdots \in \mathit{Act}$ are *actions*.

Definition 5.2.1 (Processes)**.**

$$P, Q, \cdots \in \mathit{Proc} \ ::= \ \varepsilon \mid \delta \mid a \mid P \cdot Q \mid P + Q \mid P \parallel Q \mid P \lfloor\!\lfloor Q \mid P^*$$

Clarifying the different connectives and constructs; ε is the *empty process*, which has no behaviour and terminates successfully. The δ process, on the other hand, is the *deadlocked process*, which neither progresses nor terminates. Processes of the form a are *actions*, which model the basic, observable (shared-memory) system behaviours. The process $P \cdot Q$ is the *sequential composition* of P and Q, whereas $P + Q$ is their non-deterministic *choice*. The *parallel composition* of processes P and Q is written $P \parallel Q$. The process $P \lfloor\!\lfloor Q$ is the *left-merge* of P and Q, which is similar in spirit to parallel composition, however $\lfloor\!\lfloor$ insists that the left-most process P proceeds first. The left-merge is an auxiliary connective commonly used to axiomatise parallel composition [Mol90], by having $P \parallel Q = P \lfloor\!\lfloor Q + Q \lfloor\!\lfloor P$. Finally, P^* is the *repetition*, or *iteration*, of P, and denotes a sequence of zero or more P's.

Action contracts. Our verification approach uses processes in the presence of *data*, which is implemented via *action contracts*. Action contracts consist of pre- and postconditions that logically describe the state changes that are imposed by the corresponding action. These action contracts are defined by the following expression language, where $x, y, z, \cdots \in \mathit{ProcVar}$ are *process variables*, and $m, n, \cdots \in \mathit{Lit}$ are *literal constants*.

Definition 5.2.2 (Process expressions, Process conditions)**.**

$$\begin{aligned} e \in \mathit{ProcExpr} \ &::= \ m \mid x \mid e + e \mid e - e \mid \ \cdots \\ b \in \mathit{ProcCond} \ &::= \ \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \wedge b \mid e = e \mid e < e \mid \ \cdots \end{aligned}$$

In the remainder of this chapter, each action is assumed to have an associated action contract assigned to it. We use the functions $\mathsf{pre}, \mathsf{post} : \mathit{Act} \to \mathit{ProcCond}$

for obtaining the precondition and postcondition of an action, respectively, that together constitute the action's contract. Both these conditions are of type *ProcCond*, which is the domain of Boolean expressions over process-algebraic variables.

Free variables and substitution. We write $\mathsf{fv}(e)$ and $\mathsf{fv}(b)$ to denote the set of process-algebraic variables that occur freely in the expression e or condition b, respectively. Moreover, $e[x/e']$ denotes the substitution of x for e' inside e; and likewise for $b[x/e]$. The definitions of $\mathsf{fv}(\cdot)$ and substitution are standard, and therefore deferred to Appendix A.

5.2.2 Operational Semantics

The denotational semantics of process expressions $[\![e]\!]\sigma$ and conditions $[\![b]\!]\sigma$ is defined in the standard way: as total functions that evaluate to *Val*. Furthermore, *process stores*, $\sigma \in \mathit{ProcStore} \triangleq \mathit{ProcVar} \to \mathit{Val}$, are used to give an interpretation to all process-algebraic variables. The exact definition of the denotational semantics is deferred to Appendix A.

The operational semantics of the process algebra language is expressed as a labelled binary small-step reduction relation $\dot{\longrightarrow} \subseteq \mathit{ProcConf} \times \mathit{Act} \times \mathit{ProcConf}$ over *process configurations*, defined as $\mathit{ProcConf} \triangleq \mathit{Proc} \times \mathit{ProcStore}$, i.e., pairs of processes and process stores. Any process configuration of the form (ε, σ) is said to be *final*, whereas configurations of the form (δ, σ) are defined to be *deadlocked*. Moreover, the transition labels are defined to be actions.

Before giving the transition rules, we first define a notion of *successful termination*.

Definition 5.2.3 (Successful termination)**.**

$$\downarrow\text{-EPSILON}\quad \varepsilon\downarrow \qquad \downarrow\text{-ITER}\quad P^*\downarrow \qquad \downarrow\text{-SEQ}\quad \frac{P\downarrow \qquad Q\downarrow}{P\cdot Q\downarrow} \qquad \downarrow\text{-ALT-L}\quad \frac{P\downarrow}{P+Q\downarrow} \qquad \downarrow\text{-ALT-R}\quad \frac{Q\downarrow}{P+Q\downarrow}$$

$$\downarrow\text{-PAR}\quad \frac{P\downarrow \qquad Q\downarrow}{P \parallel Q\downarrow} \qquad \downarrow\text{-MERGE}\quad \frac{P\downarrow \qquad Q\downarrow}{P \lfloor\!\lfloor Q\downarrow}$$

Intuitively, any process P can terminate successfully if P has the choice to have no further behaviour. This means that ε can always successfully terminate, while δ can never successfully terminate. Iteration P^* can successfully terminate, as it may choose not to start iterating and thereby to behave as ε.

We now define the transition rules of the operational semantics of processes.

Definition 5.2.4 (Operational semantic of processes)**.**

$$\frac{[\![\mathsf{pre}(a)]\!]\sigma \qquad [\![\mathsf{post}(a)]\!]\sigma'}{(a,\sigma) \xrightarrow{a} (\varepsilon,\sigma')} \longrightarrow\text{-ACT} \qquad \frac{(P,\sigma) \xrightarrow{a} (P',\sigma')}{(P \cdot Q),\sigma \xrightarrow{a} (P' \cdot Q,\sigma')} \longrightarrow\text{-SEQ-L}$$

$$\frac{P\downarrow \qquad (Q,\sigma) \xrightarrow{a} (Q',\sigma')}{(P \cdot Q,\sigma) \xrightarrow{a} (Q',\sigma')} \longrightarrow\text{-SEQ-R} \qquad \frac{(P,\sigma) \xrightarrow{a} (P',\sigma')}{(P + Q,\sigma) \xrightarrow{a} (P',\sigma')} \longrightarrow\text{-ALT-L}$$

$$\frac{(Q,\sigma) \xrightarrow{a} (Q',\sigma')}{(P + Q,\sigma) \xrightarrow{a} (Q',\sigma')} \longrightarrow\text{-ALT-R} \qquad \frac{(P,\sigma) \xrightarrow{a} (P',\sigma')}{(P \parallel Q,\sigma) \xrightarrow{a} (P' \parallel Q,\sigma')} \longrightarrow\text{-PAR-L}$$

$$\frac{(Q,\sigma) \xrightarrow{a} (Q',\sigma')}{(P \parallel Q,\sigma) \xrightarrow{a} (P \parallel Q',\sigma')} \longrightarrow\text{-PAR-R} \qquad \frac{(P,\sigma) \xrightarrow{a} (P',\sigma')}{(P \lfloor\!\lfloor Q,\sigma) \xrightarrow{a} (P' \parallel Q,\sigma')} \longrightarrow\text{-MERGE}$$

$$\frac{(P,\sigma) \xrightarrow{a} (P',\sigma')}{(P^*,\sigma) \xrightarrow{a} (P' \cdot P^*,\sigma')} \longrightarrow\text{-ITER}$$

Most of the reduction rules are standard in spirit [FZ94]. However, the handling of actions and their contracts make this process algebra language non-standard. More specifically, the non-standard $\longrightarrow$-ACT rule for action handling permits the state σ to change in any way, as long as these changes comply with the action contract. We will later use this transition rule to connect shared-memory updates in programs, to action contract-complying state changes on the process level.

Moreover, we use the notion of successful termination to define the transition rule for sequential composition, $\longrightarrow$-SEQ-R. This is standard in process algebra with ε [Bae00]. An example of sequential composition and termination is given below.

Example 5.2.1 (Termination and sequential composition)**.** *Consider the process $P = (a + b^*) \cdot c$ that is composed out of three actions, $a, b, c \in Act$. From any given state σ that satisfies c's precondition, P may be reduced to ε as shown by the following proof tree, assuming that c's postcondition is satisfiable (in this case with respect to σ').*

$$\dfrac{\dfrac{\dfrac{}{b^* \downarrow}\downarrow\text{-ITER}}{a + b^* \downarrow}\downarrow\text{-ALT-R} \qquad \dfrac{[\![\mathsf{pre}(c)]\!]\sigma \qquad [\![\mathsf{post}(c)]\!]\sigma'}{(c,\sigma) \xrightarrow{c} (\varepsilon,\sigma')}\longrightarrow\text{-ACT}}{((a + b^*) \cdot c,\sigma) \xrightarrow{c} (\varepsilon,\sigma')}\longrightarrow\text{-SEQ-R}$$

5.2.3 Bisimulation

Our model-based verification approach allows process-algebraic models to be handled *up to bisimulation*. Bisimulation is defined in the following way.

Definition 5.2.5 (Bisimulation)**.** *Any binary relation $\mathscr{R} \subseteq Proc \times Proc$ over processes is defined to be a* bisimulation relation *if, whenever $P \mathscr{R} Q$, then:*

(1) $P \downarrow$ if and only if $Q \downarrow$.

(2) For any σ, P', σ', a, if $(P,\sigma) \xrightarrow{a} (P',\sigma')$, then there exists a Q' such that $(Q,\sigma) \xrightarrow{a} (Q',\sigma')$ and $P' \mathscr{R} Q'$.

(3) For any σ, Q', σ', a, if $(Q,\sigma) \xrightarrow{a} (Q',\sigma')$, then there exists a P' such that $(P,\sigma) \xrightarrow{a} (P',\sigma')$ and $P' \mathscr{R} Q'$.

Two processes P and Q are defined to be *bisimilar*, or *bisimulation equivalent*, which is written $P \cong Q$, if and only if there exists a bisimulation relation $\mathscr{R}$ such that $P \mathscr{R} Q$. Bisimilarity expresses that both processes exhibit the same behaviour, in the sense that their action sequences describe the same state changes. Bisimilarity therefore also preserves successful termination.

Any bisimulation relation constitutes an equivalence relation (that is, a relation that is reflexive, symmetric and transitive). Furthermore, bisimilarity is a congruence for all process algebraic connectives.

Successful termination $P \downarrow$ can intuitively be understood as P being bisimilar to the process $\varepsilon + P$, i.e., by having the choice to have no further behaviour.

Proposition 5.2.1. *If $P \downarrow$ then $P \cong \varepsilon + P$.*

Axiomatisation. Figure 5.1 presents several standard axioms of our process algebra language, which is based on [GM14], but extended with axioms for ε and iteration, as proposed in [Mil84]. The axioms have been proven sound with respect to bisimilarity and are therefore presented as $\cong$-equalities.

Theorem 5.2.1. *All $\cong$-equivalences presented in Figure 5.1 hold.*

Equivalences for the sequential connectives

A1 $P + Q \cong Q + P$

A2 $P + (Q + R) \cong (P + Q) + R$

A3 $P + P \cong P$

A4 $(P + Q) \cdot R \cong P \cdot R + Q \cdot R$

A5 $P \cdot (Q \cdot R) \cong (P \cdot Q) \cdot R$

A6 $P + \delta \cong P$

A7 $\delta \cdot P \cong \delta$

A8 $P \cdot \varepsilon \cong P$

A9 $\varepsilon \cdot P \cong P$

Equivalences for the parallel connectives

P1 $P \parallel Q \cong Q \parallel P$

P2 $P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R$

P3 $P \parallel Q \cong P \between Q + Q \between P$

P4 $\varepsilon \parallel P \cong P$

P5 $P \parallel \delta \cong P \cdot \delta$

LM1 $\delta \between P \cong \delta$

LM2 $\varepsilon \between \delta \cong \delta$

LM3 $\varepsilon \between (a \cdot P) \cong \delta$

LM4 $(a \cdot P) \between Q \cong a \cdot (P \parallel Q)$

LM5 $\varepsilon \between \varepsilon \cong \varepsilon$

LM6 $\varepsilon \between (P + Q) \cong \varepsilon \between P + \varepsilon \between Q$

LM7 $(P + Q) \between R \cong P \between R + Q \between R$

LM8 $(P \between Q) \between R \cong P \between (Q \parallel R)$

LM9 $P \between \delta \cong P \cdot \delta$

Equivalences for iteration

KL1 $P^* \cong P \cdot P^* + \varepsilon$

KL2 $\delta^* \cong \varepsilon$

KL3 $\varepsilon^* \cong \varepsilon$

KL4 $P^{**} \cong P^*$

KL5 $P^* \cdot P^* \cong P^*$

KL6 $(P + Q)^* \cong P^* \cdot (Q \cdot P^*)^*$

Figure 5.1: Standard bisimulation equivalences of process algebra.

We will later see that the program logic allows rewriting processes up to bisimilarity, for example by using these axioms. They therefore give a good indication of how process-algebraic models can be used in our approach.

5.3 Programs

This section defines the syntax and semantics of programs.

5.3.1 Syntax

Our verification approach is formalised on the following simple concurrent pointer language, where $X, Y, \cdots \in \mathit{Var}$ are *(program) variables*.

Definition 5.3.1 (Expressions, Conditions, Abstraction binders, Commands)**.**

$$
\begin{aligned}
E \in \mathit{Expr} &::= n \mid X \mid E + E \mid E - E \mid \cdots \\
B \in \mathit{Cond} &::= \mathsf{true} \mid \mathsf{false} \mid \neg B \mid B \wedge B \mid E = E \mid E < E \mid \cdots \\
\Pi \in \mathit{AbstrBinder} &::= \{x_0 \mapsto E_0, \ldots, x_n \mapsto E_n\} \\
C \in \mathit{Cmd} &::= \mathbf{skip} \mid X := E \mid X := [E] \mid [E] := E \mid C; C \mid \\
&\quad X := \mathbf{alloc}\ E \mid \mathbf{dispose}\ E \mid \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \\
&\quad \mathbf{while}\ B\ \mathbf{do}\ C \mid \mathbf{atomic}\ C \mid \mathbf{inatom}\ C \mid C \parallel C \mid \\
&\quad X := \mathbf{process}\ p\ \mathbf{over}\ \Pi \mid \mathbf{action}\ X.a\ \mathbf{do}\ C \mid \mathbf{inact}\ C \mid \\
&\quad \mathbf{finish}\ X
\end{aligned}
$$

This language is a variation of the language proposed by O'Hearn [O'H07] and Brookes [Bro07]. In particular, we extend their language with *specification-only* commands, (code annotations) for handling process-algebraic models. These commands are coloured blue. Note that the blue colourings do not have any semantic meaning; they only indicate which language constructs are specification-only. Moreover, we interchangeably refer to commands also as programs.

We assume that process-algebraic models are externally declared, so that any (top-level) program C is used in a context $\mathbf{let}\ p_0 := P_0, \ldots, p_n := P_n\ \mathbf{in}\ C$ of process declarations, where all $p_i \in \mathit{ProcLabel}$ are process names/labels. However, for ease of presentation, instead of explicitly using such declaration contexts, we simply use the notation $\mathsf{body}(p)$ to refer to the process declared under the name p.

Standard language constructs. The notation $[E]$ stands for *heap dereferencing*, where E is an expression whose evaluation determines the heap location to

dereference. The commands $X := [E]$ and $[E] := E'$ denote heap reading and writing: they read from, and write to, the heap at location E, respectively. Moreover, $X :=$ **alloc** E allocates a free heap location and writes the value represented by E to it, whereas **dispose** E deallocates the heap location at E.

Regarding concurrency, the command $C_1 \parallel C_2$ is the statically-scoped parallel composition of C_1 and C_2, and expresses their concurrent execution. In the sequel, we sometimes refer to commands that are put in parallel as different *threads*; for example, C_1 and C_2 in the above. Moreover, **atomic** C expresses a statically-scoped lock: it represents the atomic execution of C, that is, without interference of other threads. The command **inatom** C represents *partially executed* atomic programs: ones that are currently being executed, where C is the remaining program that still has to be executed atomically. Such commands are sometimes referred to as "runtime syntax", as they are not written by users of the language, but are instead an artefact of program execution.

Specification-only constructs. The instructions that are displayed in blue are the specification-only language constructs, for handling process-algebraic models in the logic. These instructions are ignored during regular program execution and are essentially handled as if they were code comments.

Specification-wise, $X :=$ **process** p **over** Π initialises a new program model in the proof system, represented by the process $\mathsf{body}(p)$ that is declared under the name p for the program. The Π component is an *abstraction binder*, which is defined in Definition 5.3.1, and is used to connect process-algebraic variables to heap locations in the program. In particular, the abstraction binders make the connections/links between process-algebraic state and shared-memory state (heap locations) in the program. In the sequel, we often use abstraction binders as if they were finite partial mappings, i.e., $\Pi : \mathit{ProcVar} \rightharpoonup_{\mathsf{fin}} \mathit{Expr}$, from process-algebraic variables to the expressions whose evaluation determine the corresponding heap location. Finally, the variable X is used as an identifier for the freshly initialised process-algebraic model.

The command **finish** X is used to *finalise* the process-algebraic model that is identified by X. Finalisation in this respect means that the model has fully been executed (i.e., can successfully terminate), and that the program logic can obtain the model's postcondition. Finalisation is later explained in more detail, in Section 5.5.

The **action** $X.a$ **do** C specification command is used to link the execution of programs with the execution of process-algebraic models. More specifically, it executes the program C in the context of a model identified by X, as the process-algebraic action a. The soundness argument of the program logic establishes a refinement

relation between programs and their models, and this relation is established by synchronising program execution with process execution, with help of these *action blocks*.

And last, the specification command **inact** C denotes a *partially executed* action program, namely one that still has to execute C. Likewise to **inatom**, this command can only occur during runtime, and cannot be written by users of the language.

Free variables and substitution. We use the standard notations $\mathsf{fv}(E)$, $\mathsf{fv}(B)$, $\mathsf{fv}(\Pi)$ and $\mathsf{fv}(C)$ to refer to the set of free *program* variables in the given (Boolean) expression E and B, abstraction binder Π, and command C, respectively. Moreover, the notation $E[X/E']$ denotes the *substitution* of the program variable X for the expression E' inside E; and likewise for Boolean expressions, abstraction binders, and commands. The full definitions of $\mathsf{fv}(\cdot)$ and $\cdot\,[X/E]$ are mostly standard, and therefore deferred to Appendix A.

User programs. As just discussed, our simple programming language contains runtime syntax: instructions that are not written by users but are only introduced during runtime. Commands that are free of such runtime constructs are called *user commands*.

Definition 5.3.2 (User commands)**.** *Any command C is defined to be a* user command*, denoted C : user, if C does not contain sub-commands of the forms* **inatom** C' *and* **inact** C'*, for any command C'.*

Well-formedness. Moreover, our approach only applies to commands that are *well-formed*. Notably, our technique requires that, for any program of the form **action** _ **do** C and **inact** C, the inner action program C only contains a subcategory of commands, excluding atomic commands and specification-only constructs, in particular nested action blocks. The latter is needed since actions must be atomically observable by environmental threads. This restriction is captured by the following definitions.

Definition 5.3.3 (Basic programs)**.** *Any command C is defined to be* basic*, denoted C : basic, if C does not contain any atomic sub-programs, i.e.,* **atomic** *or* **inatom***, nor specification-specific language constructs, i.e.,* **process**, **action**, **inact**, *or* **finish**.

Definition 5.3.4 (Well-formed programs)**.** *A command C is defined to be* well-formed*, denoted C : wf, if, for any command* **action** _ **do** C' *or* **inact** C' *that*

occurs in C, the sub-program C' is basic.

Lemma 5.3.1. *For any command C, if C : basic, then C : wf.*

More precise definitions and additional properties of user programs, basic programs and wellformedness can be found in Appendix A.

5.3.2 Operational Semantics

The denotational semantics of expressions $[\![E]\!]s$ and conditions $[\![B]\!]s$ are again defined in the standard way, and evaluate to *Val* and *Prop*, respectively, where $s \in \mathit{Store} \triangleq \mathit{Var} \to \mathit{Val}$ is a *store* that gives an interpretation to all program variables. More details regarding denotational semantics are deferred to Appendix A.

The operational semantics of programs is defined in terms of a binary small-step reduction relation $\rightsquigarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ between *program configurations*. A program configuration $\mathfrak{C} = (C, h, s) \in \mathit{Conf} \triangleq \mathit{Cmd} \times \mathit{Heap} \times \mathit{Store}$ is a triple, consisting of a command C; as well as a *heap* h that models shared memory, and finally a store $s \in \mathit{Store}$ that models thread-local memory. Any configuration of the form $(\mathbf{skip}, h, s)$ is defined to be *final*, or *terminated*. Heaps $h \in \mathit{Heap} \triangleq \mathit{Val} \rightharpoonup_{\mathsf{fin}} \mathit{Val}$ are defined to be finite partial mappings from values to values. Notably, we assume that heap locations are simply values, that can be assigned to, and read from, local variables. We moreover use the function $\mathsf{dom} : \mathit{Heap} \to 2^{\mathit{Val}}$ to denote the *mapped domain* of a specified heap, that is, $\mathsf{dom}(h) \triangleq \{v \mid h(v) \neq \text{undefined}\}$.

Definition 5.3.5 (Small-step operational semantics of programs)**.**

$$\frac{}{(X := E, h, s) \rightsquigarrow (\mathbf{skip}, h, s[X \mapsto [\![E]\!]s])}\ \rightsquigarrow\text{-ASSIGN}$$

$$\frac{}{(X := [E], h, s) \rightsquigarrow (\mathbf{skip}, h, s[X \mapsto h([\![E]\!]s)])}\ \rightsquigarrow\text{-READ}$$

$$\frac{[\![E_1]\!]s \in \mathsf{dom}(h)}{([E_1] := E_2, h, s) \rightsquigarrow (\mathbf{skip}, h[[\![E_1]\!]s \mapsto [\![E_2]\!]s], s)}\ \rightsquigarrow\text{-WRITE}$$

$$(\mathbf{while}\ B\ \mathbf{do}\ C, h, s) \rightsquigarrow (\mathbf{if}\ B\ \mathbf{then}\ (C; \mathbf{while}\ B\ \mathbf{do}\ C)\ \mathbf{else}\ \mathbf{skip}, h, s)\ \rightsquigarrow\text{-WHILE}$$

$$\frac{(C_1, h, s) \rightsquigarrow (C_1', h', s')}{(C_1; C_2, h, s) \rightsquigarrow (C_1'; C_2, h', s')}\ \rightsquigarrow\text{-SEQ-L} \qquad \frac{}{(\mathbf{skip}; C, h, s) \rightsquigarrow (C, h, s)}\ \rightsquigarrow\text{-SEQ-R}$$

$$\frac{v \notin \mathsf{dom}(h)}{(X := \textbf{alloc}\ E, h, s) \rightsquigarrow (\textbf{skip}, h[v \mapsto [\![E]\!]s], s[X \mapsto v])}\ \rightsquigarrow\text{-ALLOC}$$

$$(\textbf{dispose}\ E, h, s) \rightsquigarrow (\textbf{skip}, h \setminus [\![E]\!]s, s)\ \rightsquigarrow\text{-DISPOSE} \qquad (\textbf{atomic}\ C, h, s) \rightsquigarrow (\textbf{inatom}\ C, h, s)\ \rightsquigarrow\text{-ATOM}$$

$$(\textbf{inatom skip}, h, s) \rightsquigarrow (\textbf{skip}, h, s)\ \rightsquigarrow\text{-INATOM-SKIP} \qquad (\textbf{skip} \parallel \textbf{skip}, h, s) \rightsquigarrow (\textbf{skip}, h, s)\ \rightsquigarrow\text{-PAR-SKIP}$$

$$(X := \textbf{process}\ p\ \textbf{over}\ \Pi, h, s) \rightsquigarrow (\textbf{skip}, h, s)\ \rightsquigarrow\text{-PROC} \qquad (\textbf{finish}\ X, h, s) \rightsquigarrow (\textbf{skip}, h, s)\ \rightsquigarrow\text{-FINISH}$$

$$(\textbf{action}\ X.a\ \textbf{do}\ C, h, s) \rightsquigarrow (\textbf{inact}\ C, h, s)\ \rightsquigarrow\text{-ACT} \qquad (\textbf{inact skip}, h, s) \rightsquigarrow (\textbf{skip}, h, s)\ \rightsquigarrow\text{-INACT-SKIP}$$

$$\frac{(C, h, s) \rightsquigarrow (C', h', s')}{(\textbf{inact}\ C, h, s) \rightsquigarrow (\textbf{inact}\ C', h', s')}\ \rightsquigarrow\text{-INACT-STEP} \qquad \frac{[\![B]\!]s}{(\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2, h, s) \rightsquigarrow (C_1, h, s)}\ \rightsquigarrow\text{-IF-TRUE}$$

$$\frac{\neg[\![B]\!]s}{(\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2, h, s) \rightsquigarrow (C_2, h, s)}\ \rightsquigarrow\text{-IF-FALSE}$$

$$\frac{(C, h, s) \rightsquigarrow (C', h', s')}{(\textbf{inatom}\ C, h, s) \rightsquigarrow (\textbf{inatom}\ C', h', s')}\ \rightsquigarrow\text{-INATOM-STEP} \qquad \frac{\neg\mathsf{locked}(C_2) \quad (C_1, h, s) \rightsquigarrow (C_1', h', s')}{(C_1 \parallel C_2, h, s) \rightsquigarrow (C_1' \parallel C_2, h', s')}\ \rightsquigarrow\text{-PAR-L}$$

$$\frac{\neg\mathsf{locked}(C_1) \quad (C_2, h, s) \rightsquigarrow (C_2', h', s')}{(C_1 \parallel C_2, h, s) \rightsquigarrow (C_1 \parallel C_2', h', s')}\ \rightsquigarrow\text{-PAR-R}$$

Most of the transition rules are standard; see for example [Vaf11]. The *update* notation $s[X \mapsto v]$ denotes a store that is equal to s, except that X is now mapped to $v \in \mathit{Val}$. We use a similar notation for heaps, namely $h[v_1 \mapsto v_2]$. Moreover, the notation $h \setminus v$ denotes the *removal* of the entry at v in h.

Definition 5.3.6 (Store update, Heap removal)**.**

$$s[X \mapsto v] \triangleq \lambda Y\,.\,\begin{cases} v & \mathit{if}\ X = Y \\ s(Y) & \mathit{otherwise} \end{cases} \qquad h \setminus v \triangleq \lambda v'\,.\,\begin{cases} h(v') & \mathit{if}\ v \neq v' \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

An interesting aspect of the operational semantics is that atomic programs are executed using a small-step reduction strategy (via $\rightsquigarrow$-INATOM-STEP and $\rightsquigarrow$-INATOM-SKIP), rather than a big-step execution, which is more customary. This is done for technical reasons: it simplifies the establishment of a simulation/refinement between programs and their models. Consequently, we use a notion of a *locked program* to define the transition rules for atomic programs. Any command C is said to be *(globally) locked* if C executes an atomic program, that is, if C has **inatom** C' as a subprogram for some C'.

The following definition formally captures this notion of locking.

Definition 5.3.7 (Locked programs)**.** *Any command C is* locked *if* $\mathsf{locked}(C)$ *holds, where* $\mathsf{locked} : \mathit{Cmd} \to \mathit{Prop}$ *is defined by structural recursion on C, as follows:*

$$\mathsf{locked}(C) \triangleq \begin{cases} \mathsf{true} & \text{if } C = \textbf{inatom } C' \\ \mathsf{locked}(C_1) & \text{if } C = C_1; C_2 \\ \mathsf{locked}(C_1) \vee \mathsf{locked}(C_2) & \text{if } C = C_1 \parallel C_2 \\ \mathsf{locked}(C') & \text{if } C = \textbf{inact } C' \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Moreover, any command C is defined to be unlocked*, if* $\neg\mathsf{locked}(C)$.

The rules $\rightsquigarrow$-PAR-L and $\rightsquigarrow$-PAR-R for parallel composition allow a thread to make an execution step only if the other thread is not locked, i.e., not executing an atomic program, thereby preventing thread interference while executing atomic programs. One might ask whether this handling of locks could not potentially lead to deadlock scenarios, for example by encountering configurations $(C_1 \parallel C_2, h, s)$ during runtime, in which both $\mathsf{locked}(C_1)$ and $\mathsf{locked}(C_2)$. However, we will later see and prove that no such deadlocks can be reached, given that one starts with an initial configuration that contains a user program.

Furthermore, the specification-only language constructs do not affect the state of the program (i.e., the heap or the store), and are essentially handled as if they were comments. Notice however, that commands of the form **action** _ **do** C are first reduced to **inact** C, before C is being executed. This is also done for technical reasons, as this makes it more convenient to later establish a simulation relation between execution steps of programs and processes.

The semantics of programs has the following preservation properties.

Lemma 5.3.2. *Program execution preserves basicality and well-formedness:*

1. *If* $C : \mathsf{basic}$ *and* $(C, h, s) \rightsquigarrow (C', h', s')$*, then* $C' : \mathsf{basic}$.
2. *If* $C : \mathsf{wf}$ *and* $(C, h, s) \rightsquigarrow (C', h', s')$*, then* $C' : \mathsf{wf}$.

5.3.3 Fault Semantics

Apart from an operational semantics, we also define a *fault semantics* [Rey02], that classifies runtime errors that may occur during program execution. Before giving its definition, let us first introduce two auxiliary functions, $\mathsf{acc}(C, s)$ and $\mathsf{writes}(C, s)$, for obtaining the set of heap locations that can be accessed or written-to in the next execution step of C, respectively.

Definition 5.3.8 (Shared-memory accesses)**.** *The function* $\mathsf{acc} : \mathit{Cmd} \to \mathit{Store} \to 2^{\mathit{Val}}$ *collects all heap locations that are* accessed *by a given command in the next computation step, and is defined by structural recursion as follows.*

$$
\begin{aligned}
\mathsf{acc}(\mathbf{skip}, s) &\triangleq \emptyset \\
\mathsf{acc}(X := E, s) &\triangleq \emptyset \\
\mathsf{acc}(X := [E], s) &\triangleq \{[\![E]\!]s\} \\
\mathsf{acc}([E_1] := E_2, s) &\triangleq \{[\![E_1]\!]s\} \\
\mathsf{acc}(C_1; C_1, s) &\triangleq \mathsf{acc}(C_1, s) \\
\mathsf{acc}(X := \mathbf{alloc}\ E, s) &\triangleq \emptyset \\
\mathsf{acc}(\mathbf{dispose}\ E, s) &\triangleq \{[\![E]\!]s\} \\
\mathsf{acc}(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s) &\triangleq \emptyset \\
\mathsf{acc}(\mathbf{while}\ B\ \mathbf{do}\ C, s) &\triangleq \emptyset \\
\mathsf{acc}(C_1 \parallel C_2, s) &\triangleq \mathsf{acc}(C_1, s) \cup \mathsf{acc}(C_2, s) \\
\mathsf{acc}(\mathbf{atomic}\ C, s) &\triangleq \emptyset \\
\mathsf{acc}(\mathbf{inatom}\ C, s) &\triangleq \mathsf{acc}(C, s) \\
\mathsf{acc}(X := \mathbf{process}\ p\ \mathbf{over}\ \{x_0 \mapsto E_0, \cdots, x_n \mapsto E_n\}, s) &\triangleq \{[\![E_i]\!]s \mid 0 \leq i < n\} \\
\mathsf{acc}(\mathbf{action}\ X.a\ \mathbf{do}\ C, s) &\triangleq \emptyset \\
\mathsf{acc}(\mathbf{inact}\ C, s) &\triangleq \mathsf{acc}(C, s) \\
\mathsf{acc}(\mathbf{finish}\ X, s) &\triangleq \emptyset
\end{aligned}
$$

Notice that in the case of heap allocation $X := \mathbf{alloc}\ E$, no heap location is being collected, as these can only be accessed after being allocated. Moreover, also

notice that the initialisation of a new program model using **process** causes all heap locations that are protected by that model to be accessed. Even though the operational semantics of programs does not actually access these heap locations, inside the program logic they *are* affected. We therefore include them in acc for later convenience, thus making acc an over-approximation of the accessed memory locations.

Definition 5.3.9 (Shared-memory writes)**.** *The set of heap locations that are* written to *by a given command in the next computation step is determined by the function* writes : *Cmd* $\to$ *Store* $\to 2^{Val}$*, which is defined as follows.*

$$
\begin{aligned}
\mathsf{writes}(\mathbf{skip}, s) &\triangleq \emptyset \\
\mathsf{writes}(X := E, s) &\triangleq \emptyset \\
\mathsf{writes}(X := [E], s) &\triangleq \emptyset \\
\mathsf{writes}([E_1] := E_2, s) &\triangleq \{[\![E_1]\!]s\} \\
\mathsf{writes}(C_1; C_2, s) &\triangleq \mathsf{writes}(C_1, s) \\
\mathsf{writes}(X := \mathbf{alloc}\ E, s) &\triangleq \emptyset \\
\mathsf{writes}(\mathbf{dispose}\ E, s) &\triangleq \{[\![E]\!]s\} \\
\mathsf{writes}(\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C, s) &\triangleq \emptyset \\
\mathsf{writes}(\mathbf{while}\ B\ \mathbf{do}\ C, s) &\triangleq \emptyset \\
\mathsf{writes}(C_1 \parallel C_2, s) &\triangleq \mathsf{writes}(C_1, s) \cup \mathsf{writes}(C_2, s) \\
\mathsf{writes}(\mathbf{atomic}\ C, s) &\triangleq \emptyset \\
\mathsf{writes}(\mathbf{inatom}\ C, s) &\triangleq \mathsf{writes}(C, s) \\
\mathsf{writes}(X := \mathbf{process}\ p\ \mathbf{over}\ \Pi, s) &\triangleq \emptyset \\
\mathsf{writes}(\mathbf{action}\ X.a\ \mathbf{do}\ C, s) &\triangleq \emptyset \\
\mathsf{writes}(\mathbf{inact}\ C, s) &\triangleq \mathsf{writes}(C, s) \\
\mathsf{writes}(\mathbf{finish}\ X, s) &\triangleq \emptyset
\end{aligned}
$$

It is not difficult to show that all shared-memory writes, according to the above definition, as also shared-memory accesses.

Lemma 5.3.3. *For every* C *and* s *it holds that* $\mathsf{writes}(C, s) \subseteq \mathsf{acc}(C, s)$.

We are now ready to define the fault semantics for our programming language. The fault semantics is expressed as a set $\lightning \subset \mathit{Conf}$ of 'faulting' program configurations, that is inductively defined in the following way, where $\lightning(\mathfrak{C})$ is written as a shorthand notation for $\mathfrak{C} \in \lightning$.

Definition 5.3.10 (Fault semantics)**.**

$$\frac{[\![E]\!]s \notin \mathsf{dom}(h)}{\lightning(X := [E], h, s)}\ \lightning\text{-READ} \qquad \frac{[\![E_1]\!]s \notin \mathsf{dom}(h)}{\lightning([E_1] := E_2, h, s)}\ \lightning\text{-WRITE} \qquad \frac{[\![E]\!]s \notin \mathsf{dom}(h)}{\lightning(\textbf{dispose}\ E, h, s)}\ \lightning\text{-DISPOSE}$$

$$\frac{\lightning(C_1, h, s)}{\lightning(C_1; C_2, h, s)}\ \lightning\text{-SEQ} \qquad \frac{\lightning(C_1, h, s) \qquad \neg\mathsf{locked}(C_2)}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-PAR-L} \qquad \frac{\lightning(C_2, h, s) \qquad \neg\mathsf{locked}(C_1)}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-PAR-R}$$

$$\frac{\mathsf{locked}(C_1) \qquad \mathsf{locked}(C_2)}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-DEADLOCK} \qquad \frac{\neg\mathsf{locked}(C_1) \qquad \neg\mathsf{locked}(C_2) \qquad \mathsf{acc}(C_1, s) \cap \mathsf{writes}(C_2, s) \neq \emptyset}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-RACE-1}$$

$$\frac{\neg\mathsf{locked}(C_1) \qquad \neg\mathsf{locked}(C_2) \qquad \mathsf{acc}(C_2, s) \cap \mathsf{writes}(C_1, s) \neq \emptyset}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-RACE-2} \qquad \frac{\lightning(C, h, s)}{\lightning(\textbf{inatom}\ C, h, s)}\ \lightning\text{-ATOMIC} \qquad \frac{\lightning(C, h, s)}{\lightning(\textbf{inact}\ C, h, s)}\ \lightning\text{-ACTION}$$

Intuitively, a program configuration exhibits a fault if it: accesses unallocated memory, or is deadlocked, or allows performing a data-race.

More specifically, $\lightning$-READ expresses that heap reading $X := [E]$ faults if the heap location at E is unoccupied. For the same reason, also heap writing ($\lightning$-WRITE) and heap deallocation ($\lightning$-DISPOSE) may fault. The $\lightning$-PAR-L rule expresses that any parallel program $C_1 \parallel C_2$ can fault if C_1 can fault, given that C_2 is not locked (the rule $\lightning$-PAR-R covers the other direction). The locking requirement is needed here, since otherwise C_1 would not be able to perform any execution step according to the reduction rules of the operational semantics. Program configurations that hold multiple global locks are also considered to be faulting, via $\lightning$-DEADLOCK. Finally, the fault semantics encodes the definition of a *data-race*, via $\lightning$-RACE-1 and $\lightning$-RACE-2. To clarify, any configuration (C, h, s) exhibits a data-race, if C has (at least) two threads that can both access a common location in h in the next computation step, where at least one of these accesses is a write. Both $\lightning$-RACE-1 and $\lightning$-RACE-2 have extra premises to ensure that the two threads are unlocked, to allow both of them to perform a computation step, and therewith to access the shared heap location.

We will later see that the soundness argument of our program logic covers that verified programs are free of faults. More specifically, we will prove that, for any program C for which a proof can be derived, we have that C is fault-free with

respect to a heap h and a store s that satisfy C's precondition, and moreover, that every configuration that is reachable from (C, h, s) is also fault-free.

Finally, to show that the operational semantics of programs is coherent with respect to the fault semantics, we prove that the operational semantics is *progressive* for all non-faulting program configurations.

Theorem 5.3.4 (Progress of $\rightsquigarrow$)**.** *For any program configuration* $\mathfrak{C} \notin \lightning$*, either* $\mathfrak{C}$ *is final, or there exists a configuration* $\mathfrak{C}'$ *such that* $\mathfrak{C} \rightsquigarrow \mathfrak{C}'$*.*

5.4 Assertions

This section discusses the syntax and interpretation of the assertion language. We first define and discuss the assertion language in Section 5.4.1. Then, Section 5.4.2 introduces *permission heaps* (§5.4.2.2) and *process maps* (§5.4.2.3), which are the composable structures that form the foundation of the models of the program logic. Finally, Section 5.4.3 defines the semantic interpretation of assertions.

5.4.1 Assertion Language

The assertion language of our approach is defined by the following grammar.

Definition 5.4.1 (Assertion language)**.**

$$\begin{aligned}\mathcal{P}, \mathcal{Q}, \mathcal{R} \in \mathit{Assn} ::= {} & B \mid \forall X.\mathcal{P} \mid \exists X.\mathcal{P} \mid \mathcal{P} \vee \mathcal{Q} \mid \mathcal{P} * \mathcal{Q} \mid \mathcal{P} \mathrel{-\!\!*} \mathcal{Q} \mid \\ & *_{i \in I} \mathcal{P}_i \mid E \xrightarrow{\pi}_t E \mid \mathsf{Proc}_\pi(X, p, P, \Pi)\end{aligned}$$

where t *is a* heap ownership type*, which is defined to be:*

$$t \in \mathit{PointsToType} ::= \mathsf{std} \mid \mathsf{proc} \mid \mathsf{act}$$

The definitions of free variables $\mathsf{fv}(\mathcal{P})$ of assertions $\mathcal{P}$, and substitution $\mathcal{P}[X/E]$ in $\mathcal{P}$, are the standard ones, and are therefore deferred to Appendix A.

Assertions can be built from plain Boolean expressions B, and may contain several standard connectives from predicate logic: universal quantifiers ($\forall X$), existential quantifiers ($\exists X$), and disjunction ($\vee$). Even though negation ($\neg$) is not included, recall that plain Boolean expressions can be negated. Moreover, logical conjunction ($\wedge$) is replaced by the *separating conjunction* $*$ from CSL. The $*_{i \in I} \mathcal{P}_i$ connective is the *iterated separating conjunction*, with I a finite set, that represents $\mathcal{P}_0 * \cdots * \mathcal{P}_n$, given that $I = \{0, \ldots, n\}$. The $\mathrel{-\!\!*}$ connective is known as the *magic wand*, also commonly called the *separating implication*, and is used to describe hypothetical judgments, much like the logical implication from predicate logic.

Apart from these standard CSL connectives, the assertion language contains three different heap ownership predicates $\overset{\pi}{\hookrightarrow}_t$, with $\pi \in \mathbb{Q}$ a rational number that represents a *fractional permission*, and t the *heap ownership type*, as well as an ownership predicate Proc_π for program abstractions.

Assertions that do not contain such heap/process ownership predicates are called *pure*, while any assertion that is not pure is defined to be *spatial*.

Heap ownership. The assertion $E_1 \overset{\pi}{\hookrightarrow}_t E_2$ is the *heap ownership assertion*, and expresses that the heap contains the value represented by the expression E_2, at heap location E_1. Moreover, π and t together determine the access rights to this heap location. In more detail, depending on the ownership type t, the $\overset{\pi}{\hookrightarrow}_t$ ownership predicates express different access rights to the associated heap location:

- **Standard heap ownership.** $E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E_2$ is the *standard heap ownership predicate* from (intuitionistic) separation logic that provides read-access whenever $0 < \pi < 1$, and write-access in case $\pi = 1$. Moreover, the subscript std indicates that the associated heap location E_1 is *not* bound to any process-algebraic model. We say that a heap location $v \in \mathit{Val}$ is *bound by*, or *subject to*, a program abstraction, if there is an active program abstraction with a binder Π that contains a mapping to v, i.e., $v \in \mathsf{dom}(\Pi)$.
- **Process heap ownership.** $E \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E'$ is the *process heap ownership predicate*, which indicates that the heap location at E is bound by an active process-algebraic abstraction, but in a purely *read-only* manner. More precisely, $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ assertions exclusively grant read-access, even in case $\pi = 1$.
- **Action heap ownership.** $E \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E'$ is the *action heap ownership predicate*, which indicates that the heap location E is bound by an active process-algebraic model, and is used in the context of an action block, in a *read/write* manner.

Observe that action points-to assertions $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ essentially give the same access rights as $\overset{\pi}{\hookrightarrow}_{\mathsf{std}}$ assertions. Nevertheless, they are both needed, to be able to distinguish between bound and unbound heap locations in the logic. For example, the program logic must not allow to deallocate memory is currently bound to (protected by) an active process-algebraic model, as this would be unsound.

Moreover, even though $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates never grant write access, we will later see that the proof system allows $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates to be upgraded to $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$, inside action blocks, and $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ again provides write access when $\pi = 1$. More precisely, $E \overset{1}{\hookrightarrow}_{\mathsf{proc}} E'$ predicates grant the *capability to regain write access to* E, in the context of an action program. This system of upgrading enforces that all modifications to E happen in the context of **action** $X.a$ **do** C commands, so that the

modifications are protected and can be recorded by the program abstraction X, as the action a.

In addition to these three heap ownership predicates, we derive a fourth such predicate, called the *process–action heap ownership predicate.* which is equivalent to $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ only if π denotes write access, and otherwise is equivalent to $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$.

Definition 5.4.2 (Process–action heap ownership predicate)**.**

$$E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{procact}} E_2 \triangleq \begin{cases} E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E_2 & \textit{if } \pi = 1 \\ E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E_2 & \textit{otherwise} \end{cases}$$

This derived predicate is for later use, in the proof system of our program logic.

We use the notation $E \overset{\pi}{\hookrightarrow}_t -$ is used as shorthand for $\exists X.E \overset{\pi}{\hookrightarrow}_t X$, where $X \notin \mathsf{fv}(E)$. Furthermore, we sometimes simply write $\overset{\pi}{\hookrightarrow}$ instead of $\overset{\pi}{\hookrightarrow}_{\mathsf{std}}$.

Process ownership. Finally, the $\mathsf{Proc}_\pi(X, p, P, \Pi)$ assertion expresses *ownership* of a program abstraction that is identified by X, where the abstraction is represented by the process P. Ownership in this sense means that the thread has knowledge of the existence of this process-algebraic model, as well as the right to execute as prescribed by this model. The label p identifies the declaration of the process-algebraic abstraction, as specified in the $X :=$ **process** p **over** Π ghost command that was used to initialise the abstract model in the program logic. Furthermore, Π connects the abstract model to the concrete program by mapping the process-algebraic variables in the abstraction to heap locations in the program, as discussed before. And last, the fractional permission π is needed to implement the ownership system of program models. Fractional permissions are only used here to be able to reconstruct the full Proc_1 predicate. We shall later see that Proc_π predicates can be split and merged along π and parallel compositions inside P, and consumed in the proof system by action programs.

5.4.2 Models of the Program Logic

Before Section 5.4.3 discusses the semantics of our assertion language, this section first introduces *permission heaps* and *process maps*, that form the basis for the models of our concurrent separation logic. Permission heaps extend ordinary program heaps (i.e., *Heap*) to capture the three different types t of heap ownership, whereas process maps capture the state and ownership of process-algebraic abstractions.

Let us start by introducing *fractional permissions*, which are used in the definitions of both permission heaps and process maps.

5.4.2.1 Fractional Permissions

In the assertion language, all heap/process ownership predicates have an associated rational number $\pi \in \mathbb{Q}$. There are used to express the "amount" of ownership that is available to the corresponding heap location or program model.

We define a rational number π to be a *(Boyland) fractional permission* in case $\pi \in (0,1]_{\mathbb{Q}}$ [Boy03]. The original work of Boyland uses fractional permissions to distinguish between write access ($\pi = 1$) and read access ($0 < \pi < 1$) to some shared resource. However, in our work this is slightly different, since the fractional access permissions π annotated to $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates never provide write access.

To conveniently handle fractional permissions, we define basic notions of *validity* ($\mathsf{valid}_{\mathbb{Q}}$) and *disjointness* ($\perp_{\mathbb{Q}}$) of rational numbers, as follows.

Definition 5.4.3 (Permission validity, Permission disjointness)**.**

$$\mathsf{valid}_{\mathbb{Q}}\,\pi \triangleq 0 < \pi \leq 1 \qquad \pi_1 \perp_{\mathbb{Q}} \pi_2 \triangleq 0 < \pi_1 \wedge 0 < \pi_2 \wedge \pi_1 + \pi_2 \leq 1$$

The predicate $\mathsf{valid}_{\mathbb{Q}} : \mathbb{Q} \to \mathit{Prop}$ determines whether the given rational number is within the range $(0,1]_{\mathbb{Q}}$, that is, is a valid Boyland fractional permission.

The binary relation $\perp_{\mathbb{Q}} : \mathbb{Q} \to \mathbb{Q} \to \mathit{Prop}$ determines *disjointness* of two rationals. Disjoint rational numbers do not overlap, in the sense that both operands are fractional permissions, as well as their addition.

Lemma 5.4.1. $\mathsf{valid}_{\mathbb{Q}}$ *and* $\perp_{\mathbb{Q}}$ *satisfy the following properties.*

1. *If* $\pi_1 \perp_{\mathbb{Q}} \pi_2$*, then* $\pi_2 \perp_{\mathbb{Q}} \pi_1$*,* $\mathsf{valid}_{\mathbb{Q}}\,\pi_1$*, and* $\mathsf{valid}_{\mathbb{Q}}\,(\pi_1 + \pi_2)$*.*
2. *If* $\pi_1 \perp_{\mathbb{Q}} \pi_2$ *and* $(\pi_1 + \pi_2) \perp_{\mathbb{Q}} \pi_3$*, then* $\pi_2 \perp_{\mathbb{Q}} \pi_3$ *and* $\pi_1 \perp_{\mathbb{Q}} (\pi_2 + \pi_3)$*.*

5.4.2.2 Permission Heaps

The models of our program logic use *permission heaps* to give a semantic meaning to heap ownership. Permission heaps and their heap cells are defined as follows, and are slightly richer than ordinary program heaps (*Heap*), to be able to administer the access permissions and the different ownership types.

Definition 5.4.4 (Permission heap cells, Permission heaps)**.**

$$hc \in \mathit{PermHeapCell} ::= \mathsf{free} \mid \langle v \rangle^{\pi}_{\mathsf{std}} \mid \langle v \rangle^{\pi}_{\mathsf{proc}} \mid \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \mid \mathsf{inv}$$
$$ph \in \mathit{PermHeap} \triangleq \mathit{Val} \to_{\mathsf{fin}} \mathit{PermHeapCell}$$

Permission heaps ph are defined to be total finite functions from values (representing heap locations) to *permission heap cells*, hc, which in turn are inductively defined to be one of the following five elements:

- free, which is an *unoccupied heap cell*.
- $\langle v \rangle^{\pi}_{\mathsf{std}}$, which is a *standard heap cell*, that stores the value $v \in \mathit{Val}$. Standard heap cells are the models of the standard heap ownership predicates, $\overset{\pi}{\hookrightarrow}_{\mathsf{std}}$.
- $\langle v \rangle^{\pi}_{\mathsf{proc}}$, which is a *process heap cell*, that stores the value v. These are used as models of the $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ ownership predicates.
- $\langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}}$, which is an *action heap cell*, that stores the value v_1. Action heap cells are used as the models for the $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ predicates. Moreover, action heap cells store a second value, v_2. This extra value is maintained for technical reasons, to help in establishing soundness of the program logic. The value v_2 is a *snapshot value*: a copy of the original value stored by the heap cell, that is made when an action block was entered.
- inv, which is an *invalid*, or *corrupted*, permission heap cell.

Definition 5.4.5 (Unit permission heap)**.** *The* unit permission heap *is defined to be* $\mathbb{1}_{\mathsf{ph}} \triangleq \lambda v \in \mathit{Val}\,.\,\mathsf{free}$*, containing* free *at every entry.*

Note that, unlike program heaps, permission heaps are defined to be total functions, where the heap cells have an explicit notion of being free (i.e., free). This is done to give permission heaps and their cells nicer algebraic properties.

Furthermore, permission heap cells also have an explicit notion of being invalid. i.e., inv. Invalid heap cells represent the erroneous result of composing two incompatible heap cells. We now define several operations on permission heaps.

Validity. Any permission heap ph is defined to be *valid* if the permissions of all ph's heap cells are valid, where free is always valid and inv is never valid.

Definition 5.4.6 (Validity of permission heaps)**.** *A permission heap* ph *is defined to be* valid*, written* $\mathsf{valid}_{\mathsf{ph}}\, ph$*, if* $\forall v \in \mathit{Val}\,.\,\mathsf{valid}_{\mathsf{hc}}\, ph(v)$*, where:*

$$\mathsf{valid}_{\mathsf{hc}}\, \mathsf{free} \triangleq \mathsf{true} \qquad \mathsf{valid}_{\mathsf{hc}}\, \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \triangleq \mathsf{valid}_{\mathbb{Q}}\, \pi$$
$$\mathsf{valid}_{\mathsf{hc}}\, \langle v \rangle^{\pi}_{\mathsf{std}} \triangleq \mathsf{valid}_{\mathbb{Q}}\, \pi \qquad \mathsf{valid}_{\mathsf{hc}}\, \mathsf{inv} \triangleq \mathsf{false}$$
$$\mathsf{valid}_{\mathsf{hc}}\, \langle v \rangle^{\pi}_{\mathsf{proc}} \triangleq \mathsf{valid}_{\mathbb{Q}}\, \pi$$

Fact 5.4.1. $\mathsf{valid}_{\mathsf{ph}}\, \mathbb{1}_{\mathsf{ph}}$ *holds.*

Disjointness. Two permission heaps ph_1 and ph_2 are *disjoint* if all their heap cells are pairwise *compatible* and their underlying permissions are disjoint.

Definition 5.4.7 (Disjointness of permission heaps)**.** *Two permission heaps, ph_1 and ph_2, are* disjoint*, denoted $ph_1 \perp_{\mathsf{ph}} ph_2$, if $\forall v \in Val\,.\, ph_1(v) \perp_{\mathsf{hc}} ph_2(v)$, where:*

$$\begin{aligned}
\mathsf{free} \perp_{\mathsf{hc}} hc &\triangleq \mathsf{valid}_{\mathsf{hc}}\, hc & \langle v\rangle^{\pi_1}_{\mathsf{proc}} \perp_{\mathsf{hc}} \langle v\rangle^{\pi_2}_{\mathsf{proc}} &\triangleq \pi_1 \perp_{\mathbb{Q}} \pi_2\\
hc \perp_{\mathsf{hc}} \mathsf{free} &\triangleq \mathsf{valid}_{\mathsf{hc}}\, hc & \langle v_1, v_2\rangle^{\pi_1}_{\mathsf{act}} \perp_{\mathsf{hc}} \langle v_1, v_2\rangle^{\pi_2}_{\mathsf{act}} &\triangleq \pi_1 \perp_{\mathbb{Q}} \pi_2\\
\langle v\rangle^{\pi_1}_{\mathsf{std}} \perp_{\mathsf{hc}} \langle v\rangle^{\pi_2}_{\mathsf{std}} &\triangleq \pi_1 \perp_{\mathbb{Q}} \pi_2 & hc_1 \perp_{\mathsf{hc}} hc_2 &\triangleq \mathsf{false}, \textit{ otherwise}
\end{aligned}$$

To elaborate, two permission heap cells are said to be *compatible*, if they have matching ownership types, where free matches with any ownership type other than inv. The intuitive meaning of disjointness, is that any two disjoint permission heaps can safely be composed, that is, without causing any heap cells to be corrupted.

Lemma 5.4.2. *Validity and disjointness of permission heaps satisfy the following basic properties (permission heap cells satisfy the same properties):*

1. *If $\mathsf{valid}_{\mathsf{ph}}\, ph$, then $ph \perp_{\mathsf{ph}} \mathbb{1}_{\mathsf{ph}}$.*
2. *If $ph_1 \perp_{\mathsf{ph}} ph_2$, then also $ph_2 \perp_{\mathsf{ph}} ph_1$ and $\mathsf{valid}_{\mathsf{ph}}\, ph_1$.*

Law 1 is the identity law for $\perp_{\mathsf{ph}}$: any valid permission heap is disjoint with the unit permission heap. Law 2 states that $\perp_{\mathsf{ph}}$ (resp. $\perp_{\mathsf{hc}}$) is symmetric and implies validity of its operands.

Disjoint union. The following operation defines the disjoint union (i.e., the composition) of two permission heaps.

Definition 5.4.8 (Disjoint union of permission heaps)**.** *The disjoint union of any two permission heaps ph_1 and ph_2, written $ph_1 \uplus_{\mathsf{ph}} ph_2$, is defined to be the permission heap $\lambda v \in Val\,.\, ph_1(v) \uplus_{\mathsf{hc}} ph_2(v)$, where $\uplus_{\mathsf{hc}}$ is defined as:*

$$\begin{aligned}
\mathsf{free} \uplus_{\mathsf{hc}} hc &\triangleq hc & \langle v\rangle^{\pi_1}_{\mathsf{proc}} \uplus_{\mathsf{hc}} \langle v\rangle^{\pi_2}_{\mathsf{proc}} &\triangleq \langle v\rangle^{\pi_1+\pi_2}_{\mathsf{proc}}\\
hc \uplus_{\mathsf{hc}} \mathsf{free} &\triangleq hc & \langle v_1, v_2\rangle^{\pi_1}_{\mathsf{act}} \uplus_{\mathsf{hc}} \langle v_1, v_2\rangle^{\pi_2}_{\mathsf{act}} &\triangleq \langle v_1, v_2\rangle^{\pi_1+\pi_2}_{\mathsf{act}}\\
\langle v\rangle^{\pi_1}_{\mathsf{std}} \uplus_{\mathsf{hc}} \langle v\rangle^{\pi_2}_{\mathsf{std}} &\triangleq \langle v\rangle^{\pi_1+\pi_2}_{\mathsf{std}} & hc_1 \uplus_{\mathsf{hc}} hc_2 &\triangleq \mathsf{inv}, \textit{ otherwise}
\end{aligned}$$

Note that $\uplus_{\mathsf{hc}}$ only gives a non-corrupted entry when applied to two compatible heap cells. Furthermore, free is neutral with respect to $\uplus_{\mathsf{hc}}$, while inv is absorbing.

Lemma 5.4.3. *Disjoint union of permission heaps satisfy the following properties (permission heap cells have the same properties):*

1. $ph_1 \uplus_{\mathsf{ph}} (ph_2 \uplus_{\mathsf{ph}} ph_3) = (ph_1 \uplus_{\mathsf{ph}} ph_2) \uplus_{\mathsf{ph}} ph_3$.
2. $ph_1 \uplus_{\mathsf{ph}} ph_2 = ph_2 \uplus_{\mathsf{ph}} ph_1$.
3. *If* $ph \uplus_{\mathsf{ph}} \mathbb{1}_{\mathsf{ph}} = ph$.
4. *If* $ph_1 \perp_{\mathsf{ph}} ph_2$, *then* $\mathsf{valid}_{\mathsf{ph}}\,(ph_1 \uplus_{\mathsf{ph}} ph_2)$.
5. *If* $ph_1 \perp_{\mathsf{ph}} ph_2$ *and* $(ph_1 \uplus_{\mathsf{ph}} ph_2) \perp_{\mathsf{ph}} ph_3$, *then also*
   (a) $ph_2 \perp_{\mathsf{ph}} ph_3$ *and*
   (b) $ph_1 \perp_{\mathsf{ph}} (ph_2 \uplus_{\mathsf{ph}} ph_3)$.

Laws 1–3 describe associativity, commutativity and identity of $\uplus_{\mathsf{ph}}$ (resp. $\uplus_{\mathsf{hc}}$). Law 4 states that $\mathsf{valid}_{\mathsf{ph}}$ is closed under disjoint union, showing that the disjoint union of two disjoint permission heaps is valid, i.e., have valid underlying fractional permissions and do not contain any corrupted heap cells. Law 5 relates disjointness with disjoint union in terms of two distribution laws.

Subheaps. We now define a subheap relation, $\preceq_{\mathsf{ph}}$, on permission heaps. Intuitively, any permission heap ph_1 is said to be a *subheap* of ph_2, if ph_2 stores at least as much information (i.e., values) as ph_1, with at least as much ownership.

Definition 5.4.9 (Permission subheaps)**.** *Any permission heap* ph_1 *is defined to be a* subheap *of* ph_2, *written* $ph_1 \preceq_{\mathsf{ph}} ph_2$, *if* $\forall v \,.\, ph_1(v) \preceq_{\mathsf{hc}} ph_2(v)$, *where:*

$$\mathsf{free} \preceq_{\mathsf{hc}} hc \triangleq \mathsf{true} \qquad \langle v_1, v_2\rangle^{\pi_1}_{\mathsf{act}} \preceq_{\mathsf{hc}} \langle v_1, v_2\rangle^{\pi_2}_{\mathsf{act}} \triangleq \pi_1 \leq_{\mathbb{Q}} \pi_2$$

$$\langle v\rangle^{\pi_1}_{\mathsf{std}} \preceq_{\mathsf{hc}} \langle v\rangle^{\pi_2}_{\mathsf{std}} \triangleq \pi_1 \leq_{\mathbb{Q}} \pi_2 \qquad \mathsf{inv} \preceq_{\mathsf{hc}} \mathsf{inv} \triangleq \mathsf{true}$$

$$\langle v\rangle^{\pi_1}_{\mathsf{proc}} \preceq_{\mathsf{hc}} \langle v\rangle^{\pi_2}_{\mathsf{proc}} \triangleq \pi_1 \leq_{\mathbb{Q}} \pi_2 \qquad hc_1 \preceq_{\mathsf{hc}} hc_2 \triangleq \mathsf{false}, \textit{ otherwise}$$

Here $\leq_{\mathbb{Q}}$ is just the standard $\leq$ ordering on rational numbers. Moreover, one might consider to define $hc_1 \preceq_{\mathsf{hc}} hc_2$ simply as $\exists hc \,.\, hc_1 \perp_{\mathsf{hc}} hc \wedge hc_1 \uplus_{\mathsf{hc}} hc = hc_2$. However, this would not allow $\mathsf{inv} \preceq_{\mathsf{hc}} \mathsf{inv}$ to be true, which is a desirable property.

Lemma 5.4.4. *The permission subheap relation* $\preceq_{\mathsf{ph}}$ *satisfies the following properties (the same properties are satisfied by* $\preceq_{\mathsf{hc}}$*):*

1. $ph \preceq_{\mathsf{ph}} ph$.
2. *If* $ph_1 \preceq_{\mathsf{ph}} ph_2$ *and* $ph_2 \preceq_{\mathsf{ph}} ph_1$, *then* $ph_1 = ph_2$.
3. *If* $ph_1 \preceq_{\mathsf{ph}} ph_2$ *and* $ph_2 \preceq_{\mathsf{ph}} ph_3$, *then* $ph_1 \preceq_{\mathsf{ph}} ph_3$.
4. *If* $ph_1 \perp_{\mathsf{ph}} ph_2$, *then either* $ph_1 \preceq_{\mathsf{ph}} ph_2$ *or* $ph_2 \preceq_{\mathsf{ph}} ph_1$.
5. *If* $ph_1 \preceq_{\mathsf{ph}} ph_2$ *and* $ph_2 \perp_{\mathsf{ph}} ph_3$, *then also*

(a) $ph_1 \preceq_{\mathsf{ph}} (ph_2 \uplus_{\mathsf{ph}} ph_3)$*, and*
(b) $(ph_1 \uplus_{\mathsf{ph}} ph_3) \preceq_{\mathsf{ph}} (ph_2 \uplus_{\mathsf{ph}} ph_3)$.

Laws 1–3 shows that $\preceq_{\mathsf{ph}}$ is a partial order, by being reflexive, antisymmetric and transitive. In fact, $\preceq_{\mathsf{ph}}$ is a total order with respect to *disjoint* permission heaps, as shown by Law 4, which states connexity. Law 5 expresses monotonicity, and shows that disjoint union preserves the subheap ordering.

Entirety. Finally, we define a notion of *entirety* of permission heaps and their cells. A permission heap ph is said to be *entire*, or *full*, if all ph's permission heap cells are full. In turn, any permission heap cell hc is *full*, if hc is occupied and has an associated fractional permission π that is 1.

Definition 5.4.10 (Permission heap entirety)**.** *Any permission heap ph is defined to be* entire*, or* full*, written* $\mathsf{full}_{\mathsf{ph}}\, ph$*, if* $\forall v \in \mathit{Val}\,.\,\mathsf{full}_{\mathsf{hc}}\, ph(v)$*, where the* $\mathsf{full}_{\mathsf{hc}}$ *predicate is inductively defined as follows:*

$$\mathsf{full}_{\mathsf{hc}}\, \langle v \rangle^{1}_{\mathsf{std}} \qquad \mathsf{full}_{\mathsf{hc}}\, \langle v \rangle^{1}_{\mathsf{proc}} \qquad \mathsf{full}_{\mathsf{hc}}\, \langle v_1, v_2 \rangle^{1}_{\mathsf{act}}$$

Lemma 5.4.5. *Permission heap cell entirety satisfies the following properties:*

1. *If* $\mathsf{full}_{\mathsf{hc}}\, hc$*, then* $\mathsf{valid}_{\mathsf{hc}}\, hc_1$ *(and likewise of permission heaps).*
2. *If* $\mathsf{full}_{\mathsf{hc}}\, hc_1$ *and* $hc_1 \perp_{\mathsf{hc}} hc_2$*, then* $hc_2 = \mathsf{free}$.
3. *If* $\mathsf{full}_{\mathsf{hc}}\, hc_1$ *and* $\mathsf{valid}_{\mathsf{hc}}\, hc_2$ *and* $hc_1 \preceq_{\mathsf{hc}} hc_2$*, then* $hc_1 = hc_2$.

Law 1 in the above lemma states that any full permission heap (cell) must also be valid. Moreover, Law 2 states that full permission heap cells can only be disjoint with *un*occupied heap cells, whereas Law 3 expresses that full heap cells cannot be extended without losing their validity.

5.4.2.3 Process Maps

Apart from permission heaps, the models of our logic also use *process maps*, to give a semantic meaning to process ownership predicates in the logic, i.e., Proc_{π}.

Process maps and their entries are defined as follows.

Definition 5.4.11 (Process map entries, Process maps)**.**

$$mc \in \mathit{ProcMapEntry} ::= \mathsf{free} \mid \langle p, P, \Lambda \rangle^{\pi} \mid \mathsf{inv}$$
$$pm \in \mathit{ProcMap} \triangleq \mathit{Val} \rightarrow_{\mathsf{fin}} \mathit{ProcMapEntry}$$

where $\Lambda \in \mathit{Binder} \triangleq \mathit{ProcVar} \rightharpoonup_{\mathsf{fin}} \mathit{Val}$ *is a finite partial mapping from process variables to values (which are intended to be heap locations).*

Process maps *pm* are defined to be total mappings from values (representing identifiers of program abstractions) to *process map entries*, *mc*, which are, in turn, inductively defined to one of the following three elements:

- free, which models an *unoccupied*, or *free*, entry in *pm*.
- $\langle p, P, \Lambda \rangle^{\pi}$, which is an *occupied* process map entry, that is used as a model for the $\mathsf{Proc}_{\pi}(X, p, P, \Pi)$ predicates in the logic (where the variable X is meant to identify this entry, inside *pm*). Furthermore, the Λ inside the entry is a *binder*, which is a model for the Π component of the Proc_{π} predicate.
- inv, which denotes an *invalid*, or *corrupted*, process map entry.

Definition 5.4.12 (Unit process map)**.** *The* unit process map *is defined to be* $\mathbb{1}_{\mathsf{pm}} \triangleq \lambda v \in \mathit{Val} \,.\, \mathsf{free}$*, containing* free *at every entry.*

Likewise to permission heaps, process maps are defined as total finite functions, with entries that can explicitly be unoccupied (free) or invalid (inv), as this provides desirable algebraic properties. Corrupted entries represent the erroneous result of taking the disjoint union of two incompatible, non-disjoint entries.

We now define several operations and relations on process maps, most of which are similar in spirit to the operations we defined earlier, for permission heaps.

Bisimilarity. Two process maps are said to be *bisimilar*, if all their entries are pairwise equal, or contain occupied entries with process components that are pairwise bisimilar. This is formally captured by the following definition.

Definition 5.4.13 (Process map bisimilarity)**.** Bisimilarity *of two process maps* pm_1 *and* pm_2 *is defined to be* $pm_1 \cong_{\mathsf{pm}} pm_2 \triangleq \forall v \,.\, pm_1(v) \cong_{\mathsf{mc}} pm_2(v)$*, where bisimilarity of process map entries,* $\cong_{\mathsf{mc}}$*, is inductively defined as follows.*

$$\mathsf{free} \cong_{\mathsf{mc}} \mathsf{free} \qquad \frac{P_1 \cong P_2}{\langle p, P_1, \Lambda \rangle^{\pi} \cong_{\mathsf{mc}} \langle p, P_2, \Lambda \rangle^{\pi}} \qquad \mathsf{inv} \cong_{\mathsf{mc}} \mathsf{inv}$$

We will later see that the program logic always allows replacing processes P inside $\mathsf{Proc}_{\pi}(X, p, P, \Pi)$ predicates by bisimilar ones. To handle such replacements at the semantic level, we allow process maps and their entries to be handled up to $\cong_{\mathsf{pm}}$ and $\cong_{\mathsf{mc}}$, respectively, with help of the following result.

Lemma 5.4.6. *Both* $\cong_{\mathsf{pm}}$ *and* $\cong_{\mathsf{mc}}$ *are equivalence relations.*

Validity. Any process map pm is *valid*, if none of pm's entries are corrupt, and moreover, all occupied entries of pm have a valid associated fractional permission.

Definition 5.4.14 (Process map validity)**.** Validity *of process maps pm is defined as* $\mathsf{valid}_{\mathsf{pm}}\, pm \triangleq \forall v\,.\, \mathsf{valid}_{\mathsf{mc}}\, pm(v)$*, where* $\mathsf{valid}_{\mathsf{mc}}$*, in turn, is defined as follows.*

$$\mathsf{valid}_{\mathsf{mc}}\, \mathsf{free} \triangleq \mathsf{true} \qquad \mathsf{valid}_{\mathsf{mc}}\, \langle p, P, \Lambda\rangle^{\pi} \triangleq \mathsf{valid}_{\mathbb{Q}}\, \pi \qquad \mathsf{valid}_{\mathsf{mc}}\, \mathsf{inv} \triangleq \mathsf{false}$$

It is not difficult to deduce from the above definition that $\mathbb{1}_{\mathsf{pm}}$ is a valid process map, and moreover, that bisimilarity is validity-preserving.

Fact 5.4.2. $\mathsf{valid}_{\mathsf{pm}}\, \mathbb{1}_{\mathsf{pm}}$ *holds.*

Lemma 5.4.7. *The relations* $\cong_{\mathsf{pm}}$ *and* $\cong_{\mathsf{mc}}$ *both preserve validity:*

1. *If* $\mathsf{valid}_{\mathsf{pm}}\, pm_1$ *and* $pm_1 \cong_{\mathsf{pm}} pm_2$*, then* $\mathsf{valid}_{\mathsf{pm}}\, pm_2$*.*
2. *If* $\mathsf{valid}_{\mathsf{mc}}\, mc_1$ *and* $mc_1 \cong_{\mathsf{mc}} mc_2$*, then* $\mathsf{valid}_{\mathsf{mc}}\, mc_2$*.*

Disjointness. Intuitively, two process maps are said to be *disjoint* if none of their entries are corrupt, and all fractional permissions of their entries are pairwise disjoint. This notion of disjointness is formally captured by the following definition.

Definition 5.4.15 (Disjointness of process maps)**.** Disjointness *of two process maps pm_1 and pm_2 is defined as* $pm_1 \perp_{\mathsf{pm}} pm_2 \triangleq \forall v\,.\, pm_1(v) \perp_{\mathsf{mc}} pm_2(v)$*, where disjointness* $\perp_{\mathsf{mc}}$ *of process map entries is defined as follows.*

$$\mathsf{free} \perp_{\mathsf{mc}} mc \triangleq \mathsf{valid}_{\mathsf{mc}}\, mc \qquad \langle p, P_1, \Lambda\rangle^{\pi_1} \perp_{\mathsf{mc}} \langle p, P_2, \Lambda\rangle^{\pi_2} \triangleq \pi_1 \perp_{\mathbb{Q}} \pi_2$$
$$mc \perp_{\mathsf{mc}} \mathsf{free} \triangleq \mathsf{valid}_{\mathsf{mc}}\, mc \qquad mc_1 \perp_{\mathsf{mc}} mc_2 \triangleq \mathsf{false}\text{, otherwise}$$

The intuitive meaning of disjointness $\perp_{\mathsf{pm}}$, is that disjoint process maps can safely be composed without causing any of the resulting entries to be corrupted.

Before giving the definition of composition (disjoint union), let us first discuss some properties of disjointness of process maps and their entries.

Lemma 5.4.8. *Process map disjointness satisfies the following properties (these properties also hold for process map entries):*

1. *If* $\mathsf{valid}_{\mathsf{pm}}\, pm$*, then* $pm \perp_{\mathsf{pm}} \mathbb{1}_{\mathsf{pm}}$*.*
2. *If* $pm_1 \perp_{\mathsf{pm}} pm_2$*, then also* $pm_2 \perp_{\mathsf{pm}} pm_1$ *and* $\mathsf{valid}_{\mathsf{pm}}\, pm_1$*.*
3. *If* $pm_1 \perp_{\mathsf{pm}} pm_2$ *and* $pm_1 \cong_{\mathsf{pm}} pm_1'$ *and* $pm_2 \cong_{\mathsf{pm}} pm_2'$*, then* $pm_1' \perp_{\mathsf{pm}} pm_2'$*.*

Laws 1 and 2 are very similar to the properties of permission heap disjointness that we gave earlier. Law 3 states that bisimilarity preserves disjointness.

Disjoint union. The following operation defines the disjoint union (composition) of two process map (entries).

Definition 5.4.16 (Disjoint union of process maps)**.** *The* disjoint union *of two process maps pm_1 and pm_2 is defined as $pm_1 \uplus_{\mathsf{pm}} pm_2 \triangleq \forall v \,.\, pm_1(v) \uplus_{\mathsf{mc}} pm_2(v)$, with $\uplus_{\mathsf{mc}}$ defined as follows.*

$$\begin{aligned}
\mathsf{free} \uplus_{\mathsf{mc}} mc &\triangleq mc \\
mc \uplus_{\mathsf{mc}} \mathsf{free} &\triangleq mc \\
\langle p, P_1, \Lambda\rangle^{\pi_1} \uplus_{\mathsf{mc}} \langle p, P_2, \Lambda\rangle^{\pi_2} &\triangleq \langle p, P_1 \parallel P_2, \Lambda\rangle^{\pi_1+\pi_2} \\
mc_1 \uplus_{\mathsf{mc}} mc_2 &\triangleq \mathsf{inv}, \textit{ otherwise}
\end{aligned}$$

Likewise to disjoint union of permission heaps, the composition of incompatible process map entries produces a corrupted entry (inv). The entry free is again neutral, whereas inv is absorbing (i.e., composing a corrupted entry with any entry again yields a corrupted entry).

Lemma 5.4.9. *Disjoint union of process maps $\uplus_{\mathsf{pm}}$ and their entries $\uplus_{\mathsf{mc}}$ satisfy the following properties (we only show them for $\uplus_{\mathsf{pm}}$ however):*

1. *If $pm_1 \cong_{\mathsf{pm}} pm_2$ and $pm_1' \cong_{\mathsf{pm}} pm_2'$, then $pm_1 \uplus_{\mathsf{pm}} pm_1' \cong_{\mathsf{pm}} pm_2 \uplus_{\mathsf{pm}} pm_2'$.*
2. *$pm_1 \uplus_{\mathsf{pm}} (pm_2 \uplus_{\mathsf{pm}} pm_3) = (pm_1 \uplus_{\mathsf{pm}} pm_2) \uplus_{\mathsf{pm}} pm_3$.*
3. *$pm_1 \uplus_{\mathsf{pm}} pm_2 = pm_2 \uplus_{\mathsf{pm}} pm_1$.*
4. *$pm \uplus_{\mathsf{pm}} \mathbb{1}_{\mathsf{pm}} = pm$.*
5. *If $pm_1 \perp_{\mathsf{pm}} pm_2$, then $\mathsf{valid}_{\mathsf{pm}}(pm_1 \uplus_{\mathsf{pm}} pm_2)$.*
6. *If $pm_1 \perp_{\mathsf{pm}} pm_2$ and $(pm_1 \uplus_{\mathsf{pm}} pm_2) \perp_{\mathsf{pm}} pm_3$, then also*
   (a) *$pm_2 \perp_{\mathsf{pm}} pm_3$, and*
   (b) *$pm_1 \perp_{\mathsf{pm}} (pm_2 \uplus_{\mathsf{pm}} pm_3)$.*

Law 1 states that $\cong_{\mathsf{pm}}$ is a *congruence* with respect to $\uplus_{\mathsf{pm}}$ (and likewise for $\cong_{\mathsf{mc}}$ and $\uplus_{\mathsf{mc}}$). Laws 2–3 express that $\cong_{\mathsf{pm}}$ and $\cong_{\mathsf{mc}}$ are associative and commutative, respectively. Law 4 shows that $\mathbb{1}_{\mathsf{pm}}$ (resp. free) is indeed a unit element with respect to $\uplus_{\mathsf{pm}}$ (resp. $\uplus_{\mathsf{mc}}$). Law 5 ensures that the disjoint union of two disjoint process maps is indeed valid (i.e, uncorrupted) Finally, Law 6 relates process map (entry) disjointness with disjoint union in terms of two distribution properties.

Submaps. We now define a submap ordering on process maps and their entries, likewise to the subheap relation of permission heaps.

Definition 5.4.17 (Process submaps)**.** *Any process map* pm_1 *is defined to be a* submap *of* pm_2*, denotes* $pm_1 \preceq_{\mathsf{pm}} pm_2$*, if* $\forall v \,.\, pm_1(v) \preceq_{\mathsf{mc}} pm_2(v)$*, such that:*

$$\mathsf{free} \preceq_{\mathsf{mc}} mc \qquad \frac{\pi_1 \leq_{\mathbb{Q}} \pi_2 \qquad P_2 \cong P_1 \parallel P_1'}{\langle p, P_1, \Lambda\rangle^{\pi_1} \preceq_{\mathsf{mc}} \langle p, P_2, \Lambda\rangle^{\pi_2}} \qquad \mathsf{inv} \preceq_{\mathsf{mc}} \mathsf{inv}$$

Observe that in the case of occupied entries, i.e., $\langle p, P_1, \Lambda\rangle^{\pi_1} \preceq_{\mathsf{mc}} \langle p, P_2, \Lambda\rangle^{\pi_2}$, the process P_2 is required to have *at least* all the behaviours that P_1 has. This is expressed by the premise $P_2 \cong P_1 \parallel P_1'$, where P_1' is existentially quantified.

Likewise to the subheap ordering, $\preceq_{\mathsf{pm}}$ and $\preceq_{\mathsf{mc}}$ are partial orders, as well as total orders with respect to disjoint process map (entries). Furthermore, bisimilarity and disjoint union are order preserving, as shown by the following lemma.

Lemma 5.4.10. *The relations* $\preceq_{\mathsf{pm}}$ *and* $\preceq_{\mathsf{mc}}$ *satisfy the following properties (for ease of presentation, only the ones for* $\preceq_{\mathsf{pm}}$ *are given):*

1. $pm \preceq_{\mathsf{pm}} pm$.
2. *If* $pm_1 \preceq_{\mathsf{pm}} pm_2$ *and* $pm_2 \preceq_{\mathsf{pm}} pm_1$*, then* $pm_1 = pm_2$.
3. *If* $pm_1 \preceq_{\mathsf{pm}} pm_2$ *and* $pm_2 \preceq_{\mathsf{pm}} pm_3$*, then* $pm_1 \preceq_{\mathsf{pm}} pm_3$.
4. *If* $pm_1 \perp_{\mathsf{pm}} pm_2$*, then either* $pm_1 \preceq_{\mathsf{pm}} pm_2$ *or* $pm_2 \preceq_{\mathsf{pm}} pm_1$.
5. *If* $pm_1 \cong_{\mathsf{pm}} pm_1'$ *and* $pm_2 \cong_{\mathsf{pm}} pm_2'$ *and* $pm_1 \preceq_{\mathsf{pm}} pm_2$*, then* $pm_1' \preceq_{\mathsf{pm}} pm_2'$.
6. *If* $pm_1 \preceq_{\mathsf{pm}} pm_2$ *and* $pm_2 \perp_{\mathsf{pm}} pm_3$*, then also*
   (a) $pm_1 \preceq_{\mathsf{pm}} (pm_2 \uplus_{\mathsf{pm}} pm_3)$*, as well as*
   (b) $(pm_1 \uplus_{\mathsf{pm}} pm_3) \preceq_{\mathsf{pm}} (pm_2 \uplus_{\mathsf{pm}} pm_3)$.

5.4.2.4 Worlds

Finally, we introduce *worlds*, which are the models of our program logic. Worlds $\mathfrak{M}$ are defined to be quadruples (ph, pm, s, g) consisting of a permission heap ph, a process map pm, and two stores, s and g, that give an interpretation to program variables and "ghost" variables, respectively. With ghost variables we mean any variable that is used to represent a program abstraction. For example, the variable X in any $X :=$ **process** p **over** Π command is considered to be a ghost variable. The extra store, g, is therefore referred to as the *ghost store*.

Definition 5.4.18 (Worlds)**.** *The domain of* worlds *is defined to be:*

$$\mathfrak{M} \in \mathit{World} \triangleq \mathit{PermHeap} \times \mathit{ProcMap} \times \mathit{Store} \times \mathit{Store}$$

Definition 5.4.19 (Unit world)**.** *The* unit world $\mathbb{1}_W \in World$ *is defined to be* $\mathbb{1}_W \triangleq (\mathbb{1}_{\mathsf{ph}}, \mathbb{1}_{\mathsf{pm}}, \lambda X.\mathbb{1}_{Val}, \lambda X.\mathbb{1}_{Val})$*, where* $\mathbb{1}_{Val}$ *is assumed to be the unit element in the domain Val of values.*

Moreover, we simply lift the notions of bisimilarity, validity, disjointness, disjoint union, and subheaps/maps, to worlds, in the following ways.

Definition 5.4.20. *Let* $\mathfrak{W}_i = (ph_i, pm_i, s_i, g_i)$ *for* $i \in \{1, 2\}$*. Then:*

$$\begin{aligned}
\mathfrak{W}_1 \cong_W \mathfrak{W}_2 &\triangleq ph_1 = ph_2 \wedge pm_1 \cong_{\mathsf{pm}} pm_2 \wedge s_1 = s_2 \wedge g_1 = g_2 \\
\mathsf{valid}_W \mathfrak{W} &\triangleq \mathsf{valid}_{\mathsf{ph}}\, ph \wedge \mathsf{valid}_{\mathsf{pm}}\, pm \\
\mathfrak{W}_1 \perp_W \mathfrak{W}_2 &\triangleq ph_1 \perp_{\mathsf{ph}} ph_2 \wedge pm_1 \perp_{\mathsf{pm}} pm_2 \\
\mathfrak{W}_1 \uplus_W \mathfrak{W}_2 &\triangleq (ph_1 \uplus_{\mathsf{ph}} ph_2,\ pm_1 \uplus_{\mathsf{pm}} pm_2,\ s_1,\ g_1) \\
\mathfrak{W}_1 \preceq_W \mathfrak{W}_2 &\triangleq ph_1 \preceq_{\mathsf{ph}} ph_2 \wedge pm_1 \preceq_{\mathsf{pm}} pm_2 \wedge s_1 = s_2 \wedge g_1 = g_2
\end{aligned}$$

These operations have the same properties as the ones of permission heaps and process maps as defined in §5.4.2.2 and §5.4.2.3. We do not repeat them here.

5.4.3 Semantics of Assertions

Let us now define the interpretation of assertions. The semantics of assertions is defined in terms of a satisfaction relation $\mathfrak{M} \models \mathcal{P}$, stating that the assertion $\mathcal{P}$ is satisfied by the model $\mathfrak{M}$. However, its definition depends on the following operation, $[\![\Pi]\!]s$, for evaluating abstraction binders Π.

Definition 5.4.21 (Abstraction binder evaluation)**.** *The evaluation of abstraction binders is defined in terms of the function* $[\![\cdot]\!] : AbstrBinder \to Store \to Binder$*, in the following way:*

$$[\![\{x_0 \mapsto E_0, \ldots, x_n \mapsto E_n\}]\!]s \triangleq \lambda x\,.\, \begin{cases} [\![E_i]\!]s & \text{if } x = x_i \text{ for some } 0 \leq i < n \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 5.4.22 (Semantics of assertions)**.** *The modelling relation* $\mathfrak{M} \models \mathcal{P}$ *is defined by structural recursion on* $\mathcal{P}$*, in the following way, with* $\mathfrak{M} = (ph, pm, s, g)$*:*

$$\begin{aligned}
\mathfrak{M} \models B \quad &\text{iff} \quad [\![B]\!]s \\
\mathfrak{M} \models \forall X.\mathcal{P} \quad &\text{iff} \quad \forall v\,.\,(ph, pm, s[X \mapsto v], g[X \mapsto v]) \models \mathcal{P} \\
\mathfrak{M} \models \exists X.\mathcal{P} \quad &\text{iff} \quad \exists v\,.\,(ph, pm, s[X \mapsto v], g[X \mapsto v]) \models \mathcal{P} \\
\mathfrak{M} \models \mathcal{P} \vee \mathcal{Q} \quad &\text{iff} \quad \mathfrak{M} \models \mathcal{P} \vee \mathfrak{M} \models \mathcal{Q} \\
\mathfrak{M} \models \mathcal{P} * \mathcal{Q} \quad &\text{iff} \quad \exists \mathfrak{M}_1, \mathfrak{M}_2\,.\,\mathfrak{M}_1 \perp_W \mathfrak{M}_2 \wedge \mathfrak{M}_1 \uplus_W \mathfrak{M}_2 \cong_W \mathfrak{M} \wedge
\end{aligned}$$

$$\mathfrak{M}_1 \models \mathcal{P} \wedge \mathfrak{M}_2 \models \mathcal{Q}$$

$$\begin{array}{rcl}
\mathfrak{M} \models \mathcal{P} \mathrel{-\!\!*} \mathcal{Q} & \textit{iff} & \forall \mathfrak{M}'.\, (\mathfrak{M} \perp_W \mathfrak{M}' \wedge \mathfrak{M}' \models \mathcal{P}) \implies \mathfrak{M} \uplus_W \mathfrak{M}' \models \mathcal{Q} \\
\mathfrak{M} \models *_{i \in I} \mathcal{P}_i & \textit{iff} & \mathfrak{M} \models \mathcal{P}_{i_0} * \cdots * \mathcal{P}_{i_n} \textit{ for } I = \{i_0, \ldots, i_n\} \\
\mathfrak{M} \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E_2 & \textit{iff} & \langle [\![E_2]\!] s \rangle^{\pi}_{\mathsf{std}} \preceq_{\mathsf{hc}} ph([\![E_1]\!] s) \\
\mathfrak{M} \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E_2 & \textit{iff} & \langle [\![E_2]\!] s \rangle^{\pi}_{\mathsf{proc}} \preceq_{\mathsf{hc}} ph([\![E_1]\!] s) \\
\mathfrak{M} \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E_2 & \textit{iff} & \exists v\,.\, \langle [\![E_2]\!] s, v \rangle^{\pi}_{\mathsf{act}} \preceq_{\mathsf{hc}} ph([\![E_1]\!] s) \\
\mathfrak{M} \models \mathsf{Proc}_{\pi}(X, p, P, \Pi) & \textit{iff} & \langle p, P, [\![\Pi]\!] s \rangle^{\pi} \preceq_{\mathsf{mc}} pm(g(X))
\end{array}$$

Clarifying the semantic meaning of assertions, the separating conjunction $\mathcal{P}_1 * \mathcal{P}_2$ is satisfied by a world $\mathfrak{M}$, if $\mathfrak{M}$ can be partitioned into two disjoint worlds, $\mathfrak{M}_1$ and $\mathfrak{M}_2$, using $\uplus_W$, such that $\mathfrak{M}_1$ satisfies $\mathcal{P}_1$ and $\mathfrak{M}_2$ satisfies $\mathcal{P}_2$. Magic wands $\mathcal{P}_1 \mathrel{-\!\!*} \mathcal{P}_2$ are satisfied by a world $\mathfrak{M}$, if, for any disjoint extension $\mathfrak{M}'$ of $\mathfrak{M}$ that satisfies $\mathcal{P}_1$, the extended world $\mathfrak{M} \uplus_W \mathfrak{M}'$ satisfies $\mathcal{P}_2$. Moreover, the semantic meaning of iterated separating conjunctions, $*_{i \in I} \mathcal{P}_i$, can be expressed simply in terms of the interpretation of the binary separating conjunction, $*$.

Moving to the non-standard connectives; heap ownership assertions $E \overset{\pi}{\hookrightarrow}_t E'$ are satisfied if the permission heap holds an entry at location E that matches with the ownership type t, with an associated fractional permission that is at least π. Process ownership assertions $\mathsf{Proc}_{\pi}(X, p, P, \Pi)$ are satisfied if the process map holds a matching entry at position $g(X)$ with a fractional permission at least π, and a process that at least includes all the behaviours of the process P.

Lemma 5.4.11. *The $\models$ modelling relation satisfies the following properties:*

1. *If $\mathfrak{M}_1 \preceq_W \mathfrak{M}_2$ and $\mathsf{valid}_W \mathfrak{W}_2$, then $\mathfrak{M}_1 \models \mathcal{P}$ implies $\mathfrak{M}_2 \models \mathcal{P}$.*
2. *If $\mathfrak{M}_1 \cong_W \mathfrak{M}_2$, then $\mathfrak{M}_1 \models \mathcal{P}$ implies $\mathfrak{M}_2 \models \mathcal{P}$.*

Property 1 in the above lemma expresses monotonicity, and states that adding resources does not invalidate the satisfiability of an assertion (i.e., adding more resources only makes the assertion "more true"). This is a key property of *intuitionistic* separation logic. Moreover, Property 2 is essential for allowing process-algebraic abstractions to be replaced by bisimilar ones inside the program logic.

5.4.3.1 Semantic Entailment

Let the denotation $[\![\mathcal{P}]\!] \triangleq \{\mathfrak{W} \mid \mathfrak{W} \models \mathcal{P}\} \subseteq \mathit{World}$ be the set of all models that are satisfied the assertion $\mathcal{P}$. As one would expect, we have that $[\![\mathsf{true}]\!] = \mathit{World}$, $[\![\mathsf{false}]\!] = \emptyset$, and $[\![\mathcal{P} \vee \mathcal{Q}]\!] = [\![\mathcal{P}]\!] \cup [\![\mathcal{Q}]\!]$. Given any two assertions $\mathcal{P}$ and $\mathcal{Q}$, the assertion $\mathcal{P}$ is defined to *semantically entail* $\mathcal{Q}$, denoted as $\mathcal{P} \models \mathcal{Q}$, if every model of $\mathcal{P}$ is also a model of $\mathcal{Q}$. This can now concisely be expressed via set inclusion.

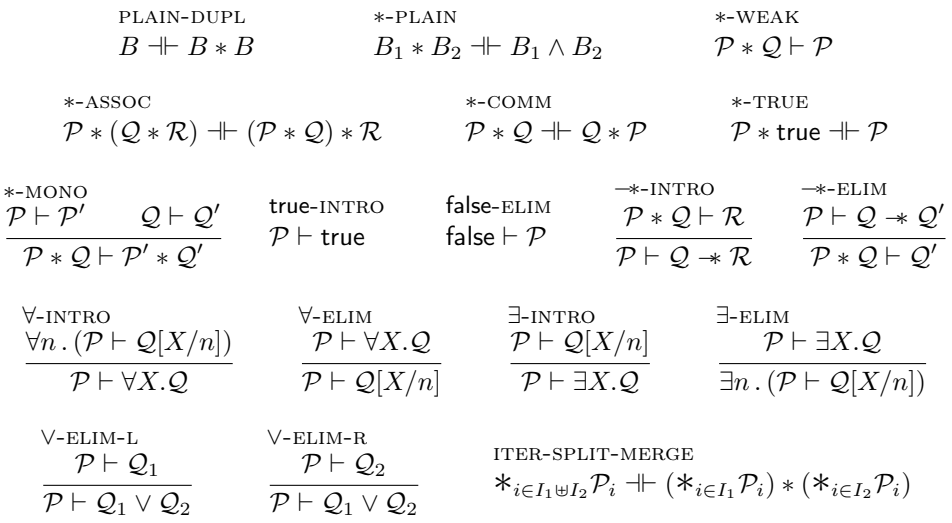


Figure 5.2: The primitive entailment rules of our program logic.

Definition 5.4.23 (Semantic entailment)**.** $\mathcal{P} \models \mathcal{Q} \triangleq [\![\mathcal{P}]\!] \subseteq [\![\mathcal{Q}]\!]$.

Consequently, semantic entailment $\models$ is a preorder (i.e., reflexive and transitive), as well as a congruence for all connectives of the assertion language.

5.5 Proof System

This section introduces the proof system of our model-based verification technique, which consists of structural proof rules (§5.5.1) as well as Hoare proof rules (§5.5.2). This proof system essentially extends the CSL of [Vaf11], by adding permission accounting [Boy03, BCOP05] and machinery for handling process-algebraic program abstractions.

5.5.1 Entailment Rules

Figure 5.2 shows the standard structural rules of the program logic. The notation $\mathcal{P} \dashv\vdash \mathcal{Q}$ is a shorthand notation for both $\mathcal{P} \vdash \mathcal{Q}$ and $\mathcal{Q} \vdash \mathcal{P}$, and indicates that the rule can be used in both directions.

Clarifying the rules, PLAIN-DUPL expresses that plain expressions can freely be duplicated, whereas $*$-PLAIN shows that $*$ has the same meaning as $\wedge$ in the case of plain assertions. The rule $*$-WEAK shows that our concurrent separation logic

$$\frac{\pi_1 \perp_{\mathbb{Q}} \pi_2}{E_1 \overset{\pi_1+\pi_2}{\hookrightarrow}_t E_2 \dashv\vdash E_1 \overset{\pi_1}{\hookrightarrow}_t E_2 * E_1 \overset{\pi_2}{\hookrightarrow}_t E_2} \hookrightarrow\text{-SPLIT-MERGE} \qquad \frac{t_1 \neq t_2}{E \overset{\pi_1}{\hookrightarrow}_{t_1} E' * E \overset{\pi_2}{\hookrightarrow}_{t_2} E' \vdash \mathsf{false}} \hookrightarrow\text{-INCOMPATIBLE}$$

$$\frac{\forall i \in I_1 \,.\, \pi_i < 1 \qquad \forall i \in I_2 \,.\, \pi_i = 1}{\ast_{i \in I_1 \uplus I_2} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{procact}} E_i' \dashv\vdash \left(\ast_{i \in I_1} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i'\right) * \left(\ast_{i \in I_2} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{act}} E_i'\right)} *\text{-PROCACT-SPLIT-MERGE}$$

$$\frac{P \cong Q}{\mathsf{Proc}_\pi(X, p, P, \Pi) \dashv\vdash \mathsf{Proc}_\pi(X, p, Q, \Pi)} \mathsf{Proc}\text{-}\cong$$

$$\frac{\pi_1 \perp_{\mathbb{Q}} \pi_2}{\mathsf{Proc}_{\pi_1+\pi_2}(X, p, P_1 \parallel P_2, \Pi) \dashv\vdash \mathsf{Proc}_{\pi_1}(X, p, P_1, \Pi) * \mathsf{Proc}_{\pi_2}(X, p, P_2, \Pi)} \mathsf{Proc}\text{-SPLIT-MERGE}$$

Figure 5.3: The extended entailment rules for heap and process ownership.

is *affine* (i.e., intuitionistic), by allowing to throw away (forget about) resources. The rules $*$-ASSOC and $*$-COMM express that the separating conjunction is associative and commutative, respectively, whereas $*$-TRUE allows any resource to be composed with true. The rule true-INTRO is the introduction rule for true, while false-ELIM is the elimination rule for false, stating that anything can be derived from falsehood. The $-\!\!*$-INTRO and $-\!\!*$-ELIM rules show that magic wands can be used similarly to the modus ponens inference rule of propositional logic, with respect to $*$. The rules $\forall$-INTRO, $\forall$-ELIM, $\exists$-INTRO and $\exists$-ELIM are the standard introduction and elimination rules for universal and existential quantifiers. Rules $\vee$-ELIM-L and $\vee$-ELIM-R allow to eliminate one of the operands of a disjunction $\vee$. Finally, ITER-SPLIT-MERGE allows iterated separating conjunctions to be split and merged.

Figure 5.2 shows our new entailment rules, that deal with heap ownership and ownership of process-algebraic models.

The rule $\hookrightarrow$-SPLIT-MERGE expresses that heap ownership predicates $\cdot \overset{\pi}{\hookrightarrow}_t \cdot$ of any type t may be *split* (in the left-to-right direction) as well as be *merged* (right-to-left direction) along π. Note however, that multiple points-to predicates for the same heap location may only co-exist if they have the same ownership type, as indicated by the $\hookrightarrow$-INCOMPATIBLE rule. Moreover, the $*$-PROCACT-SPLIT-MERGE inference rule states that iterated procact heap ownership predicates can be split into disjoint iterated proc and act predicates, or be merged into one such iteration.

The Proc-$\cong$ rule allows process-algebraic abstractions to be replaced by bisimilar ones. Finally, Proc-SPLIT-MERGE allows splitting and merging process ownership predicates in the same style as $\overset{\pi}{\hookrightarrow}_t$, to distribute parallel processes over parallel threads. Notably, by splitting a predicate $\mathsf{Proc}_{\pi_1+\pi_2}(X, p, P_1 \parallel P_2, \Pi)$ into two, both parts can be distributed over different concurrent threads in the program logic, so that thread i can establish that it executes as prescribed by its part $\mathsf{Proc}_{\pi_i}(X, p, P_i, \Pi)$ of the abstract model. Afterwards, when the threads join again, the remaining partial abstractions can be merged back into a single $\mathsf{Proc}_{\pi_1+\pi_2}$ predicate. This system of splitting and merging provides a compositional, thread-modular way of verifying that programs meet their abstraction. The logical machinery of this is further discussed in the next section.

All the entailment rules presented in Figure 5.2 and Figure 5.3 are sound in the standard sense.

Theorem 5.5.1 (Soundness of the entailment rules)**.** *$\mathcal{P} \vdash \mathcal{Q}$ implies $\mathcal{P} \models \mathcal{Q}$.*

5.5.2 Program Judgments

We now define *program judgments*, and give the Hoare proof rules of the program logic. Judgments of programs are sequents (quintuples) $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$. The right-hand side is a traditional Hoare triple, whereas $\mathcal{R}$ is a *resource invariant* that is used to handle atomic programs, and Γ an environment in the style of interface specifications of [OYR04]. Such *process environments* are defined as follows.

Definition 5.5.1 (Process environments)**.**

$$\Gamma \in \mathit{ProcEnv} \quad ::= \quad \emptyset \mid \Gamma, \{b\}\, p\, \{b\}$$

Process environments describe assumptions as Hoare-triples $\{b_1\}\, p\, \{b_2\}$ for process-algebraic models, where p identifies the process declaration. These Hoare triples constitute the contracts of the process-algebraic abstractions that are defined for the program that is to be verified. In particular, they allow for assume-guarantee style reasoning: these process Hoare triples may be *assumed* in the proof system while dealing with (finalising) process-algebraic models, and must be *guaranteed* externally, for example via model checking, e.g., using mCRL2.

Standard rules. Figure 5.4 gives the standard proof rules of our logic. These are essentially the same as the proof rules of classical CSL [Vaf11]. One minor difference is that the HT-ATOMIC leaves true instead of emp[2], while using a resource

[2]The assertion language of the classical version of CSL contains an extra emp construct, for explicitly denoting that the heap is empty. In our intuitionistic version of the logic, resources

Standard structural rules

HT-FRAME
$$\frac{\mathsf{fv}(\mathcal{F}) \cap \mathsf{mod}(C) = \emptyset \qquad \Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P} * \mathcal{F}\}\, C\, \{\mathcal{Q} * \mathcal{F}\}}$$

HT-CONSEQ
$$\frac{\mathcal{P} \vdash \mathcal{P}' \qquad \Gamma; \mathcal{R} \vdash \{\mathcal{P}'\}\, C\, \{\mathcal{Q}'\} \qquad \mathcal{Q}' \vdash \mathcal{Q}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}}$$

HT-SHARE
$$\frac{\Gamma; \mathcal{R} * \mathcal{R}' \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P} * \mathcal{R}'\}\, C\, \{\mathcal{Q} * \mathcal{R}'\}}$$

HT-DISJ
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P}_1\}\, C\, \{\mathcal{Q}_1\} \qquad \Gamma; \mathcal{R} \vdash \{\mathcal{P}_2\}\, C\, \{\mathcal{Q}_2\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}_1 \vee \mathcal{P}_2\}\, C\, \{\mathcal{Q}_1 \vee \mathcal{Q}_2\}}$$

HT-EX
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \qquad X \notin \mathsf{fv}(C)}{\Gamma; \mathcal{R} \vdash \{\exists X.\mathcal{P}\}\, C\, \{\exists X.\mathcal{Q}\}}$$

Standard proof rules

HT-SKIP
$$\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, \mathbf{skip}\, \{\mathcal{P}\}$$

HT-SEQ
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C_1\, \{\mathcal{P}'\} \qquad \Gamma; \mathcal{R} \vdash \{\mathcal{P}'\}\, C_2\, \{\mathcal{Q}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C_1; C_2\, \{\mathcal{Q}\}}$$

HT-ITE
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P} * B\}\, C_1\, \{\mathcal{Q}\} \qquad \Gamma; \mathcal{R} \vdash \{\mathcal{P} * \neg B\}\, C_2\, \{\mathcal{Q}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\, \{\mathcal{Q}\}}$$

HT-WHILE
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P} * B\}\, C\, \{\mathcal{P}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, \mathbf{while}\ B\ \mathbf{do}\ C\, \{\mathcal{P} * \neg B\}}$$

HT-ASSIGN
$$\frac{X \notin \mathsf{fv}(\mathcal{R})}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}[X/E]\}\, X := E\, \{\mathcal{P}\}}$$

HT-PAR
$$\frac{\begin{array}{ll}\Gamma; \mathcal{R} \vdash \{\mathcal{P}_1\}\, C_1\, \{\mathcal{Q}_1\} & \mathsf{fv}(\mathcal{R}, \mathcal{P}_1, C_1) \cap \mathsf{mod}(C_2) = \emptyset \\ \Gamma; \mathcal{R} \vdash \{\mathcal{P}_2\}\, C_2\, \{\mathcal{Q}_2\} & \mathsf{fv}(\mathcal{R}, \mathcal{P}_2, C_2) \cap \mathsf{mod}(C_1) = \emptyset\end{array}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}_1 * \mathcal{P}_2\}\, C_1 \parallel C_2\, \{\mathcal{Q}_1 * \mathcal{Q}_2\}}$$

HT-ATOMIC
$$\frac{\Gamma; \mathsf{true} \vdash \{\mathcal{P} * \mathcal{R}\}\, C\, \{\mathcal{Q} * \mathcal{R}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, \mathbf{atomic}\ C\, \{\mathcal{Q}\}}$$

Figure 5.4: Standard proof rules of the program logic.

$$\textsc{ht-read}\quad \frac{X \notin \mathsf{fv}(\mathcal{R}, E, E')}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}[X/E'] * E \overset{\pi}{\hookrightarrow}_t E'\}\, X := [E]\, \{\mathcal{P} * E \overset{\pi}{\hookrightarrow}_t E'\}}$$

$$\textsc{ht-write}\quad \frac{t \neq \mathsf{proc}}{\Gamma; \mathcal{R} \vdash \{E_1 \overset{1}{\hookrightarrow}_t -\}\, [E_1] := E_2\, \{E_1 \overset{1}{\hookrightarrow}_t E_2\}}$$

$$\textsc{ht-alloc}\quad \frac{X \notin \mathsf{fv}(\mathcal{R}, E)}{\Gamma; \mathcal{R} \vdash \{\mathsf{true}\}\, X := \mathbf{alloc}\, E\, \{X \overset{1}{\hookrightarrow}_{\mathsf{std}} E\}}$$

$$\textsc{ht-dispose}\quad \frac{}{\Gamma; \mathcal{R} \vdash \{E \overset{1}{\hookrightarrow}_{\mathsf{std}} -\}\, \mathbf{dispose}\, E\, \{\mathsf{true}\}}$$

Figure 5.5: The non-standard proof rules related to heap handling.

invariant, since our logic is intuitionistic. Moreover, our assertion language does not contain the logical conjunction $\wedge$ connective, but uses $*$ instead.

Handling heaps. Figure 5.5 gives the proof rules that we extend with respect to classical CSL, related to heap ownership.

The rule HT-READ states that reading from the heap is allowed with *any* type t of heap ownership $\overset{\pi}{\hookrightarrow}_t$, whereas heap writing (HT-WRITE) is only allowed with points-to predicates of type std or act. The HT-WRITE rule thus restricts $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ assertions to exclusively grant read-access to the association location. We will in a moment see that the proof rule for **action** programs can upgrade $E \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E'$ predicates to $E \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E'$ predicates, to regain write access to the heap location at E. This system of upgrading enforces that all modifications to E are captured by the program abstraction the heap location is subject to, inside an action block.

The rule HT-ALLOC for heap allocation generates a new points-to predicate of type std, indicating that the allocated heap location is not (yet) subject to any program abstraction. Heap deallocation (HT-DISPOSE) requires a full *standard* ownership predicate for the associated heap location, thereby making sure that the deallocation does not break any bindings of active program abstractions.

are allowed to be thrown away, using the $*$-WEAK entailment rule. As a consequence, assertions cannot express "precise" properties about the content of the heap, including emptiness of heaps.

$$\textsc{ht-proc-init}\quad \frac{\mathsf{fv}(b_1) \subseteq \mathsf{dom}(\Pi) = \{x_0, \ldots, x_n\} \qquad I = \{0, \ldots, n\} \qquad X \notin \mathsf{fv}(\mathcal{R}, E_0, \ldots, E_n) \qquad B = b_1[x_i/E_i]_{\forall i \in I}}{\Gamma, \{b_1\}\, p\, \{b_2\}; \mathcal{R} \vdash \left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \overset{1}{\hookrightarrow} E_i * B \end{array} \right\} \; X := \textbf{process}\; p\; \textbf{over}\; \Pi \; \left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \overset{1}{\hookrightarrow}_{\mathsf{proc}} E_i * B * \\ \mathsf{Proc}_1(X, p, \mathsf{body}(p), \Pi) \end{array} \right\}}$$

$$\textsc{ht-proc-update}\quad \frac{\mathsf{fv}(a) = \{x_0, \ldots, x_n\} \subseteq \mathsf{dom}(\Pi) \qquad I = \{0, \ldots, n\} \qquad B_1 = \mathsf{pre}(a)[x_i/E_i]_{\forall i \in I} \qquad B_2 = \mathsf{post}(a)[x_i/E_i]_{\forall i \in I} \qquad \Gamma; \mathcal{R} \vdash \{*_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{procact}} E_i * B_1 * \mathcal{P}\}\, C\, \{*_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{procact}} E_i' * B_2 * \mathcal{Q}\}}{\Gamma; \mathcal{R} \vdash \left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i * B_1 * \\ \mathsf{Proc}_\pi(X, p, a \cdot P + Q, \Pi) * \mathcal{P} \end{array} \right\} \; \textbf{action}\; X.a\; \textbf{do}\; C \; \left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i' * B_2 * \\ \mathsf{Proc}_\pi(X, p, P, \Pi) * \mathcal{Q} \end{array} \right\}}$$

$$\textsc{ht-proc-finish}\quad \frac{\mathsf{fv}(b) \subseteq \mathsf{dom}(\Pi) = \{x_0, \ldots, x_n\} \qquad I = \{0, \ldots, n\} \qquad B = b_2[x_i/E_i]_{\forall i \in I} \qquad P \downarrow}{\Gamma, \{b_1\}\, p\, \{b_2\}; \mathcal{R} \vdash \left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \overset{1}{\hookrightarrow}_{\mathsf{proc}} E_i * \\ \mathsf{Proc}_1(X, p, P, \Pi) \end{array} \right\} \; \textbf{finish}\; X \; \left\{ *_{i \in I} \Pi(x_i) \overset{1}{\hookrightarrow} E_i * B \right\}}$$

Figure 5.6: The extended proof rules related to handling process-algebraic models.

Handling process-algebraic abstractions. Figure 5.6 gives the extended proof rules for introducing, eliminating, and updating process-algebraic abstractions.

The HT-PROC-INIT rule handles initialisation of an abstract model P over a set of heap locations as specified by the Π mapping. This rule requires *standard* points-to predicates with write-permission for any heap location that is to be bound by P, and these are converted to $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$. Moreover, HT-PROC-INIT requires that the precondition B of P holds, which is constructed from b_1 by replacing all process variables by the values at the corresponding heap locations as specified by Π[3].

[3]Here we slightly abuse notation however, for ease of presentation. In the proof rule, we

A Proc_1 predicate with full permission is ensured, containing the label p of the declared process, so that the postcondition b_2 can later be retrieved from Γ.

The HT-PROC-UPDATE rule handles updates to program abstractions, by performing an action a in the context of an **action** $X.a$ **do** C program. This rule imposes four preconditions on handling **action** programs. First, a predicate of the form $\mathsf{Proc}_\pi(X, p, a \cdot P + Q, \Pi)$ is required for some π. In particular, the process component must be of the form $a \cdot P + Q$ and therewith allow to perform the a action. After performing a, the process will be reduced to P, and Q will be discarded, as the choice is made not to follow execution as prescribed by Q. To get processes in the required format, one may apply Proc-$\cong$ (page 160), together with the standard axioms of process algebras given earlier. For example, processes of the form $a \cdot P$ can always be rewritten to $a \cdot P + \delta$ to obtain the required choice. Second, $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates are required for any heap location that is bound by Π. These points-to predicates are needed to resolve the precondition and postcondition of a. Third, the guard of a must hold as a precondition. And last, the remaining resource $\mathcal{P}$ should hold.

Among the premises of HT-PROC-UPDATE is a proof derivation for the sub-program C, in which all required $\overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}}$ predicates are essentially upgraded to $\overset{\pi_i}{\hookrightarrow}_{\mathsf{act}}$ and thereby regain write access when $\pi_i = 1$. However, in case $\pi_i < 1$ the upgrade does not give any additional privileges, since $\overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}}$ provides read-access just the same. We found that these unnecessary conversions complicate the soundness proof. To avoid unnecessary upgrades, we convert all affected $\overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}}$ predicates to $\overset{\pi_i}{\hookrightarrow}_{\mathsf{procact}}$ instead, which simplifies the correctness proof.

The HT-PROC-UPDATE rule ensures a process ownership predicate that holds the resulting process P after execution of a. In addition, updates to the heap are ensured that comply with the postconditions of the proof derivation of C.

Finally, the HT-PROC-FINISH rule handles finalisation of abstractions that have fully been executed (i.e., can successfully terminate). A predicate $\mathsf{Proc}_1(X, p, P, \Pi)$ with *full* permission is required (thereby implying that no other thread can have a fragment of the abstraction), where P must be able to successfully terminate. Successful termination in this sense means that P is bisimilar to $\varepsilon + Q$ for some process Q, and thus has the choice to have no further behaviour. The Proc_1 predicate is exchanged for the postcondition B of the abstraction, again constructed by replacing all process variables in b_2 by concrete values obtained via points-to assertions. The postcondition B can be established at this point, since (i) the contracts of processes in Γ are assumed, as their validity is checked externally, and b_2 is a postcondition of one of these contracts; (ii) the abstraction has been initialised in

write $b[x_i/E_i]_{\forall i \in I}$ for converting a process condition b to a condition over program variables, by substituting all free variables x_i occurring in b by an program expression E_i. However, in our Coq formalisation, we have a special operation for such conversions.

a state satisfying the precondition of that contract; and (iii) the process has been reduced to a point of successful termination. Hence all the classical assumptions of Hoare-triple reasoning are fulfilled.

Lastly, all $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$ predicates are converted back to $\overset{1}{\hookrightarrow}_{\mathsf{std}}$ to indicate that the associated heap locations are no longer subject to the abstraction.

5.6 Soundness

In this section we define the semantic meaning of program judgments, and discuss the soundness proof of the program logic.

The soundness proof has been mechanised using Coq. Proving soundness was non-trivial and required substantial auxiliary definitions. This section discusses these auxiliary definitions and explains how they are used. For further proof details we refer to the Coq development.

The soundness theorem relates program judgments with the operational semantics of programs and boils down to the following: if

1. a proof $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ can be derived for a program C; and
2. the contracts in Γ of all abstract models of C are satisfied (proven externally),

then C *executes safely for any number of computation steps*. Execution safety in this sense also includes that C does not fault for any number of execution steps, with respect to the fault semantics $\lightning$ that we defined in Section 5.3.3.

Our definition of execution safety extends the well-known inductive definition of configuration safety of Vafeiadis [Vaf11] by adding machinery to handle process-algebraic abstractions. The most important extension is a *simulation argument* between concrete program executions (with respect to $\rightsquigarrow$) and abstract program executions of all active models (with respect to $\xrightarrow{a}$). However, as the reduction steps of these two semantics do not directly correspond one-to-one, this simulation is established via an intermediate, instrumented semantics referred to as the *ghost operational semantics*. This intermediate semantics is defined in Section 5.6.1, in terms of *ghost transitions* $\rightsquigarrow_{\mathsf{ghost}}$ that essentially define the lock-step execution of program transitions $\rightsquigarrow$ and the transitions $\xrightarrow{a}$ of their abstractions. Our definition of "*executing safely for n execution steps*" includes that all $\rightsquigarrow$ steps can be simulated by $\rightsquigarrow_{\mathsf{ghost}}$ steps, and vice versa, for n execution steps. Thus, the end-result is a *refinement* between programs and their abstractions.

In addition to establishing such refinements, our definition of execution safety must also allow the postconditions of abstractions to be used inside the program logic, particularly in the context of the HT-PROC-FINISH proof rule. To account for

these, the definition of execution safety uses two extra ingredients, both of which are defined in Section 5.6.2. The first ingredient is the notion of *process execution safety*, from which the semantic meaning of process Hoare triples and of process environments Γ are defined. Informally, execution safety of a process Hoare triple $\{b_1\}\, p\, \{b_2\}$ states that all finite traces of $\mathsf{body}(p)$ starting from a state satisfying b_1, terminate in a state that satisfies b_2. The second ingredient is an invariant, stating that all *active* program abstractions *preserve their execution safety* for n execution steps, with respect to the current state of the program. Maintaining this invariant allows the postconditions of fully reduced process-algebraic models to be obtained and used, when executing a **finish _** ghost command.

Finally, Section 5.6.3 formally defines process execution safety—the semantic meaning of program judgments—and presents the soundness statement.

5.6.1 Ghost Operational Semantics

To establish the refinements between programs and their abstractions, an intermediate semantics is used that administers the states of all active program abstractions. This intermediate semantics is referred to in the sequel as the *ghost operational semantics*. The ghost semantics is expressed as a transition relation $\rightsquigarrow_{\mathsf{ghost}} \subseteq \mathit{GhostConf}$ between *ghost configurations* $\mathfrak{G} = (C, h, pm, s, g) \in \mathit{GhostConf}$, which extend program configurations by two extra components, namely:

- A process map $pm \in \mathit{ProcMap}$ that is used to administer the state of all *active* (initialised, but not yet finalised) process-algebraic abstractions; and
- An extra store $g \in \mathit{Store}$, referred to as a *ghost store*, as it is used to map variable names to process identifiers in the context of "ghost" instructions.

The ghost operational semantics uses two stores instead of one, to keep the administration of program data and specification-only (ghost) data strictly separated. By doing so, it is easier to establish that the variables referred to in ghost code do not interfere with regular program execution, and vice versa.

Ghost transitions essentially describe the *lock-step execution* of concrete programs ($\rightsquigarrow$ steps) and their program abstractions ($\xrightarrow{a}$ steps). An excerpt of the transition rules of the ghost semantics is presented in Figure 5.7. This excerpt only contains the transition rules related to program abstraction; all other transition rules are essentially the same as those of $\rightsquigarrow$, with the two extra configuration components simply carried over and left unchanged. Recall that the blue colourings are merely visual cues, do not have any special semantical meaning.

To clarify the ghost transition rules, ghost-PROC-INIT instantiates a new program abstraction and stores it in a free entry in pm. Finalisation of program abstractions is handled by ghost-PROC-FINISH, under the condition that the process-algebraic

ghost-PROC-INIT
$$\frac{pm(v) = \mathsf{free}}{(X := \textbf{process}\ p\ \textbf{over}\ \Pi, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (\textbf{skip}, h, pm[v \mapsto \langle p, \mathsf{body}(p), [\![\Pi]\!]s\rangle^1], s, g)}$$

ghost-PROC-FINISH
$$\frac{pm(g(X)) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^\pi \qquad P \downarrow}{(\textbf{finish}\ X, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (s, h, pm \setminus g(X), s, g)}$$

ghost-ACT-INIT
$$(\textbf{action}\ X.a\ \textbf{do}\ C, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (\textbf{inact}\ (a, g(X), h)\ C, h, pm, s, g)$$

ghost-ACT-STEP
$$\frac{(C, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (C', h', pm', s', g')}{(\textbf{inact}\ m\ C, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (\textbf{inact}\ m\ C', h', pm', s', g')}$$

ghost-ACT-END
$$\frac{pm(v) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^\pi \qquad P, \|\Lambda\|(h_{old}) \xrightarrow{a} P', \|\Lambda\|(h)}{(\textbf{inact}(a, v, h_{old})\ \textbf{skip}, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (\textbf{skip}, h, pm[v \mapsto \langle p, P', \Lambda\rangle^\pi], s, g)}$$

Figure 5.7: An excerpt of the transition rules of the ghost operational semantics.

model is able to terminate successfully. The remaining three ghost transition rules handle the execution of action blocks. Before discussing these, first observe that the ghost semantics maintains an extra component m in **inact** m C commands. This component contains *(ghost) metadata*: extra runtime information regarding the process-algebraic model in whose context the program C is being executed. Concretely, ghost metadata m is defined as a triple $m = (a, v, h) \in \mathit{Act} \times \mathit{Val} \times \mathit{Heap}$, consisting of:

1. The label a of the action that is being executed;
2. The identifier v of the corresponding process-algebraic model in the process map, in which the action a is being executed; and
3. A copy h of the heap, made when the program started to execute the action block; that is, when the **action** program was reduced to **inact** by $\rightsquigarrow$.

The ghost-ACT-INIT transition rule starts to execute an **action** block by reducing it to an **inact** program, thereby assembling and attaching the required ghost metadata. In particular, a copy of the heap is made at this point, so that the

ghost-ACT-END transition rule for finalising **inact** programs is able to access the old, original contents of the heap. This is needed to allow the abstraction to make a matching $\xrightarrow{a}$ step; in particular to calculate the pre-state of such a step. To see how this works, first recall that the process-algebraic state of program abstractions are linked to concrete program state—entries in the heap—via the Λ binders that are maintained in process maps. Therefore, to be able to make an $\xrightarrow{a}$ step, the ghost-ACT-END rule first needs to construct process-algebraic state out of the state of the program. This is done using the auxiliary function $\|\cdot\| : \mathit{Binder} \to \mathit{Heap} \to \mathit{ProcStore}$ that is referred to as the *abstract state reification function*, which has the following definition.

Definition 5.6.1 (Abstract state reification)**.**

$$\|\Lambda\|(h) \triangleq \lambda x \in \mathit{ProcVar}\,.\, \begin{cases} h(\Lambda(x)) & \text{if } x \in \mathsf{dom}(\Lambda) \text{ and } \Lambda(x) \in \mathsf{dom}(h) \\ \mathbb{1}_{\mathit{Val}} & \text{otherwise} \end{cases}$$

Finally, the ghost-ACT-STEP transition rule allows making a computation step in the context of an action (**inact**) program

5.6.1.1 Faulting Ghost Configurations

Likewise to the fault semantics that we defined for program configurations, we now also define a fault semantics of ghost configurations, which we call the *ghost fault semantics*. The ghost fault semantics is expressed in terms of a set $\lightning_{\mathsf{ghost}} \subset \mathit{GhostConf}$ of ghost configurations that are defined to "fault". Figure 5.8 gives an excerpt of the faulting ghost configurations. Only the faulting configurations related to ghost code are shown; the other cases are similar to the ones presented in Section 5.3.3. The notation $\lightning_{\mathsf{ghost}}(\mathfrak{G})$ is used to abbreviate $\mathfrak{G} \in \lightning_{\mathsf{ghost}}$.

Clarifying the ghost fault semantics; the initialisation of a process-algebraic model faults if there is no free entry available in pm ($\lightning_{\mathsf{ghost}}$-PROC-FULL). The finalisation of program abstractions can fault if the corresponding entry in the process map is: (i) either unoccupied or invalid ($\lightning_{\mathsf{ghost}}$-PROC-FINISH-1), or (ii) contains a process-algebraic abstraction that is unable to successfully terminate (by the rule $\lightning_{\mathsf{ghost}}$-PROC-FINISH-2). Finally, computation within action blocks **inact** m C may fault if: (i) m does not refer to an abstraction ($\lightning_{\mathsf{ghost}}$-ACT-SKIP-1), or (ii) the abstraction relies on process variables that have an incorrect binding (by the rule $\lightning_{\mathsf{ghost}}$-ACT-SKIP-2), or (iii) the process is not able to make a matching step ($\lightning_{\mathsf{ghost}}$-ACT-SKIP-3), or (iv) C is able to fault (by $\lightning_{\mathsf{ghost}}$-ACT-STEP).

The ghost semantics enjoys the same progress property as the standard operational semantics, as stated by Theorem 5.3.4.

$$\frac{\forall v\,.\,pm(v) \neq \mathsf{free}}{\lightning_{\mathsf{ghost}}(X := \textbf{process } p \textbf{ over } \Pi, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-PROC-FULL}$$

$$\frac{pm(g(X)) \in \{\mathsf{free}, \mathsf{inv}\}}{\lightning_{\mathsf{ghost}}(\textbf{finish } X, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-PROC-FINISH-1}$$

$$\frac{pm(g(X)) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} \qquad P \not\downarrow}{\lightning_{\mathsf{ghost}}(\textbf{finish } X, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-PROC-FINISH-2}$$

$$\frac{\lightning_{\mathsf{ghost}}(C, h, pm, s, g)}{\lightning_{\mathsf{ghost}}(\textbf{inact } m\ C, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-ACT-STEP}$$

$$\frac{pm(v) \in \{\mathsf{free}, \mathsf{inv}\}}{\lightning_{\mathsf{ghost}}(\textbf{inact } (a, v, h_{old})\ \textbf{skip}, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-ACT-SKIP-1}$$

$$\frac{pm(v) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} \qquad \mathsf{image}(\Lambda) \not\subseteq \mathsf{dom}(h) \cap \mathsf{dom}(h')}{\lightning_{\mathsf{ghost}}(\textbf{inact } (a, v, h_{old})\ \textbf{skip}, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-ACT-SKIP-2}$$

$$\frac{pm(v) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} \qquad \neg\exists P'\,.\,P, \|\Lambda\|(h_{old}) \xrightarrow{a} P', \|\Lambda\|(h)}{\lightning_{\mathsf{ghost}}(\textbf{inact } (a, v, h_{old})\ \textbf{skip}, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-ACT-SKIP-3}$$

Figure 5.8: An excerpt of the fault semantics of ghost configurations.

Theorem 5.6.1 (Progress of $\rightsquigarrow_{\mathsf{ghost}}$)**.** *For any ghost configuration* $\mathfrak{G} \notin \lightning_{\mathsf{ghost}}$*, either* $\mathfrak{G}$ *is final, or there exists a* $\mathfrak{G}'$ *such that* $\mathfrak{G} \rightsquigarrow_{\mathsf{ghost}} \mathfrak{G}'$*.*

Moreover, it is quite straightforward to establish a *forward simulation* between $\rightsquigarrow$ and $\rightsquigarrow_{\mathsf{ghost}}$. A matching backward simulation is ensured by the soundness argument of the program logic, as is customary for establishing refinements [REB98].

Lemma 5.6.2 (Forward simulation)**.** *The standard operational semantics and the fault semantics of programs are* embedded *in the ghost operational semantics and ghost fault semantics, respectively:*

1. *If* $(C, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (C', h', pm', s', g')$*, then* $(C, h, s) \rightsquigarrow (C', h', s')$*.*
2. *If* $\lightning(C, h, s)$*, then also* $\lightning_{\mathsf{ghost}}(C, h, pm, s, g)$*, for any* pm *and* g*.*

The above theorem also shows that the ghost fault semantics extends $\lightning$. The soundness argument of the program logic establishes that verified programs do not fault with respect to $\rightsquigarrow_{\mathsf{ghost}}$, and therefore also do not fault with respect to $\rightsquigarrow$.

5.6.2 Process Execution Safety

In addition to establishing refinements between programs and their abstract models, our notion of program execution safety (defined later, in Section 5.6.3) also needs logical mechanisms that allow the postconditions of finalised models to be used inside the program logic. These mechanisms are essential for establishing soundness of the HT-PROC-FINISH Hoare proof rule. We now discusses these mechanisms, consisting of the following two components:

1. A notion of *process execution safety*, from which a semantical notion of correctness of process Hoare triples and of process environments can be defined.
2. Machinery for expressing and maintaining an *invariant*, stating that all active program abstractions *preserve their execution safety* (that they established from the previous point, when they were initialised) throughout program execution, with respect to (reification of) the current program state.

To better discuss the use of such an invariant, let us first define process execution safety, and the semantic meaning of process Hoare triples.

5.6.2.1 Semantics of Process Hoare Triples

Process execution safety is defined as follows.

Definition 5.6.2 (Process execution safety)**.** Execution safety *of a process P with respect to a process store and a (post)condition, is defined in terms of a predicate $\checkmark(\cdot,\cdot,\cdot) : Proc \to ProcStore \to ProcCond \to Prop$. This predicate is coinductively defined such that, if $\checkmark(P,\sigma,b)$ holds, then:*

1. *If $P\downarrow$, then $[\![b]\!]\sigma$, and*
2. *For any a, P' and σ', if $P,\sigma \xrightarrow{a} P',\sigma'$, then $\checkmark(P',\sigma',b)$.*

Thus, any process configuration (P,σ) *executes safely* with respect to a postcondition b if, for any successfully terminating process configuration (P',σ') with $P'\downarrow$ that can be reached from (P,σ), it holds that satisfies $[\![b]\!]\sigma'$. It follows that the $\checkmark$ predicate is closed under bisimilarity.

Lemma 5.6.3. *If $\checkmark(P,\sigma,b)$ and $P \cong Q$, then $\checkmark(Q,\sigma,b)$.*

This notion of process execution safety is used to define *partial* correctness of process Hoare triples and of process environments, in the following way.

Definition 5.6.3 (Semantics of process Hoare triples)**.** *The semantics of process Hoare triples $\{b_1\}\,p\,\{b_2\}$ is expressed as a modelling relation $\models_{\mathsf{proc}} \{b_1\}\,p\,\{b_2\}$, that*

is defined as follows:

$$\models_{\mathsf{proc}} \{b_1\}\, p\, \{b_2\} \triangleq \models_{\mathsf{proc}} \{b_1\}\, \mathsf{body}(p)\, \{b_2\},\ \textit{where}$$
$$\models_{\mathsf{proc}} \{b_1\}\, P\, \{b_2\} \triangleq \forall \sigma \in \mathit{ProcStore}\,.\, [\![b_1]\!]\sigma \implies \checkmark(P, \sigma, b_2)$$

Here we overload the notation $\models_{\mathsf{proc}} \{b_1\} \cdot \{b_2\}$ for both processes and process labels. Observe that the above definition indeed states partial correctness. For example, we have that $\models_{\mathsf{proc}} \{b_1\}\, \delta\, \{b_2\}$ for any b_1 and b_2.

Definition 5.6.4 (Semantics of process environments)**.** *The semantics of process environments* Γ *is expressed as a satisfaction relation* $\models_{\mathsf{env}} \Gamma$*, which is defined by structural induction on* Γ*, in the following way:*

$$\models_{\mathsf{env}} \emptyset \qquad\qquad \frac{\models_{\mathsf{env}} \Gamma \qquad \models_{\mathsf{proc}} \{b_1\}\, p\, \{b_2\}}{\models_{\mathsf{env}} \Gamma, \{b_1\}\, p\, \{b_2\}}$$

A process Hoare triple $\{b_1\}\, p\, \{b_2\}$ is defined to be *semantically valid* if $\models_{\mathsf{proc}} \{b_1\}\, p\, \{b_2\}$. Likewise, a process environment Γ is defined to be *semantically valid* if all process Hoare triples in Γ are semantically valid, i.e., $\models_{\mathsf{env}} \Gamma$.

5.6.2.2 Preservation of Process Execution Safety

The invariant mentioned in the preamble will express that all active program abstractions retain their execution safety throughout program execution, with respect to $\checkmark$. Since active program abstractions are administered in process maps, we lift the notion of program execution safety to *process map safety*, expressed as judgments of the form $\Gamma; h \models_{\mathsf{pm}} pm$. Intuitively, a process map pm is *safe* if all process-algebraic abstractions stored in pm execute safely with respect to $\checkmark$, together with their postconditions in Γ. The heap h represents the current program state, and is reified into process-algebraic state using $\|\cdot\|(h)$.

Definition 5.6.5 (Process map safety)**.**

$$\Gamma; h \models_{\mathsf{pm}} pm \triangleq \forall v \in \mathit{Val}\,.\, \Gamma; h \models_{\mathsf{mc}} pm(v)$$

where $\Gamma; h \models_{\mathsf{mc}} mc$ *is defined by case distinction on* mc*, so that*

$$\begin{aligned}
\Gamma; h \models_{\mathsf{mc}} \mathsf{free} &\triangleq \mathsf{true} \\
\Gamma; h \models_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} &\triangleq \exists b_1, b_2\,.\, \{b_1\}\, p\, \{b_2\} \in \Gamma \wedge \checkmark(P, \|\Lambda\|(h), b_2) \\
\Gamma; h \models_{\mathsf{mc}} \mathsf{inv} &\triangleq \mathsf{false}
\end{aligned}$$

Free process cells are always safe, whereas corrupted entries inv are never safe. Moreover, the judgments $\models_{\mathsf{pm}}$ and $\models_{\mathsf{mc}}$ are both closed under bisimilarity.

Lemma 5.6.4.

1. *If* $\Gamma; h \models_{\mathsf{pm}} pm$ *and* $pm \cong_{\mathsf{pm}} pm'$, *then* $\Gamma; h \models_{\mathsf{pm}} pm'$.
2. *If* $\Gamma; h \models_{\mathsf{mc}} mc$ *and* $mc \cong_{\mathsf{mc}} mc'$, *then* $\Gamma; h \models_{\mathsf{mc}} mc'$.

In a moment (in Section 5.6.3) we will also define a notion of execution safety for commands. This notion of *program execution safety* maintains the aforementioned invariant that $\Gamma; h \models_{\mathsf{pm}} pm$ always holds throughout program execution, where h and pm are constructed from the current state, at every execution step. This invariant is needed to establish soundness of the HT-PROC-FINISH proof rule, as it requires the concerned process to be able to successfully terminate, and thus by Definition 5.6.2 must satisfy the postcondition of the abstraction.

However, one has to be careful on how to exactly state this invariant, to allow it to be re-established after every computation step. In most cases re-establishing the invariant is straightforward. For example, $\Gamma; h \models_{\mathsf{pm}} pm$ can be re-established after initialising a new program abstraction using the HT-PROC-INIT proof rule, by Definition 5.6.3 and by the structure of that proof rule. The invariant can also trivially be re-established after finalising an abstraction using HT-PROC-FINISH, as the abstraction is then no longer active and thereby removed from pm. However, computation steps that involve heap writing (i.e., handling of $[E] := E'$ programs) may be problematic, as illustrated by the following example.

Technicality 5.6.1 (Potential problems due to heap writing)**.** *To see the potential problem, consider the following code snippet.*

```
1 requires x > 0;
2 ensures x = 0;
3 action reset;
```

```
4 {Proc_π(X, p, reset · P, {x ↦ E}) ∗ ···}
5 action X.reset do {
6     [E] := −2; // the problem is here
7     [E] := 0;
8 }
9 {Proc_π(X, p, P, {x ↦ E}) ∗ ···}
```

Suppose that $\Gamma; h \models_{\mathsf{pm}} pm$ *holds on line 5. After computing line 6, the heap* h *holds the value* -2 *at location* $[\![E]\!]s$. *Moreover, the process map* pm *has not been changed, because the action program (lines 5–8) has not fully been executed yet. Nevertheless,* $\Gamma; h[[\![E]\!]s \mapsto -2] \models_{\mathsf{pm}} pm$ *may now be violated, as the* reset *action can no longer be performed, since* $x = -2$ *after reification, while* reset*'s precondition requires* x *to be positive.*

The root of the problem is that the invariant should not necessarily have to hold during intermediate steps while executing **action** programs, but only at the pre- and poststate of the action program. Program execution safety will solve this

by making a *snapshot of the heap* every time an action program is being started on (likewise to ghost-ACT-INIT), and expressing the invariant over these snapshot heaps. Snapshots are recorded at the level of permission heaps, which already have the required structure to do this: action heap cells $\langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}}$ allow to store *snapshot values* v_2 alongside "concrete" values v_1. These snapshot values are used to construct *snapshot heaps*, with help of the following operation.

Definition 5.6.6 (Snapshot heap)**.** *The* snapshot *of a permission heap is defined in terms of a total function* $\lfloor \cdot \rfloor_{\mathsf{snapshot}} : \mathit{PermHeap} \to \mathit{Heap}$*, so that*

$$\lfloor ph \rfloor_{\mathsf{snapshot}} \triangleq \lambda v \in \mathit{Val} \,.\, \lfloor ph(v) \rfloor_{\mathsf{snapshot}}$$

where $\lfloor hc \rfloor_{\mathsf{snapshot}}$ *is defined by case distinction on* hc*, so that*

$$\begin{aligned} \lfloor \langle v \rangle^{\pi}_{\mathsf{proc}} \rfloor_{\mathsf{snapshot}} &\triangleq v \\ \lfloor \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \rfloor_{\mathsf{snapshot}} &\triangleq v_2 \\ \lfloor hc \rfloor_{\mathsf{snapshot}} &\triangleq \textit{undefined, in all other cases} \end{aligned}$$

The snapshot $\lfloor ph \rfloor_{\mathsf{snapshot}}$ of a permission heap ph only contains heap cells bound by process-algebraic models, and is constructed by taking the snapshot values of all ph's action heap cells. As we shall see in Section 5.6.3, the final invariant maintained by program execution safety will be $\Gamma; \lfloor ph \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm$, where ph and pm are taken from the models of the program logic and represent the current state of the program. This invariant, combined with establishing a refinement between the program and its abstract models, provide sufficient means for proving soundness of the program logic.

5.6.3 Adequacy

This section defines *program execution safety* and uses it to define the semantic meaning of program judgments, from which the soundness theorem (i.e., adequacy of the logic) can be formulated. Program execution safety extends on the well-known notion of *configuration safety* of [Vaf11], by adding permission accounting, process-algebraic state, and the machinery introduced earlier, in §5.6.1 and §5.6.2.

First, in order to help connect the models of the program logic to concrete program state, we define a *concretisation function* for permission heaps.

Definition 5.6.7 (Concretisation)**.** Concretisation *of permission heaps is defined as a total function* $\lfloor \cdot \rfloor_{\mathsf{concr}} : \mathit{PermHeap} \to \mathit{Heap}$*, so that*

$$\lfloor ph \rfloor_{\mathsf{concr}} \triangleq \lambda v \in \mathit{Val} \,.\, \lfloor ph(v) \rfloor_{\mathsf{concr}}$$

where $\lfloor hc \rfloor_{\mathsf{concr}}$ *is defined by case distinction on* hc*, so that*

$$\begin{aligned}
\lfloor \langle v \rangle^{\pi}_{\mathsf{std}} \rfloor_{\mathsf{concr}} &\triangleq v \\
\lfloor \langle v \rangle^{\pi}_{\mathsf{proc}} \rfloor_{\mathsf{concr}} &\triangleq v \\
\lfloor \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \rfloor_{\mathsf{concr}} &\triangleq v_1 \\
\lfloor hc \rfloor_{\mathsf{concr}} &\triangleq \textit{undefined, in all other cases}
\end{aligned}$$

The heap concretisation operator constructs concrete program heaps out of permission heaps, by simply discarding all internal structure regarding process-algebraic models. Only the information relevant for regular program execution is retained. $\lfloor \cdot \rfloor_{\mathsf{snapshot}}$ essentially does the same, but only retains heap cells bound to program abstractions and prefers to take snapshot values whenever possible.

We now have all the required ingredients for defining adequacy. *Program execution safety* is defined in terms of a predicate $\mathsf{safe}^n_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$, stating that the program C is *safe* for n computation steps with respect to a permission heap ph, a process map pm, two stores s and g, a resource invariant $\mathcal{R}$, and a postcondition $\mathcal{Q}$.

Definition 5.6.8 (Program execution safety)**.** *The* $\mathsf{safe}^0_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *predicate always holds, whereas* $\mathsf{safe}^{n+1}_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *holds if and only if the following five conditions hold.*

1. *If* $C = \mathbf{skip}$*, then* $ph, pm, s, g \models \mathcal{Q}$*.*
2. *For every* ph_F *and* pm_F *such that* $ph \perp_{\mathsf{hc}} ph_F$ *and* $pm \perp_{\mathsf{mc}} pm_F$*, it holds that* $(C, \lfloor ph \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}}, pm \uplus_{\mathsf{pm}} pm_F, s, g) \notin \lightning_{\mathsf{ghost}}$*.*
3. *For any* $v \in \mathsf{acc}(C, s)$ *it holds that* $ph(v) \notin \{\mathsf{free}, \mathsf{inv}\}$*.*
4. *For any* $v \in \mathsf{writes}(C, s)$ *it holds that* $\mathsf{full}_{\mathsf{hc}}\, ph(v)$*.*
5. *For any* ph_J*,* ph_F*,* pm_J*,* pm_F*,* pm_C*,* h'*,* s'*, and* C' *such that, if:*

   5a. $ph \perp_{\mathsf{ph}} ph_J$ *and* $(ph \uplus_{\mathsf{ph}} ph_J) \perp_{\mathsf{ph}} ph_F$*, and*
   5b. $pm \perp_{\mathsf{pm}} pm_J$ *and* $(pm \uplus_{\mathsf{pm}} pm_J) \perp_{\mathsf{pm}} pm_F$*, and*
   5c. $\neg\mathsf{locked}(C)$ *implies* $ph_J, pm_J, s, g \models \mathcal{R}$*, and*
   5d. $(pm \uplus_{\mathsf{pm}} pm_J \uplus_{\mathsf{pm}} pm_F) \cong_{\mathsf{pm}} pm_C$*, and*
   5e. $\Gamma; \lfloor ph \uplus_{\mathsf{ph}} ph_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm_C$*, and*
   5f. $C, \lfloor ph \uplus_{\mathsf{ph}} ph_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}}, s \rightsquigarrow C', h', s'$*;*

   then there exists ph'*,* ph'_J*,* pm'*,* pm'_J*,* pm'_C*, and* g'*, such that*

   5g. $ph' \perp_{\mathsf{ph}} ph'_J$ *and* $(ph' \uplus_{\mathsf{ph}} ph'_J) \perp_{\mathsf{ph}} ph_F$*, and*
   5h. $pm' \perp_{\mathsf{pm}} pm'_J$ *and* $(pm' \uplus_{\mathsf{pm}} pm'_J) \perp_{\mathsf{pm}} pm_F$*, and*
   5i. $\lfloor ph' \uplus_{\mathsf{ph}} ph'_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}} = h'$*, and*

5j. $(pm' \uplus_{\mathsf{pm}} pm'_J \uplus_{\mathsf{pm}} pm_F) \cong_{\mathsf{pm}} pm'_C$, *and*
5k. $\Gamma; \lfloor ph' \uplus_{\mathsf{ph}} ph'_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm'_C$, *and*
5l. $\neg\mathsf{locked}(C')$ *implies* $ph'_J, pm'_J, s', g' \models \mathcal{R}$, *and*
5m. $(C, \lfloor ph \uplus_{\mathsf{ph}} ph_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}}, pm_C, s, g) \rightsquigarrow_{\mathsf{ghost}} (C', h', pm'_C, s', g')$, *and*
5n. $\mathsf{safe}^n_\Gamma(C', ph', pm', s', g', \mathcal{R}, \mathcal{Q})$.

To clarify, any configuration is safe for $n+1$ steps if: the postcondition is satisfied if the program C has terminated (***1***); the program C does not fault (***2***); C only accesses heap entries that are allocated (***3***); C only writes to heap locations for which full permission is available (***4***); and finally, after making a computation step the program remains safe for another n steps (***5***). Condition ***2*** implies race freedom, while conditions ***3*** and ***4*** account for memory safety.

Condition ***5*** is particularly involved. In particular, it encodes the backward simulation: if the program can do a $\rightsquigarrow$ step (***5f***), then it must be able to make a matching $\rightsquigarrow_{\mathsf{ghost}}$ ghost step (***5m***). Moreover, the resource invariant $\mathcal{R}$ must remain satisfied (due to ***5c*** and ***5l***) after making a computation step, whenever the program is not locked. In addition, the process maps invariably remain safe with respect to Γ and the snapshot heap due to ***5e*** and ***5k***, as discussed in the previous section. All the other (sub-)conditions are for the most part standard.

Let $\mathfrak{M} = (ph, pm, s, g) \in \mathit{World}$ be a world. We sometimes use the shorthand notation $\mathsf{safe}^n_\Gamma(C, \mathfrak{M}, \mathcal{R}, \mathcal{P})$ to abbreviate $\mathsf{safe}^n_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$.

Lemma 5.6.5. *Program execution safety satisfies the following properties:*

1. *If* $\mathsf{safe}^n_\Gamma(C, \mathfrak{M}, \mathcal{R}, \mathcal{Q})$ *and* $m \leq n$, *then* $\mathsf{safe}^m_\Gamma(C, \mathfrak{M}, \mathcal{R}, \mathcal{Q})$.
2. *If* $\mathsf{safe}^n_\Gamma(C, \mathfrak{M}_1, \mathcal{R}, \mathcal{Q})$ *and* $\mathfrak{M}_1 \cong_W \mathfrak{M}_2$, *then* $\mathsf{safe}^n_\Gamma(C, \mathfrak{M}_2, \mathcal{R}, \mathcal{Q})$.
3. *If* $\mathsf{safe}^n_\Gamma(C, \mathfrak{M}, \mathcal{R}, \mathcal{Q})$ *and* $\mathcal{Q} \models \mathcal{Q}'$, *then* $\mathsf{safe}^n_\Gamma(C, \mathfrak{M}, \mathcal{R}, \mathcal{Q}')$.

Property 1 in the above lemma states *monotonicity* of program execution safety: if a program C is safe for n computation steps, then C is also safe for less than n computation steps. Property 2 shows that program execution safety is *closed under bisimilarity*: process maps can always be replaced by bisimilar ones. Property 3 states that postconditions may always be weakened.

Semantics of program judgments. The semantics of program judgments is defined in terms of a quintuple $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$, expressing that C is safe for any number of execution steps, starting from any state (world) satisfying $\mathcal{P}$.

Definition 5.6.9 (Semantics of program judgments)**.** $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ *holds if and only if the following two conditions hold:*

(1) $C : \mathsf{user}$*, and*
(2) If $\models_{\mathsf{env}} \Gamma$ *and* $C : \mathsf{wf}$*, then:*

$$\forall \mathfrak{M} \,.\, \mathsf{valid}_W \mathfrak{M} \implies \mathfrak{M} \models \mathcal{P} \implies \forall n \,.\, \mathit{safe}^n_\Gamma(C, \mathfrak{M}, \mathcal{R}, \mathcal{Q}).$$

The underlying idea of the above definition, i.e., having a continuation-passing style definition for program judgments, has first been applied in [AB07] and has further been generalised in [Hob08] and [HAN08]. Moreover, the idea of defining (program) execution safety in terms of an inductive predicate originates from [AMRV07]. These two concepts have been reconciled in [Vaf11] into a formalisation for the classical CSL of Brookes [Bro07], that has been encoded and mechanically been proven in both Isabelle and Coq. Our definition builds on the latter, by having a refinement between programs and abstractions encoded in safe.

Observe that only judgments of user programs (i.e., commands free of runtime constructs like **inatom** and **inact**) have a semantic meaning. Also observe that the semantics of program judgments is conditional on the safety of Γ: it states that, if Γ is safe, only then the program C executes safely for any number n of computation steps, with respect to any valid world $\mathfrak{M}$ that satisfies C's precondition $\mathcal{P}$.

From the above definition, it trivially follows that $\Gamma; \mathcal{R} \models \{\mathsf{false}\}\, C\, \{\mathcal{P}\}$, for any Γ, $\mathcal{R}$, $\mathcal{P}$ and user program C. Notice however, that it does not follow that always $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathsf{true}\}$, since C might be able to fault in the general case.

Soundness. The following main soundness theorem states that verified programs (i.e., program for which a proof can be derived according to the proof rules given earlier) are semantically valid (i.e., are fault-free, memory-safe, and refine their process-algebraic models).

Theorem 5.6.6 (Soundness)**.**

$$\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \implies \Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$$

The soundness proofs of all proof rules have been mechanised using the Coq proof assistant and can be found on the Git repository accompanying this thesis.

The HT-PROC-UPDATE proof rule was the most difficult to prove sound, as it includes, among other things, (1) showing that the abstract model can always match the program with a simulating execution step, as well as (2) handling the invariant that was discussed in §5.6.2. On top of that, the combination of (1) and (2) requires some extra bookkeeping to ensure that the snapshot heaps stored in ghost metadata (discussed in §5.6.1) agree with the snapshot values stored in permission heaps. This additional bookkeeping has been left out of the formalisation presented so far, but the details of this can be studied in the Coq formalisation.

5.7 Implementation

The presented verification approach has been implemented in the VerCors verifier, which specialises in automated verification of parallel and concurrent programs written in high-level languages like (subsets of) Java and C [BDHO17]. VerCors can reason about programs using both heterogeneous concurrency (as in Java) and homogeneous concurrency (e.g., OpenCL), as well as compiler directives (as in OpenMP). VerCors allows (concurrent) programs to be specified with permission-based separation logic annotations. VerCors supports reasoning about data-race freedom, memory safety and functional program behaviour.

5.7.1 Tool Support

Tool support for our technique has been implemented in VerCors for languages with fork/join concurrency and statically-scoped parallel constructs [OBG+17]. Our technique has been implemented by defining an axiomatic domain for process types in Viper, consisting of constructors for the process-algebraic connectives and standard process-algebraic axioms to support these. The three different ownership types $\overset{\pi}{\hookrightarrow}_t$ are encoded in Viper by defining extra fields that maintain the ownership status t for each global reference. The Proc_π assertions are encoded as predicates over process types.

VerCors is able to reason about process-algebraic abstractions, by first linearising process terms and then encoding the linear processes and their contracts into Viper input. A process term is *linear* if it does not use the $\parallel$ and $\between$ connectives. The linearisation algorithm is based on a rewrite system that uses a subset of the standard process-algebraic axioms as rewrite rules [Use02] to eliminate parallel connectives. For example, a process term $(a_1 \cdot a_2) \parallel a_3$ can be linearised to the bisimilar process $a_1 \cdot a_2 \cdot a_3 + a_1 \cdot a_3 \cdot a_2 + a_3 \cdot a_1 \cdot a_2$. Alternatively, process-algebraic abstractions may also algorithmically be analysed; we are currently investigating the use of the mCRL2 [GM14] toolset and the Ivy verifier [PMP+16].

Moreover, the VerCors implementation of the abstraction technique is much richer than the simple language of Section 5.3 that is used to formalise the approach on. Notably, the abstraction language in VerCors supports general recursion instead of Kleene iteration, and allows process and action declarations to be parameterised by data. VerCors also has support for several axiomatic data types that enrich the expressivity of reasoning with program abstractions, like (multi)sets, bags, sequences, and option types.

5.7.2 Coq Formalisation

The formalisation and soundness proof (Sections 5.2–5.6) of the program logic have been fully mechanised using Coq, as a deep embedding that is inspired by [Vaf11]. The overall implementation comprises roughly 15.000 lines of code. The Coq development and its documentation can be found online [Sup].

5.8 Related Work

Significant progress has been made on the theory of concurrent software verification over the last years [FFS07, DYDG+10, SB14, SBP13, TDB13, NLWSD14, Fen09, RPDYG14]. This line of research proposes advanced program logics that all provide some notion of expressing and restricting thread interference of various complexity, via *protocols* [JSS+15]: formal descriptions of how shared-memory is allowed to evolve over time. In our approach protocols have the form of process-algebraic abstractions.

The original work on CSL [O'H07] allows specifying simple thread interference in shared-memory programs via resource invariants and critical regions. Later, RGSep [VP07] merges CSL with rely-guarantee (RG) reasoning to enable describing more fine-grained inter-thread interference by identifying atomic concurrent actions. Many modern program logics build on these principles and propose even more advanced and elaborate ways of verifying shared-memory concurrency. For example, TaDa [RPDYG14] and CaReSL [TDB13] express thread interference protocols through state-transition systems. iCAP [SB14] and Iris [KJB+17] propose a more unified approach by accepting user-defined monoids to express protocols on shared state, together with invariants restricting these protocols. The Iris logic therefore goes by the slogan *"Monoids and invariants are all you need"*[JSS+15]. In our technique the invariants take the form of process- and action contracts. Iris provides reasoning support for proving language properties in Coq, where our focus is on proving programs correct.

In the distributed setting, Disel [SWT17] allows specifying protocols for distributed systems. Disel builds dependent type theory and is implemented as a shallow embedding in Coq. Even though this approach is more expressive than ours, it can only semi-automatically be applied in the context of Coq. Villard et al. [VLC09] present a program logic for message passing concurrency, where threads may communicate over channels using native send/receive primitives. This program logic allows protocols to be specified via *contracts*, which are state-machines in the style of Session Types [HVK98], to describe channel behaviour. Our technique is more general, as the approach of Villard et al. is tailored specifically to basic shared-memory message passing. Actor Services [SM16] is a program logic with

assertions to express the consequences of asynchronous message transfers between actors. However, the meta-theory of Actor Services has not been proven sound.

Most of the related work given so far is essentially theoretical and mainly focuses on expressiveness rather than usability. Our approach is a balanced trade-off between expressivity and usability. It allows specifying process-algebraic protocols over a general class of concurrent systems, while also allowing the approach to be implemented in automated verification toolsets for concurrency, like VerCors. Related verification toolsets for concurrency are SmallfootRG [CPV07], VeriFast [JSP+11], CIVL [SZL+15] and Viper [JKM+14, MSS16]; the latter tool is used as the main back-end of VerCors. SmallfootRG is a memory-safety verifier based on RGSep. VeriFast is a rich toolset for formal verification of (multi-threaded) Java and C programs using separation logic. Notably, Penninckx et al. [PJP15] extend VeriFast with Petri-net extensions to reason about the I/O behaviour of programs. This Petri-net approach is similar to ours, however our technique supports reasoning about abstract models and allows reasoning about more than just I/O behaviour. The CIVL framework can reason about race-freedom and functional correctness of MPI programs written in C [ZRL+15, LZS17]. The reasoning is done via bounded model checking and symbolic execution.

Apart from the proposed technique, VerCors also allows process-algebraic abstractions to be used as *histories* [BHZS15, ZS15]. This is more or less dual to our approach: instead of *reducing* abstract models, the history approach *records* all actions a encountered in **action** $X.a$ **do** C programs during computation, and thereby builds-up a history process. This process can be analysed to reason about the history of changes in the corresponding concrete shared-memory locations in the program. Our work subsumes this approach, as history recording is only suitable for terminating programs. Our approach performs the reasoning up-front and allows specifying behavioural patterns of the functional behaviour of non-terminating programs. A locking protocol is a classical example of this, consisting of two states, "locked" and "unlocked", and expressing that clients of the protocol may only transition to "locked" while being in an "unlocked" state, and vice versa. Also related in this respect are the time-stamped histories of Sergey et al. [SNB15b] and the more general work on proving linearisability [HW90, Vaf10a, KSW17]. These approaches allow to reason about fine-grained concurrency by using sequential verification techniques. Our technique, as well as the history-based technique use process-algebraic linearisation to do so.

Other type theoretical approaches to reason about concurrency and distribution are Session Types [HVK98, HYC08, HMM+12]. These approaches typically use process calculi (e.g., the π-calculus) to describe the type of the communication protocol. Behavioural safety of programs is then checked through type checking. Our technique integrates with separation logic and supports reasoning about

communication behaviour, as shown by the case study.

5.9 Conclusion

To reason effectively about realistic concurrent and distributed software, we have presented a verification technique in the current and previous chapter that:

1. Performs the reasoning at a *suitable level of abstraction* that hides irrelevant implementation details;
2. Is *scalable* to realistic programs by being modular and compositional; and
3. Is *practical* by being supported by automated tools.

The approach is a trade-off between expressivity and usability: it is expressive enough to allow reasoning about realistic software, as is demonstrated by the case studies in Chapter 4 (Section 4.6), and at the same time usable enough to be implemented as part of a deductive program verifier, viz. VerCors. In contrast, many related program logics mainly aim for expressiveness, while their usage requires substantial manual labour (either because the proof is hand-written, which is also error-prone, or because all steps need to be done interactively within a theorem prover) and therefore hardly scale to realistic programs.

For **1.** we use process algebra with data, which offers a mathematically elegant way of expressing program behaviour at a suitable abstraction level. Such process-algebraic specifications can be seen as *models*, over which we can check safety properties, for example via model checking against temporal logic formulas.

For **2.** we use a concurrent separation logic that handles process-algebraic models as *resources* that can be split and consumed. This allows verifying in a thread-modular way that programs behave as specified by their abstract models, and allows process-algebraic reasoning to be projected onto program behaviour.

For **3.** the approach has been implemented in VerCors [OBG+17] and has been illustrated on various case studies. The proof system underlying our technique has mechanically been proven sound using Coq. Our technique is therefore supported by a strong combination of theoretical justification (this chapter) and practical usability (Chapter 4) for reasoning about realistic software.

In the upcoming chapter, we apply our abstraction approach on an industrial case study: the formal verification of a safety-critical traffic tunnel control system. More specifically, we model the state-machine specification of this tunnel control system in mCRL2, and use VerCors with the presented technique to verify that the implementation correctly follows its specification.

5.9.1 Future Directions

We consider the presented technique as just the beginning of a comprehensive verification framework that aims to capture many different concurrent and distributed programming paradigms, covering for example shared-memory concurrency (this chapter) and message passing concurrency (Chapter 7).

We are currently investigating the use of mCRL2 and Ivy to reason algorithmically about process-algebraic abstractions. It would also be interesting to investigate to what extent this reasoning can be done *compositionally*, e.g., by using techniques like [CK99, CGP03]. Moreover, the approach currently only preserves safety properties when connecting programs to their abstract models. We are planning to investigate the preservation of liveness properties as well.

CHAPTER 6

Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System

Abstract

Over the last decades, significant progress has been made on formal techniques for software verification. However, despite this progress, these techniques are not yet structurally applied in industry. To reduce the well-known industry–academia gap, industrial case studies are much-needed, to demonstrate that formal methods are now mature enough to help increase the reliability of industrial software. Moreover, case studies also help researchers to get better insight into industrial needs.

This chapter contributes such a case study, concerning the formal verification of an industrial, safety-critical traffic tunnel control system that is currently employed in Dutch traffic. We made a formal, process-algebraic model of the informal design of the tunnel system, and analysed it using mCRL2. Additionally, we deductively verified that the implementation adheres to its intended behaviour, by proving that the code refines our mCRL2 model, using VerCors. By doing so, we detected undesired behaviour: an internal deadlock due to an intricate, unlucky combination of timing and events. Even though the developers were already aware of this, and deliberately provided us with an older version of their code, we demonstrate that formal methods can indeed help to detect undesired behaviours within reasonable time, that would otherwise be hard to find.

6.1 Introduction

Despite tremendous progress over the last decades on both the theory and practice of formal techniques for software verification [HJ18], these techniques are not yet structurally applied in industrial practice, not even in the case of safety-critical software. Even though formal methods have shown to be able to increase software

reliability [Cla08, Fil11, GRB+15], their application is often time consuming and may additionally require expert knowledge. Nevertheless, especially in the case of safety-critical software where reliability demands are high, industry can benefit *greatly* from the current state-of-the-art in formal verification research.

To make this apparent, industrial case studies are needed that show industry and society that formal methods are now ready to help increase software dependability in practice. In turn, such industrial case studies also help researchers and developers of verification tools to get insight into the needs of industry. By doing so, researchers can improve and adapt their techniques to industrial needs, and thereby reduce the well-known gap between academia and industry.

This chapter discusses such an industrial case study. It elaborates on our experiences and results of the formal verification of a safety-critical component of a *control system for a traffic tunnel* that is currently in use in the Netherlands. This particular software component is responsible for handling emergencies. When a fire breaks out inside the tunnel, or a traffic accident occurs, it should start an emergency procedure that evacuates the tunnel, starts the fans to blow away any smoke, turns on the emergency lights to guide people out, and so on. Naturally, the Dutch government imposes very high reliability demands on the traffic tunnel control software, and in particular on this emergency component, which are specified in a document of requirements that is over 500 pages in length [NTS].

The tunnel control software is developed by Technolution [Tec], a Dutch software and hardware development company located in Gouda. Technolution has hands-on experience in developing safety-critical, industrial software[1]. The development process of the traffic tunnel control system came together with a very elaborate process of quality assurance/control, to satisfy the high demands on reliability. Significant time and energy has been spent on software design and specification, code inspection, peer reviewing, unit and integration testing, etc.

In particular, during the design phase, the intended behaviour of the tunnel control software has been worked out in great detail: all system behaviours have been specified in pseudo code beforehand. Moreover, these pseudo code descriptions together have been structured further into a finite state machine, whose transitions describe how the different software behaviours change the internal state of the system. Nevertheless, both the pseudo code and this finite state machine have been specified *informally*, and do not have a precise, checkable formal semantics. Throughout the software development process, no formal methods or techniques have been used to assist in the major effort of quality control.

In this case study, we investigate how formal methods can help Technolution to find potential problems in their specification and (Java) implementation, with realistic

[1] To illustrate, Technolution also delivers commercial software written in Rust.

effort, and preferably at an early stage of development. Technolution is above all interested in establishing whether (**1**) the specification is *itself* consistent, by not being able to reach problematic states, e.g., deadlocks in the finite state machine; and (**2**) whether the Java code implementation is written correctly with respect to the pseudo code specification of the intended behaviour.

To address both these properties, we use a combination of existing verification techniques, to deal with their different nature. More specifically, for (**1**) we construct a *formal model* of the pseudo code specification and the underlying finite state machine. This model is specified as a process algebra with data, using the mCRL2 modelling language. After that, we use the mCRL2 *model checker* to verify whether the model adheres to certain requirements (e.g., deadlock freedom and strong connectivity), which we formalise in the modal μ-calculus.

For (**2**), we use VerCors [BDHO17] to *deductively* verify whether the control system is correctly implemented with respect to the pseudo code specification, using the techniques from Chapter 4 and Chapter 5 [OBH16, OBG+17]. This is done by proving that the implementation is a *refinement* of our mCRL2 model.

Our verification effort actually led to the detection of undesired behaviour: the system can potentially reach an internal state in which the calamity procedure is not invoked when an emergency has occurred, due to an intricate, unlucky combination of timing and events. Even though Technolution was well-aware of this—they deliberately provided us with an older version of their specification and implementation—we demonstrate that formal methods *can* indeed help to find such undesired behaviours at an early stage of development[2]. We also demonstrate that formal techniques are able to provide results within reasonable time, that are otherwise hard to find manually. To illustrate, this undesired behaviour was found within approximately 7 working days.

6.1.1 Contributions

This chapter contributes a successful industrial verification case study that concerns real-world, safety-critical code, and discusses our verification effort and results[3]. The contributions of the case study itself are:

- A formal process-algebraic model of the informal pseudo code description of the tunnel control software, that is defined using mCRL2.

[2] Prior to our formalisation work, we were unaware of any details of this undesired behaviour, but we were aware that there was a problem somewhere in the specification that Technolution had already found.

[3] This chapter is based on the article [OH19b].

- An analysis of this mCRL2 model, via state-space exploration, and by checking desired μ-calculus properties on the model, like deadlock-freedom.
- A machine-checked proof that the (Java) implementation adheres to the pseudo code specification, by proving that the program refines our mCRL2 model. This refinement proof is done using the automated verifier VerCors.

Here we should note that the actual Java implementation of the tunnel control system is confidential, as well as the documents from the design phase, and therewith also the mCRL2 model and VerCors files that we produced. We therefore sometimes slightly simplify their presentation for the purpose of this chapter, for example by using different variable/method/transition names. Nevertheless, the presentation of the case study does not deviate very much from the original, so this chapter still gives an accurate overview of our approach and results.

6.1.2 Chapter Outline

The remainder of this chapter is organised as follows. Section 6.2 gives preliminaries on the use of mCRL2. Preliminaries on the use of VerCors can be found in Chapter 2. Then, Section 6.3 gives more detail on how the tunnel control system is informally specified by Technolution, by discussing the structure of the pseudo code and the finite state machine. Section 6.4 explains how we modelled this informal specification in mCRL2, after which Section 6.5 discusses its analysis. Section 6.6 explains how VerCors is used to deductively prove that the tunnel control system correctly implements our mCRL2 model (the preliminaries for this can be found in Chapters 4 and 5). Section 6.7 relates our work to existing approaches and industrial case studies, before Section 6.8 concludes.

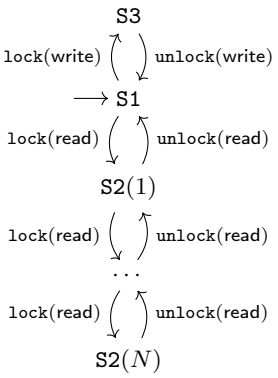
6.2 Preliminaries on mCRL2

During the case study, we modelled the (informal) tunnel control software specification as a process algebra with support for data. This was done using the specification language of mCRL2 [GM14]. mCRL2 is a toolset that comes with an ACP-style process-algebraic modelling language, and contains more than sixty tools to support visualisation, simulation, minimisation, state-space generation and model checking of these mCRL2 processes [BGK$^+$19]. The back-end for model checking takes as input an mCRL2 model, together a temporal property specified in the modal μ-calculus, and determines whether the model satisfies this property. By default this is done via exhaustive (symbolic) state space analysis.

We further illustrate how this modelling and analysis works by means of a small example, that is presented in Figure 6.1. This example demonstrates how a simple read–write (RW) lock can be specified and verified with mCRL2. A RW lock can

```
1  sort AccessType ≜ read | write;
2
3  act lock, unlock : AccessType;
4
5  proc
6     S1 ≜ lock(read) · S2(1) +
      lock(write) · S3;
7     S2(n : ℕ>0) ≜
8        n < N → lock(read) · S2(n + 1) +
9        1 < n → unlock(read) · S2(n − 1) +
10       n = 1 → unlock(read) · S1;
11    S3 ≜ unlock(write) · S1;
12
13 init S1;
```

(a) The mCRL2 model of a RW lock.



(b) The underlying state machine.

Figure 6.1: An mCRL2 specification of a RW lock, and the underlying state space.

be acquired multiple times for read-only purposes, for example to allow multiple clients to read from a shared segment of memory, but can also provide exclusive write access for a single client: a multiple-reader/single writer lock.

Figure 6.1b shows the corresponding state machine. Initially the RW lock is unlocked (S1). From here the lock can be acquired once for the purpose of writing (S3), via a lock(write) action, and can subsequently be released again via unlock(write). Similarly, from S1, the lock can be acquired/released multiple times for reading purposes. The state S2(n) represents a read lock that has been acquired n times, where n is bounded to some constant threshold N.

Figure 6.1a presents the mCRL2 encoding of this locking protocol. The specification language of mCRL2 has various built-in data types (like positive numbers; see line 7), but also allows defining custom abstract data types, as **sort**'s. Line 1 defines a sort that enumerates the different kinds of accesses that can be granted by the RW lock: read-only (read) access, and read/write (write) access.

Line 3 defines the *actions* for the locking protocol, which represent the basic, observable behaviours of the system. In this example, there are only two observable events, namely locking and unlocking. In mCRL2, actions can be parameterised by data. In this case, both actions are parameterised by *AccessType*.

These two actions can be composed into *processes* (lines 5–11). This example defines three processes, corresponding to the three locking states: S1 (unlocked),

`S2` (locked for reading purposes), and `S3` (locked for read/write purposes).

Processes are of the following form, where e is an expression:

$$P, Q \ ::= \ \varepsilon \mid \delta \mid a(\overline{e}) \mid \tau \mid P \cdot Q \mid P + Q \mid b \to P \mid X(\overline{e}) \qquad \text{(processes)}$$

Of course, the mCRL2 modelling language is actually much richer than the above language [GM14], e.g., by supporting parallelism and communication. Yet this is the fraction of the mCRL2 language that we will use throughout the chapter.

Clarifying the constructs: ε is the *empty* process, without behaviour, whereas δ is the *deadlocked* process, which neither progresses nor terminates. The process $a(\overline{e})$ is an *action invocation*, with $\overline{e}$ a sequence of arguments, while τ is a special, reserved action that models internal, *un*observable system events. $P \cdot Q$ is the sequential composition of P and Q, whereas $P + Q$ is their non-deterministic choice, The process $b \to P$ is the *conditional* process, that behaves as P only if b is a Boolean expression that evaluates to *true*, and otherwise it behaves as δ. Finally, $X(\overline{e})$ is the invocation of a process named X, with input arguments $\overline{e}$.

Moving back to the example, the `S1` process can either perform a `lock(read)` action, followed by the process invocation `S2`(1), or can do a `lock(write)`, after which `S3` is invoked (see line 6). Here the $\cdot$ connective has the highest precedence, followed by $\to$, and then $+$. Process `S3` (line 11) is only able to release the write lock and therewith to continue as `S1`. Finally, `S2`(n) allows (re)acquiring/releasing read locks, given that n is small/large enough, respectively, on lines 8–10.

For the actual case study, we modelled the tunnel control system in a similar way: by studying the state machine specification, and encoding it into mCRL2.

Modal μ-calculus. After having constructed a model, mCRL2 allows analysing it, by checking whether it satisfies a given temporal specification. These specifications are written in the *modal μ-calculus*, a powerful formalism that allows specifying properties about sequences of actions, i.e., traces of the input model.

Properties in the modal μ-calculus are defined by the following language.

$$\alpha, \beta \ ::= \ \mathit{true} \mid a(\overline{e}) \mid \neg\alpha \mid \alpha \cdot \beta \mid \alpha + \beta \mid \alpha^* \qquad \text{(action formulae)}$$
$$\phi, \psi \ ::= \ b \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle\phi \mid [\alpha]\phi \mid \mu x.\phi \mid \nu x.\phi \qquad \text{(state formulae)}$$

The actual specification language of mCRL2 is again much richer; we refer to [GM14] for a complete overview and a more detailed description.

Properties in the modal μ-calculus are defined in terms of *action* and *state* formulae. Action formulae α describe sequences of actions $a(\overline{e})$, where *true* stands for

any action. Such descriptions are negatable: $\neg\alpha$ expresses any sequence except for α. Action formulae can also sequentially be composed, $\alpha_1 \cdot \alpha_2$, or alternatively be composed, $\alpha_1 + \alpha_2$, and α^* is the *repetition* (Kleene iteration) of α.

State formulae ϕ, ψ express properties that should hold in the current state. These properties may for example be built from pure Boolean expressions b, but may also contain *modalities*, $\langle\alpha\rangle\phi$ and $[\alpha]\phi$, to express that ϕ must hold after a certain sequence of actions α has been observed. More specifically, $\langle\alpha\rangle\phi$ is the *may modality*, which expresses that, from the current state, the model is able to perform a sequence of actions complying with α, after which ϕ directly holds. Its dual is the *must modality*, $[\alpha]\phi$, which expresses that, from the current state, after the performance of any action sequence α, the property ϕ directly holds.

State formulae may also contain *fixpoint operators*, μ and ν, to specify infinite system behaviour. Here $\mu x.\phi$ is the *least fixpoint* of ϕ, i.e., the smallest reachable set of states satisfying ϕ, where x is the fixpoint variable. These are used to express liveness properties. Its dual is the *greatest fixpoint*, $\nu x.\phi$, used to express safety properties, representing the largest reachable set of states satisfying ϕ.

Below three example properties are given that hold for our RW lock model:

$$[\texttt{unlock}(\mathsf{read})]\mathit{false} \tag{6.1}$$

$$\nu x.(\,\langle \mathit{true}^* \cdot \texttt{lock}(\mathsf{write})\rangle \mathit{true} \wedge [\mathit{true}^*]x\,) \tag{6.2}$$

$$\nu x.(\,[(\neg\texttt{unlock}(\mathsf{write}))^* \cdot \texttt{lock}(\mathsf{read})]\mathit{false} \wedge [\mathit{true}^* \cdot \texttt{lock}(\mathsf{write})]x\,) \tag{6.3}$$

Property (6.1) states that read locks cannot be released from the initial state, as initially no locks have been acquired. Furthermore, (6.2) expresses that it always remains possible to acquire the write lock. Finally, (6.3) states that, when holding the write lock, no read lock can be obtained until the write lock is released.

For our case study we specified and verified properties similar to these three.

6.3 Informal Tunnel Software Specification

Before detailing how mCRL2 and VerCors are applied on the actual case study, let us first discuss the informal specification of the traffic tunnel control system.

Technolution invested significantly in an extensive design phase, to ensure the quality of the control system and to cope with the high reliability demands. During this phase, the intended behaviour of the control software was written-out in pseudo code, together with domain experts. These pseudo code specifications were further structured into a finite state machine (FSM). The states of this FSM are the operational states of the tunnel system (e.g., operating normally, under

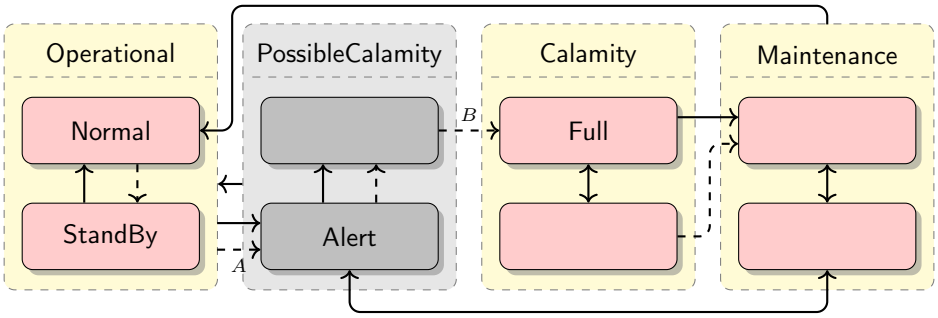


Figure 6.2: A simplified visual representation of the FSM. The two transitions that are later written-out as pseudo code in Figure 6.3 are labelled A and B.

repair, evacuating, etc.), while the transitions are the pseudo code descriptions of the system behaviour. The FSM thus illustrates how the different behaviours of the tunnel system should change its operational state.

Moreover, during the development phase, significant time and effort were invested in ensuring that the code was correctly implemented with respect to this specification. This was done primarily via unit testing and code reviewing.

This section gives more detail on how the tunnel control software was (informally) specified. §6.3.1 discusses the structure of the FSM, after which §6.3.2 elaborates on the pseudo code specification, i.e., the transitions of the FSM.

6.3.1 Structure of the FSM

Figure 6.2 illustrates how the FSM specification of the tunnel control system is structured. This illustration is simplified for confidentiality reasons: the actual FSM contains many more states and transitions. Nevertheless, the overall structure and the described behaviour are close to the original FSM specification.

The operational states are organised in a 2-layer hierarchy. For example, the composite state Operational contains two sub-states: Normal and StandBy. Transitions come in two flavours. Solid transitions (→) represent *manual interactions*, made by human operators through control panels. Dashed transitions (-→) are *automatic events* that are taken autonomously by the control system itself, for example to react to time-outs or sensor output. Any transition whose source is a composite state can also be taken by any of the underlying substates. Moreover, the composite PossibleCalamity state (displayed in grey) is a *ghost state*. Ghost states are special, in the sense that the system can be in a ghost state while also being

in a non-ghost state (e.g., to specify that a GUI dialog is being displayed). For example, the tunnel system can be in Alert and Normal simultaneously.

The functional meaning of the specification is roughly as follows. Being in Normal means that the system is in the normal operating state. From Normal the system may autonomously go in StandBy state, as result of, e.g., smoke or heat in the traffic tunnel that is detected via sensor reading. If the system finds enough reason to suspect a real calamity, it may autonomously decide to go from Operational to the Alert state. The Alert state can also manually be entered, when a human operator presses the emergency button on a control panel. The PossibleCalamity composite ghost state starts a timer upon entering. While in this state, if a human operator does not intervene in time by manually cancelling the alert status (thereby going back to the Operational state), the system will automatically launch the Calamity programme, for example to evacuate the traffic tunnel. Such calamities can be recovered from via Maintenance: manually repairing or resolving the calamity's cause. By doing so, the system can manually be brought back to the Normal operating state. However, it may also re-enter Alert in case new potential calamities are detected during maintenance.

6.3.2 Pseudo Code Specification

Figure 6.3 gives an idea of the structure of the pseudo code specification of the tunnel system. These pseudo code descriptions were provided by the Dutch Ministry of Infrastructure and Water Management, as part of a national standard on traffic tunnels [NTS]. The figure highlights two transitions of Figure 6.2, labelled as A and B, that describe interesting, important key system behaviours. Transition A specifies how the control system should autonomously request a calamity status when it suspects the traffic tunnel to be in an emergency situation. This will cause the system to go into the Alert ghost state, and therewith start the timer. Transition B specifies what should happen when this timer expires.

Elaborating on the textual format, all autonomous/manual system behaviours are specified in pseudo code style. Any such system behaviour corresponds to a transition in the FSM (denoted by `transition`) and is given a unique identifier (`name`). The internal state of the system is determined by the values of a set of *pseudo-variables*, which are prefixed with a `#` in the figure. The `effect` clauses exactly describe how the transition changes the internal state. The `condition` clauses specify under which conditions these state changes are allowed.

Transition A is able to request the calamity procedure to be initiated, by setting `#request_calamity` to `true`, given that `#possible_calamity_detected` has been set to `true` by some other system behaviour, e.g., as result of sensor reading. Such a request will also configure a timer, named `#calamity_timeout`, for cancelling

```
1  transition: A (autonomous)
2  name: ‘ProceedToAlertStatus‘
3  condition:
4    #possible_calamity_detected = true &&
5    #request_calamity = false &&
6    #state = Operational;
7  effect:
8    #request_calamity := true;
9    #calamity_timeout := now() +
         __calamity_timeout_frame;
10
11 transition: B (autonomous)
12 name: ‘StartCalamityProgrammeAfterTimeout‘
13 condition:
14   #state != Calamity &&
15   #request_calamity = true &&
16   now() > #calamity_timeout;
17 effect:
18   #request_calamity := false;
19   #state := Calamity::Full;
20   invoke CalamityProgramme();
```

Figure 6.3: The format of the textual specification of the tunnel system.

the request. Transition B specifies what should happen when this timer expires: in that case the system should enter the operational state Calamity (if not already in there) and start the `CalamityProgramme()`.

The control software of every Dutch traffic tunnel is required to comply with these specifications. This is checked by an external code review committee.

6.4 Modelling the Control System using mCRL2

Even though the tunnel control software has been specified extensively, prior to our work there had been no formal, structural effort to establish whether the specification *itself* obeys the desired properties. For Technolution, the main properties of interest concern *reliability* and *recoverability*: does the system always go into the Calamity state in real emergency situations? And is it always possible to recover from calamities, and thereby go back to the Normal operational state?

```
1  sort
2    State ≜ struct Normal | StandBy | Alert | Full | ···;
3    Var ≜ struct possibleCalamityDetected | requestCalamity | ···;
4    Val ≜ struct true | false | unknown | ···;
5
6  act enter : State;
7
8  proc
9    %% Encoding of transition A (autonomous)
10   ProceedToAlertStatus(state : State, σ : Var → Val, phase : Nat) ≜
11     σ(possibleCalamityDetected) ∧ ¬σ(requestCalamity) ∧
12       isInOperational(state) →
13         enter(Alert) · System(state, σ[requestCalamity := true], phase);
14
15   %% Encoding of transition B (autonomous)
16   StartCalamityProgrammeAfterTimeout(state, σ, phase) ≜
17     ¬isInCalamity(state) ∧ σ(requestCalamity) →
18       enter(Full) · System(Full, σ[requestCalamity := false], phase);
19
20   %% Encoding of the top-level specification
21   System(state : State, σ : Var → Val, phase : Nat) ≜
22     %% First phase: handling GUI input
23     (phase = 1) →
24       (CancelPossibleCalamity(state, σ, phase) +
25       omitted + τ · System(state, σ, 2)) +
26     %% Second phase: handling internal/external controls and function calls
27     (phase = 2) → ( omitted + τ · System(state, σ, 3)) +
28     %% Third phase: processing autonomous system behaviour
29     (phase = 3) →
30       (ProceedToAlertStatus(state, σ, phase) +
31       StartCalamityProgrammeAfterTimeout(state, σ, phase) +
32       omitted + τ · System(state, σ, 4)) +
33     %% Fourth phase: processing sensor data and update all variables
34     (phase = 4) → (τ · System(state, updateVars(σ), 1));
35
36 init System(Maintenance, σ_init, 1);
```

Figure 6.4: The main structure of the mCRL2 formalisation of the specification.

To automatically check for such desired properties, we modelled the pseudo code specifications and the underlying FSM as a process algebra, using mCRL2. Figure 6.4 shows the main structure of our mCRL2 model. This is again a simplified representation; the actual model consists of roughly 700 lines of code.

mCRL2 allows new data types to be defined using the **sort** keyword. We use data sorts to explicitly model the different operational states that the tunnel system might be in, as the structured sort *State*, defined on line 2. Also explicitly modelled are the various "pseudo-variables" that are used in the textual specification (defined on line 3), together with a domain of values for these variables (on line 4). These three data types are used to model the internal state of the tunnel control system.

Line 6 covers the definition of actions, which model the basic, observable units of computation. One of the main challenges was to determine which observable behaviours of the tunnel system to model explicitly. We experienced that modelling too many behaviours leads to search space explosions, while modelling too few hampers analysis. As the main properties of interest are properties of operational state reachability, the most important observable events to model are the transitions between the operational states. These are modelled as `enter(`s`)` actions, where $s \in$ *State* is the operational state that is being entered.

The traffic tunnel control system is modelled as the `System(`$state, \sigma, phase$`)` process (lines 21–34), whose arguments determine the internal state of the tunnel. In particular, *state* determines its operational state, whereas σ provides a valuation for all pseudo-variables. The third argument, *phase*, is maintained for technical reasons. This is because the overall system is specified and implemented as a (busy) working loop, that continuously cycles through four different phases, to (1) handle GUI input, (2) process internal requests, (3) autonomously make decisions, and (4) read from sensors and update all variables accordingly. These phases have been made explicit in our model, using *phase*. Every phase has the non-deterministic choice to advance to the next phase, as an internal τ action.

The earlier highlighted transitions A and B both describe autonomous behaviour, and thus are both handled in phase 3 (lines 30–31). Their behaviours are modelled on lines 9–18, and closely follow the pseudo code specification.

Finally, line 36 specifies the initial state of the control system. The system initialises in Maintenance state, and starts by handling phase 1 events. The mapping σ_{init} is a constant that holds the initial valuation of pseudo-variables.

6.5 Analysing the Control System with mCRL2

Now that we have a formal model of the tunnel system specification in mCRL2, we can study its state space, and determine whether it satisfies desired properties,

formulated in the modal μ-calculus, with relatively little effort. Technolution was primarily interested in verifying these properties: (i) Deadlock freedom and strong connectivity: are all operational states reachable at any point during execution? (ii) Reliability: does the system automatically go to Full after an emergency has been detected, unless this is manually cancelled? (iii) Recoverability: can calamities always be recovered from, by getting the system back to Normal?

A major challenge during analysis was to keep the model's state space small enough to be able to analyse it in a reasonable time. In particular, we needed to improve our mCRL2 model various times, as earlier versions suffered from state space explosions resulting from the explicit modelling of time. Recall that the informal specification includes software behaviours that depend on time, for example the timers that are maintained by the PossibleCalamity ghost state. In earlier versions of our model, these timers were modelled explicitly, as discrete values: natural numbers that were bounded by some threshold. However, their analysis was only feasible with thresholds no larger than three time units, which is insufficient. We later solved this scalability issue by modelling time *implicitly*, by constructing the model in such a way that certain actions must happen before others. More specifically, instead of having certain actions depend on timers or timeouts to happen before others, we let them happen non-deterministically, but in such a way that the original order of action occurrences is preserved.

Our latest model has an underlying state space of roughly 4.200 states and 25.400 transitions, which takes about 4 minutes to generate[4]. This clearly shows that the tunnel system specification comprises far too many behaviours for software designers/developers to comprehend, without the help of automated tooling. In fact, mCRL2 helped us further, by allowing to minimise this state space modulo branching bisimilarity [GW16], leaving only 27 states and 98 transitions. This reduction gave us better insight into the system's behaviour.

Together with Technolution we formulated several dozens of desired properties as μ-calculus formulae, and checked these on the reduced mCRL2 model. An example of such a formula is given below, expressing that the StandBy state can only ever be reached via the Normal state of operation.

$$\nu x.\big(\ [(\neg \texttt{enter(Normal)})^* \cdot \texttt{enter(StandBy)}]\mathit{false} \wedge \tag{6.4a}$$

$$[\mathit{true}^* \cdot \texttt{enter(StandBy)}]x\ \big) \tag{6.4b}$$

More precisely, this greatest fixed-point formula expresses that StandBy cannot be reached via any path of non-"enter(Normal)" actions (by 6.4a), and that this reachability property remains preserved each time StandBy is entered (6.4b).

[4]On a Macbook with an Intel Core i5 CPU with 2.9 GHz, and 8Gb internal memory.

In addition to checking these properties, we also inspected the state space of the minimised model and discussed its structure with Technolution. Ultimately, our verification exposed an intricate violation of the requirement of reliability. We found that the control system can reach a potentially dangerous situation, in which the Calamity state cannot automatically be entered after having detected a potential emergency (unless a human operator manually interferes), due to an intricate, unlucky combination of timing and events. The following reliability property exposes this behaviour, by stating that, while being in the Alert ghost state, it must always be possible to *directly* enter Full, unless the alert status is manually `cancel`led. This property does not hold for our mCRL2 model.

$$[\mathit{true}^* \cdot \texttt{enter(Alert)}]\,\nu x.\big(\tag{6.5a}$$

$$[\neg(\texttt{cancel} + \texttt{enter(Full)})^*]\langle \texttt{enter(Full)}\rangle \mathit{true} \wedge \tag{6.5b}$$

$$[\mathit{true}^* \cdot \texttt{enter(Alert)}]x\ \big) \tag{6.5c}$$

Note that this is precisely the violating behaviour that Technolution hoped we would find. This is because they already found it, by chance, and deliberately provided us with an older version of their specification and implementation. Our case study therefore shows that formal techniques can indeed help to find such problematic behaviours in a more reliable and structural manner, and at an early stage of development, within reasonable time: we found it within 7 working days.

6.6 Specification Refinement using VerCors

As a next step, we use VerCors to deductively verify that the code implementation adheres to the FSM and pseudo code specification. This is done by proving that the code correctly implements (refines) our mCRL2 model, using the approach of Chapters 4 and 5. Such a proof also adds value to the model, as it establishes that the model is a sound abstraction of the program's behaviour.

As explained in Chapters 4 and 5, the process algebra language that VerCors uses is an extension of mCRL2, in which all process and action definitions are enriched with pre/postcondition-style contracts. These contracts are used to connect/link processes and actions to program code: they logically describe how the performance of an action corresponds to an update to shared memory, very much like the `effect` and `condition` clauses used in the pseudo code specification. With these contracts we can mechanically prove with VerCors that every execution of the program corresponds to an action trace (a run) in the mCRL2 model. These links between programs and models preserve safety properties (i.e., $\nu x.\phi$).

For this project, we manually encoded our mCRL2 model into the process algebra

```
1  shared bool possibleCalamityDetected, requestCalamity;
2  shared State state;
3
4  modifies state;
5  effect state = s;
6  action enter(State s);
7
8  // The encoding of transition A, as a single action
9  accesses possibleCalamityDetected, state;
10 modifies requestCalamity;
11 guard possibleCalamityDetected ∧ ¬requestCalamity;
12 guard isInOperational(state);
13 effect requestCalamity;
14 action ProceedToAlertStatus;
15
16 accesses state;
17 modifies requestCalamity;
18 guard ¬isInCalamity(state) ∧ requestCalamity;
19 effect ¬requestCalamity;
20 action flipCalamityRequest;
21
22 // The encoding of transition B, as a sequential composition of two actions
23 process StartCalamityProgrammeAfterTimeout ≜
24   flipCalamityRequest · enter(Full);
```

Figure 6.5: The VerCors encoding of transitions A and B, as processes with contracts.

language of VerCors[5]. Figure 6.5 shows an excerpt of this encoding, in which transitions A and B are again highlighted. The VerCors encoding consists of a large number of action declarations, corresponding to the FSM transitions, with contracts that closely follow the pseudo code specifications. Moreover, this version does not use a valuation σ for the pseudo variables, like in §6.4, but rather connects to the actual shared fields in the program code (e.g., lines 1–2). The variables *state* and *phase* have been translated likewise. Our VerCors encoding is intended, but not (yet) proven, to be equivalent to the mCRL2 version.

Line 6 defines the `enter(`s`)` action, whose performance has the effect of modifying the shared variable *state*, by assigning s to it. Lines 9–14 give the specification

[5]Both these languages can be translated into one another, and we are actively working on mechanising these translations.

of transition A, as a single action, with a contract that closely follows the corresponding textual specification. Transition B is defined to be composed out of two actions: `flipCalamityRequest` for setting the *requestCalamity* flag to *false*, and `enter` for changing the operating state of the tunnel to Full.

Program annotations. The next step is to deductively prove that the implementation adheres to the VerCors encoding of the specification. The actual tunnel control system is implemented in Java. However, we converted this implementation to PVL—an object-oriented toy input language of VerCors—since our model-based verification approach is currently best supported by the PVL front-end (we are currently improving its support for Java).

Figure 6.6 shows and highlights the annotations of the PVL code implementations of transitions A (lines 2–16) and B (lines 19–35). The **yields bool** *branch* annotations on lines 2 and 19 indicate that *branch* is an extra (ghost) *output* parameter that only exists for the sake of specification. In the figure, *branch* represents which branch has been executed by the program, and is used in the postconditions to ensure the matching, corresponding process-algebraic choice.

The contract of proceedToAlertStatus states that it will execute as prescribed by the process `ProceedToAlertStatus` $\cdot P + Q$ for some P and Q (line 3), and depending on the execution branch taken, is left with either P (line 4) or with Q (line 5) upon termination. The contract of startCalamityProgrammeAfterTimeout follows the same specification pattern, as well as most of the other methods. Since this model-based verification approach is compositional, we could use it to verify that the entire implementation complies with the process-algebraic specification.

Our deductive verification effort did not directly reveal any problems or violations in the implementation: all methods comply with their specified behaviour. This is expected, as the implementation has been unit tested and code reviewed very rigorously. Nevertheless, this compliance between specification and implementation is now confirmed, by means of a machine-checked proof.

However, this verification did help us, as tool developers, to better understand the needs from industry, and to identify weaknesses in our approach and tooling. To give an example, for future use, Technolution finds it important that our model-based verification technique is applicable on Java code, instead of PVL, and in a more automated manner. We now actively work on improving this.

```
1  // The annotated code implementation of Transition A
2  yields bool branch;
3  requires Proc(ProceedToAlertStatus · P + Q);
4  ensures branch ⟹ Proc(P);
5  ensures ¬branch ⟹ Proc(Q);
6  void proceedToAlertStatus() {
7    branch := false;
8    if (possibleCalamityDetected ∧ ¬requestCalamity ∧
9          state = Normal ∨ state = StandBy) {
10     action ProceedToAlertStatus {
11       requestCalamity := true;
12     }
13   }
14   calamityTimeout := now() + __calamity_timeout_frame();
15   branch := true;
16 }
17
18 // The annotated code implementation of Transition B
19 yields bool branch;
20 requires Proc(StartCalamityProgrammeAfterTimeout · P + Q);
21 ensures branch ⟹ Proc(P);
22 ensures ¬branch ⟹ Proc(Q);
23 void startCalamityProgrammeAfterTimeout() {
24   branch := false;
25   if ("state is in calamity" ∧ requestCalamity) {
26     action flipCalamityRequest {
27       requestCalamity := false;
28     }
29     action enter(Full) {
30       state := Full;
31       calamityProgramme();
32     }
33     branch := true;
34   }
35 }
```

Figure 6.6: Relating the tunnel specification to the implementation using VerCors.

6.7 Related Work

Various successes have been reported in the use of model checking in industrial case studies. mCRL2, for example, maintains a gallery of industrial showcases on-

line [MSC], which includes, among others, the modelling and analysis of firmware for a pacemaker [Wig07], as well as control software used for experiments at the Large Hadron Collider at CERN [HKK+13]. Glabbeek et al. formalised the AODV wireless routing protocol in AWN (Algebra for Wireless Networks) [GHPT16]—a process algebra for modelling mobile ad-hoc networks—and used it to reason about safety-critical routing properties. Ruijters et al. [RGNS16] uses statistical model checking to study different maintenance strategies for railway joint, in collaboration with ProRail—a Dutch national railway infrastructure manager. Moreover, [Bee08] reports on the experiences of the use of TLA+ at Intel, for formal verification.

In the context of deductive verification, in 2015, de Gouw et al. [GRB+15] successfully detected an intricate bug in the standard implementation of OpenJDK's TimSort algorithm, which is used daily by billions of users worldwide. Another successful application of deductive verification is the use of Infer at Facebook [CDD+15], to detect potential regressions during continuous integration testing. In [BKLL15], a formal verification of a cloud hypervisor is reported, using Frama-C. Also OpenJML has been used successfully for the verification of industrial code; [Cok18] discusses several observations and experiences. Moreover, [PMP+14] discusses four industrial case studies that have been performed with VeriFast: two Java Card smart card applets, a Linux networking component, and a Linux device driver.

Regarding combinations of deductive verification and model checking, in [SOP12], CBMC and Frama-C have been used to verify embedded software for satellite launching. But apart from this work, we are not aware of any other industrial applications of model checking combined with deductive verification.

6.8 Conclusion

During our case study, we found that, even though the specification of the tunnel control system is informal, it *is* well-structured, and therefore has the potential to be formalised within reasonable time. In roughly 7 working days, we constructed a formal model of the informal specification, analysed it using mCRL2, and used VerCors to deductively prove that the code implementation adheres to it. This resulted in the detection of undesired behaviour, preventing the control system from automatically starting the calamity procedure after an emergency has been detected. Even though Technolution was already aware of this behaviour, they found it coincidentally. We demonstrate that formal methods can indeed help to find such undesired behaviours more structurally, and within realistic time.

This case study also helped us to learn about the needs from industry, and the shortcomings of our tooling, which we will work on before starting the follow-up project. More specifically, we will improve VerCors's support for Java, and work

on automated translations between mCRL2 and the process algebra language of VerCors. We will also investigate if the pseudo code specification language can be formalised into a DSL, that is automatically translatable to mCRL2.

In the next chapter, we adapt our model-based verification approach for the distributed case, and use process algebra to abstract message passing communication, instead of shared-memory behaviour and evolvement.

Part III

Advances in Distributed Program Verification

CHAPTER 7

Practical Abstractions for Automated Verification of Message Passing Concurrency

Abstract

Distributed systems are notoriously difficult to develop correctly, due to the concurrency in their communicating subsystems. Several techniques are available to help developers to improve the reliability of message passing software, including deductive verification and model checking. Both these techniques have advantages as well as limitations, which are complementary in nature.

This chapter contributes a novel verification technique that combines the strengths of deductive- and algorithmic verification to reason elegantly about message passing concurrent programs, thereby reducing their limitations. Our approach allows verifying data-centric properties of message passing programs using concurrent separation logic (CSL), and allows specifying their communication behaviour as a process-algebraic model. The key novelty of the approach is that it formally bridges the typical abstraction gap between programs and their models, by extending CSL with logical primitives for proving deductively that a program refines its process-algebraic model. These models can then be analysed via model checking, using mCRL2, to reason indirectly about the program's communication behaviour. Our verification approach is compositional, comes with a mechanised correctness proof in Coq, and is implemented as an encoding in Viper.

7.1 Introduction

Distributed software is notoriously difficult to develop correctly. This is because distributed systems typically consist of multiple communicating components, which together have too many concurrent behaviours for a programmer to comprehend. Software developers therefore need formal techniques and tools to help them understand the full system behaviour, with the goal to guarantee the reliability of

safety-critical distributed software. Two such formal techniques are *deductive verification* and *model checking*, both well-established in research [BK08, ABB+16, BO16, CHVB18] and proven successful in practice [Cla08, GRB+15]. Nevertheless, both these techniques have their limitations. Deductive verification is often labour-intensive as it requires the system behaviour to be specified manually, via non-trivial code annotations, which is especially difficult for concurrent and distributed systems. Model checking, on the other hand, suffers from the typical abstraction gap [PGS01] (i.e., discrepancies between the program and the corresponding model), as well as the well-known state-space explosion problem.

This chapter contributes a scalable and practical technique for automated verification of message passing concurrency that reduces these limitations, via a sound combination of deductive verification and model checking, based on the ideas of Chapters 4–6. Our verification technique builds on the insight that deductive and algorithmic verification are complementary [MN95, Uri00, ACPS15, ACPS17, Sha18]: the former specialises in verifying data-oriented properties (e.g., the function `sort`(xs) returns a sorted permutation of xs), while the latter targets temporal properties of control-flow (e.g., any `send` must be preceded by a `recv`). Since realistic distributed software deals with both computation (data) and communication (control-flow), such a combination of complementary verification techniques is needed to handle both data-centric and temporal program aspects.

More specifically, our verification approach uses concurrent separation logic (CSL) to reason about data properties of message passing concurrent programs, and allows specifying their communication behaviour as a process-algebraic model. The key innovation is that CSL is used not only to specify data-oriented properties, but also to formally link the programs' communication behaviour to the process-algebraic specification of its behaviour, thereby bridging the typical abstraction gap between programs and their models. These process-algebraic models can then be analysed algorithmically, e.g., using the mCRL2 toolset [GM14, BGK+19], to reason indirectly about the communication behaviour of the program. These formal links preserve *safety properties*; the preservation of liveness properties is left as future work. This approach has been proven sound using the Coq theorem prover, and has been implemented as a manual encoding in the Viper concurrency verifier [MSS16].

7.1.1 Running Example

To further motivate the approach, consider the example program in Figure 7.1, consisting of three threads that exchange integer sequences via *synchronous* (block-

```
1  send (⟨4, 7, 5⟩, 1);
2  xs := recv 2;
3  assert xs =
4      ⟨4, 5, 6, 7, 8⟩;
```

(a) Thread 1

```
5  while (true) {
6    (ys, t) := recv;
7    if (t = 1) then
8      send (ys + ⟨8, 6⟩, 3);
9    else send (ys, 2);
10 }
```

(b) Thread 2

```
11 while (true) {
12   (zs, t) := recv;
13   zs′ := ParSort(zs);
14   send (zs′, t);
15 }
```

(c) Thread 3

Figure 7.1: Our example message passing program, consisting of three threads that communicate via synchronous (blocking) message passing.

ing) message passing[1]. The goal is to verify whether the asserted property in Thread 1's program holds. This program is a simplified version of a typical scenario in message passing concurrency: it involves computation as well as communication and has a complicated communication pattern, which makes it difficult to see and prove that the asserted property indeed holds.

Clarifying the program, Thread 1 first sends the sequence $\langle 4, 7, 5\rangle$ to the environmental threads as a message with tag 1, and then receives any outstanding integer sequence tagged 2. Thread 2 continuously listens for incoming messages of any tag with a wildcard receive, and redirects these messages (as a network router), possibly with slightly modified content depending on the message tag. Thread 3 is a computing service: it sorts all incoming requests and sends back the result with the original tag. `ParSort` is assumed to be the implementation of an intricate, heavily optimised parallel sorting algorithm. Note that the asserted property on line 4 holds because the **send** on line 8 is always executed, no matter the interleaving of threads.

Two standard potential approaches for verifying this property are deductive verification and model checking. However, neither of these approaches provides a satisfying solution. Techniques for deductive verification, e.g., concurrent separation logic, have their power in modularity and compositionality: they require modular independent proofs for the three threads, and allow to compose these into a single proof for the entire program. This would not work in our example scenario, as the asserted property is inherently global. One could attempt to impose global invariants on the message exchanges [WL89], but these are generally hard to come by. Finding a global invariant for this example would already be difficult, since there is no obvious relation between the contents of messages and their tags. Other approaches use ideas from assume-guarantee reasoning [RHH$^+$01, VP07] to

[1]The approach is not limited to synchronous message passing; Section 7.5 discusses extensions to asynchronous (non-blocking) message passing.

somehow intertwine the independent proofs of the threads' programs, but these require extra non-trivial specifications of thread-interference and are difficult to integrate into (semi-)automatic verifiers.

Alternatively, one may construct a model of this program and apply a model checker, which fits more naturally with the temporal nature of the program's communication behaviour. However, this does not give a complete solution either. In particular, certain assumptions have to be made while constructing the model, for example that `ParSort` is correctly implemented. The correctness property of `ParSort` is data-oriented (it relates the output of `ParSort` to its input) and thus in turn fits more naturally with deductive verification. But even when one uses both these approaches—deductive verification for verifying `ParSort` and model checking for verifying communication behaviour—there still is no formal connection between their results: perhaps the model incorrectly reflects the program's communication behaviour.

7.1.2 Contributions and Outline

This chapter contributes a novel approach that allows making such formal connections[2], by extending CSL with primitives for proving that a program *refines* a process-algebraic model, with respect to send/receive behaviour. Section 7.2 introduces the syntax and semantics of programs and process-algebraic models. Notably, our process algebra language is similar to mCRL2, but has a special *assertion primitive* of the form $?b$, that allows encoding Boolean properties b into the process itself, as logical assertions. These properties can be verified via a straightforward reduction to mCRL2, and can subsequently be used (relied upon) inside the deductive proof of the program, via special program annotations of the form **query** b (allowing to "query" for properties b proven on the process algebra level). Section 7.3 illustrates in detail how this works on the example program of Figure 7.1, before Section 7.4 discusses its underlying logical machinery and its soundness proof. This soundness argument has been mechanised using Coq, and the program logic has been encoded in the Viper concurrency verifier. Section 7.5 discusses various extensions of the verification approach. Finally, Section 7.6 relates our work to existing approaches and Section 7.7 concludes.

7.2 Programs and Processes

This section introduces the syntax and semantics of the programming language (§7.2.1) and the process algebra language of models (§7.2.2) that is used to formalise the approach.

[2]This chapter is based on the article [OH19c].

7.2.1 Programs

The syntax of our simple concurrent pointer language, inspired by Brookes [Bro07] and O'Hearn [O'H07], is as follows, where $x, y, z, \cdots \in \mathit{Var}$ are *variables* and $v, w, \cdots \in \mathit{Val}$ are *values*.

Definition 7.2.1 (Expressions, Conditions, Commands)**.**

$$
\begin{aligned}
e \in \mathit{Expr} ::= {} & v \mid x \mid e + e \mid e - e \mid \cdots \\
b \in \mathit{Cond} ::= {} & \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \wedge b \mid e = e \mid e < e \mid \cdots \\
C \in \mathit{Cmd} ::= {} & \mathbf{skip} \mid C; C \mid x := e \mid x := [e] \mid [e] := e \mid \\
& \mathbf{send}\ (e, e) \mid (x, y) := \mathbf{recv} \mid x := \mathbf{recv}\ e \mid \\
& \mathbf{query}\ b \mid x := \mathbf{alloc}\ e \mid \mathbf{dispose}\ e \mid C \parallel C \mid \\
& \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ b\ \mathbf{do}\ C \mid \mathbf{atomic}\ C
\end{aligned}
$$

This language has instructions to handle dynamically allocated memory, i.e., heaps, as well as primitives for message passing, to allow reasoning about both shared-memory and message passing concurrency models, and their combination.

The notation $[e]$ is used for *heap dereferencing*, where e is an expression whose evaluation determines the heap location to dereference. Memory can be dynamically allocated on the heap using the **alloc** e instruction, where e will be the initial value of the fresh heap cell, and be deallocated using **dispose**.

The command **send** (e_1, e_2) sends a message e_1 to the environmental threads, where e_2 is a *message tag* that can be used for message identification. Messages are received in two ways: $x :=$ **recv** e receives a message with a tag that matches e, whereas $(x, y) :=$ **recv** receives any message and writes the message tag to the extra variable y. Message passing is assumed to be synchronous for now. Note that, although this language does not have a notion of communication channels, directed communication can be achieved by using message tags.

The specification command **query** b is used to connect process-algebraic reasoning to deductive reasoning: it allows the deductive proof of a program to rely on (or *assume*) a Boolean property b, which is proven to hold (or *guaranteed*) via process-algebraic analysis. This is a ghost command that does not interfere with regular program execution.

The function $\mathsf{fv} : \mathit{Expr} \to 2^{\mathit{Var}}$ is used to determine the set of free variables of expressions as usual, and is overloaded for conditions. Substitution is written $e_1[x/e_2]$ (and likewise for conditions) and has a standard definition: replacing each occurrence of x by e_2 in e_1. Their definitions are deferred to Appendix B.

7.2.1.1 Semantics of Programs

The denotational semantics of expressions $[\![e]\!]\sigma$ and conditions $[\![b]\!]\sigma$ is defined in the standard way, with $\sigma \in \mathit{Store} \triangleq \mathit{Var} \to \mathit{Val}$ a *store* that gives an interpretation to variables. Their definitions can be found in Appendix B as well. Sometimes $[\![e]\!]$ is written instead of $[\![e]\!]\sigma$ when e is closed, and likewise for $[\![b]\!]$.

The operational semantics of commands is defined as a labelled small-step reduction relation $\dot{\longrightarrow} \subseteq \mathit{Conf} \times \mathit{Label} \times \mathit{Conf}$ between configurations $\mathit{Conf} \triangleq \mathit{Cmd} \times \mathit{Heap} \times \mathit{Store}$ of programs. Heaps $h \in \mathit{Heap} \triangleq \mathit{Val} \rightharpoonup_{\mathsf{fin}} \mathit{Val}$ are used to model shared memory and are defined as finite partial mappings, such that heap locations are modelled simply as values from Val. The transition labels represent the atomic (inter)actions of threads, and are defined as follows.

Definition 7.2.2 (Program transition label)**.**

$$l \in \mathit{Label} \;::=\; \mathit{send}(v, v) \mid \mathit{recv}(v, v) \mid \mathit{comm}(v, v) \mid \mathit{cmp} \mid \mathit{qry}$$

Transitions labelled $\mathit{send}(v, v')$ indicate that the program sends a value v from the current configuration, together with a tag v'. These can be received by a thread, as a transition labelled $\mathit{recv}(v, v')$. By doing so, the sending and receiving threads *communicate*, represented by the *comm* label. Internal computations that are not related to message passing are labelled *cmp*, e.g., heap reading or writing. The only exception to this are the reductions of **query** commands, which are given the label *qry* instead, for later convenience in proving soundness.

Figure 7.2 gives the reduction rules for message exchanging. All other reduction rules are standard in spirit [Vaf11, Mil89] and are therefore deferred to Appendix B. For ease of presentation, a *synchronous* (blocking) message passing semantics is used for now. However, our approach can easily be extended to asynchronous (non-blocking) message passing, which is explained in Section 7.5.

7.2.2 Processes

In this work the communication behaviour of programs is specified as a process algebra with data, whose language is based on mCRL2 and defined by the following grammar.

Definition 7.2.3 (Processes)**.**

$$\begin{aligned} P, Q \in \mathit{Proc} \;::=\; & \varepsilon \mid \delta \mid \mathsf{send}(e, e) \mid \mathsf{recv}(e, e) \mid ?b \mid b : P \mid \\ & P \cdot P \mid P + P \mid P \parallel P \mid \Sigma_x\, P \mid P^* \end{aligned}$$

$$(\textbf{send}\ (e_1, e_2), h, \sigma) \xrightarrow{\mathit{send}([\![e_1]\!]\sigma, [\![e_2]\!]\sigma)} (\textbf{skip}, h, \sigma)$$

$$((x, y) := \textbf{recv}, h, \sigma) \xrightarrow{\mathit{recv}(v, v')} (\textbf{skip}, h, \sigma[x \mapsto v, y \mapsto v'])$$

$$(x := \textbf{recv}\ e, h, \sigma) \xrightarrow{\mathit{recv}(v, [\![e]\!]\sigma)} (\textbf{skip}, h, \sigma[x \mapsto v])$$

$$(\textbf{query}\ b, h, s) \xrightarrow{\mathit{qry}} (\textbf{skip}, h, s)$$

$$\frac{(C_1, h, \sigma) \xrightarrow{\mathit{send}(v_1, v_2)} (C_1', h, \sigma) \qquad (C_2, h, \sigma) \xrightarrow{\mathit{recv}(v_1, v_2)} (C_2', h, \sigma')}{(C_1 \parallel C_2, h, \sigma) \xrightarrow{\mathit{comm}(v_1, v_2)} (C_1' \parallel C_2', h, \sigma')}$$

$$\frac{(C_1, h, \sigma) \xrightarrow{\mathit{recv}(v_1, v_2)} (C_1', h, \sigma') \qquad (C_2, h, \sigma) \xrightarrow{\mathit{send}(v_1, v_2)} (C_2', h, \sigma)}{(C_1 \parallel C_2, h, \sigma) \xrightarrow{\mathit{comm}(v_1, v_2)} (C_1' \parallel C_2', h, \sigma')}$$

Figure 7.2: Selected reduction rules of the small-step semantics of programs.

Clarifying the standard connectives, ε is the empty process without behaviour, and δ is the deadlocked process that neither progresses nor terminates. The process $\Sigma_x\, P$ is the infinite summation $P[x/v_0] + P[x/v_1] + \cdots$ over all values $v_0, v_1, ... \in \mathit{Val}$. Sometimes $\Sigma_{x_0,\ldots,x_n}\, P$ is written to abbreviate $\Sigma_{x_0} \cdots \Sigma_{x_n}\, P$. The guarded process $b : P$ behaves as P if the guard b holds, and otherwise behaves as δ. The process P^* is the Kleene iteration of P and denotes a sequence of zero or more P's. The infinite iteration of P is derived to be $P^\omega \triangleq P^* \cdot \delta$.

Since processes are used to reason about the send/receive behaviour of programs, this process algebra language exclusively supports two actions, $\mathsf{send}(e_1, e_2)$ and $\mathsf{recv}(e_1, e_2)$, for sending and receiving data elements e_1 together with a message tag e_2, respectively.

Finally, $?b$ is the *assertive process*, which is very similar to guarded processes: $?b$ is behaviourally equivalent to δ in case b does not hold. However, assertive processes have a special role in our approach: they are the main subject of process-algebraic analysis, as they encode the properties b to verify, as logical assertions. Moreover, they are a key component in connecting process-algebraic reasoning with deductive reasoning, as their properties can be relied upon in the deductive proofs of programs via the **query** b ghost command.

The function fv is overloaded to determine the set of unbound variables in process

terms, notably $\mathsf{fv}(\Sigma_x P) \triangleq \mathsf{fv}(P) \setminus \{x\}$. As always, any process P is defined to be *closed* if $\mathsf{fv}(P) = \emptyset$.

7.2.2.1 Semantics of Processes

Figure 7.3 presents the operational semantics of processes, which is defined in terms of a labelled binary reduction relation $\dot{\longrightarrow} \subseteq \mathit{Proc} \times \mathit{ProcLabel} \times \mathit{Proc}$ between processes. The labels of the reduction rules are defined as follows.

Definition 7.2.4 (Process transition labels)**.**

$$\alpha \in \mathit{ProcLabel} \ ::= \ \mathit{send}(v, v) \mid \mathit{recv}(v, v) \mid \mathit{comm}(v, v) \mid \mathit{assn}$$

The labels *send*, *recv* and *comm* are used in the same manner as those of program transitions, whereas the label *assn* indicates reductions of assertional processes.

The reduction rules are mostly standard [FZ94, GM14]. Processes are assumed to be closed as a wellformedness condition, which prevents the need to include stores in the transition rules. Moreover, it is common to use an explicit notion of *successful termination* in process algebra with ε [Bae00]. More specifically, $P \downarrow$ intuitively means that P has the choice to have no further behaviour and thus to behave as ε:

Lemma 7.2.1. *If $P \downarrow$, then $P \cong P + \varepsilon$.*

The send and recv actions communicate in the sense that they synchronise as a *comm* transition.

The property of interest for process-algebraic verification is to check for absence of faults. Any closed process P *exhibits a fault*, denoted $P \longrightarrow \lightning$, if P is able to violate an assertion.

Example 7.2.1 (Faulting summation)**.** *Below is a proof derivation showing that the process $\Sigma_{x,y} ?(x < y)$ is faulting.*

$$\dfrac{\dfrac{\dfrac{\neg[\![24 < 2]\!]}{?(24 < 2) \longrightarrow \lightning}}{\Sigma_y ?(24 < y) \longrightarrow \lightning}}{\Sigma_{x,y} ?(x < y) \longrightarrow \lightning}$$

Furthermore, any process P is defined to be *safe*, written $P\checkmark$, if P can never reach a state that exhibits a fault, while following the reduction rules of the operational semantics.

Successful termination

$$\varepsilon\downarrow \qquad P^*\downarrow \qquad \frac{P\downarrow \quad Q\downarrow}{P\cdot Q\downarrow} \qquad \frac{P\downarrow}{P+Q\downarrow} \qquad \frac{Q\downarrow}{P+Q\downarrow} \qquad \frac{P\downarrow \quad Q\downarrow}{P\parallel Q\downarrow}$$

$$\frac{P[x/v]\downarrow}{\Sigma_x P\downarrow} \qquad \frac{[\![b]\!] \quad P\downarrow}{b:P\downarrow}$$

Operational semantics

$$\mathsf{send}(e_1,e_2)\xrightarrow{\mathit{send}([\![e_1]\!],[\![e_2]\!])}\varepsilon \qquad \mathsf{recv}(e_1,e_2)\xrightarrow{\mathit{recv}([\![e_1]\!],[\![e_2]\!])}\varepsilon \qquad \frac{[\![b]\!]}{?b\xrightarrow{\mathit{assn}}\varepsilon}$$

$$\frac{P\xrightarrow{\alpha}P'}{P\cdot Q\xrightarrow{\alpha}P'\cdot Q} \qquad \frac{P\downarrow \quad Q\xrightarrow{\alpha}Q'}{P\cdot Q\xrightarrow{\alpha}Q'} \qquad \frac{P\xrightarrow{\alpha}P'}{P+Q\xrightarrow{\alpha}P'} \qquad \frac{Q\xrightarrow{\alpha}Q'}{P+Q\xrightarrow{\alpha}Q'}$$

$$\frac{P\xrightarrow{\alpha}P'}{P\parallel Q\xrightarrow{\alpha}P'\parallel Q} \qquad \frac{Q\xrightarrow{\alpha}Q'}{P\parallel Q'\xrightarrow{\alpha}P\parallel Q'} \qquad \frac{P[x/v]\xrightarrow{\alpha}P'}{\Sigma_x P\xrightarrow{\alpha}P'}$$

$$\frac{[\![b]\!] \quad P\xrightarrow{\alpha}P'}{b:P\xrightarrow{\alpha}P'} \qquad \frac{P\xrightarrow{\alpha}P'}{P^*\xrightarrow{\alpha}P'\cdot P^*} \qquad \frac{P\xrightarrow{\mathit{send}(v_1,v_2)}P' \quad Q\xrightarrow{\mathit{recv}(v_1,v_2)}Q'}{P\parallel Q\xrightarrow{\mathit{comm}(v_1,v_2)}P'\parallel Q'}$$

$$\frac{P\xrightarrow{\mathit{recv}(v_1,v_2)}P' \quad Q\xrightarrow{\mathit{send}(v_1,v_2)}Q'}{P\parallel Q\xrightarrow{\mathit{comm}(v_1,v_2)}P'\parallel Q'}$$

Fault semantics

$$\frac{\neg[\![b]\!]}{?b\longrightarrow\lightning} \qquad \frac{P\longrightarrow\lightning}{P\cdot Q\longrightarrow\lightning} \qquad \frac{P\downarrow \quad Q\longrightarrow\lightning}{P\cdot Q\longrightarrow\lightning} \qquad \frac{P\longrightarrow\lightning}{P+Q\longrightarrow\lightning} \qquad \frac{Q\longrightarrow\lightning}{P+Q\longrightarrow\lightning}$$

$$\frac{P\longrightarrow\lightning}{P\parallel Q\longrightarrow\lightning} \qquad \frac{Q\longrightarrow\lightning}{P\parallel Q\longrightarrow\lightning} \qquad \frac{P[x/v]\longrightarrow\lightning}{\Sigma_x P\longrightarrow\lightning} \qquad \frac{[\![b]\!] \quad P\longrightarrow\lightning}{b:P\longrightarrow\lightning} \qquad \frac{P\longrightarrow\lightning}{P^*\longrightarrow\lightning}$$

Figure 7.3: The small-step operational semantics of processes.

Definition 7.2.5 (Process safety)**.** *The $\checkmark$ predicate is coinductively defined such that, if $P\checkmark$ holds, then $P \not\longrightarrow \lightning$, and $P \xrightarrow{\alpha} P'$ implies $P'\checkmark$ for any α and P'.*

Given any closed process P, determining whether $P\checkmark$ holds can straightforwardly and mechanically be reduced to an mCRL2 model checking problem. This is done by modelling an explicit fault state that is reachable whenever an assertive process is violated, as a distinctive $\lightning$ action. Checking for fault absence is then reduced to checking the μ-calculus formula $[\mathsf{true}^* \cdot \lightning]\mathsf{false}$ on the translated model, meaning "no faulty transitions are ever reachable".

Lemma 7.2.2. *Process safety has the following properties.*

1. *If $(P \cdot Q)\checkmark$ or $(P + Q)\checkmark$ or $(P \parallel Q)\checkmark$, then $P\checkmark$ and $Q\checkmark$.*
2. *If $(b : P)\checkmark$ and $[\![b]\!]$, then $P\checkmark$.*
3. *If $P\checkmark$ and $Q\checkmark$, then also $(P \cdot Q)\checkmark$ and $(P + Q)\checkmark$ and $(P \parallel Q)\checkmark$.*
4. *$(\Sigma_x P)\checkmark$ if and only if $\forall v \,.\, P[x/v]\checkmark$.*
5. *If $P\checkmark$, then $P^*\checkmark$.*

Process bisimilarity is defined as usual, and preserves faults and termination.

Definition 7.2.6 (Bisimulation)**.** *A binary relation $\mathscr{R} \subseteq Proc \times Proc$ is a* bisimulation relation *over closed processes if, whenever $P\mathscr{R}Q$, then*

- *$P\downarrow$ if and only if $Q\downarrow$.*
- *$P \longrightarrow \lightning$ if and only if $Q \longrightarrow \lightning$.*
- *If $P \xrightarrow{\alpha} P'$, then there exists a Q' such that $Q \xrightarrow{\alpha} Q'$ and $P'\mathscr{R}Q'$.*
- *If $Q \xrightarrow{\alpha} Q'$, then there exists a P' such that $P \xrightarrow{\alpha} P'$ and $P'\mathscr{R}Q'$.*

Two closed process P and Q are defined to be *bisimilar* or *bisimulation equivalent*, denoted $P \cong Q$, if there exists a bisimulation relation $\mathscr{R}$ such that $P\mathscr{R}Q$. Any bisimulation relation constitutes an equivalence relation. Appendix B gives an overview of standard bisimulation equivalences that are sound for this language. As usual, bisimilarity is a *congruence* for all process-algebraic connectives. Moreover, process safety is closed under bisimilarity:

Lemma 7.2.3. *If $P\checkmark$ and $P \cong Q$, then $Q\checkmark$.*

7.3 Verification Example

Before discussing the logical details of our approach, let us first demonstrate it on the example program of Section 7.1.1. Application of the technique consists of the following three steps:

```
1  map sort: List(Nat) → List(Nat);
2  var x: Nat, xs: List(Nat);
3  eqn sort(⟨⟩) = ⟨⟩;
4      sort(⟨x⟩) = ⟨x⟩;
5      sort(x :: xs) = merge(⟨x⟩, sort(xs));
6
7  map merge: List(Nat) × List(Nat) → List(Nat);
8  var x: Nat, y: Nat, xs: List(Nat), ys: List(Nat);
9  eqn merge(⟨⟩, ⟨⟩) = ⟨⟩;
10     merge(xs, ⟨⟩) = xs;
11     merge(⟨⟩, xs) = xs;
12     x ≤ y → merge(x :: xs, y :: ys) = x :: merge(xs, y :: ys);
13     x > y → merge(x :: xs, y :: ys) = y :: merge(x :: xs, ys);
```

Figure 7.4: An axiomatic description of a sorting algorithm, defined in mCRL2.

1. Constructing a process-algebraic model that captures the program's send/recv behaviour;
2. Analysing the model to determine whether the value received by Thread 1 is always the sorted sequence $\langle 4, 5, 6, 7, 8\rangle$, via a reduction to an mCRL2 model checking problem; and
3. Deductively verifying that the program correctly implements the process-algebraic model with respect to send/receive behaviour, by using concurrent separation logic.

The remainder of this section discusses each of these three steps in detail.

Step 1: Constructing a process-algebraic model. The communication behaviour of the example program can straightforwardly be captured as a process $\mathsf{P} = \mathsf{P}_1 \parallel \mathsf{P}_2 \parallel \mathsf{P}_3$ (assuming that the expression language is rich enough to handle sequences), so that process P_i captures Thread i's send/receive behaviour, where

$$\mathsf{P}_1 \triangleq \mathsf{send}(\langle 4, 7, 5\rangle, 1) \cdot \Sigma_{xs}\, \mathsf{recv}(xs, 2) \cdot ?(xs = \langle 4, 5, 6, 7, 8\rangle)$$

$$\mathsf{P}_2 \triangleq \mathsf{P}_2'^{\omega} \text{ with } \mathsf{P}_2' \triangleq \Sigma_{ys,t}\, \mathsf{recv}(ys, t) \cdot \big(t = 1 : \mathsf{send}(ys +\!\!+ \langle 8, 6\rangle, 3) + t \neq 1 : \mathsf{send}(ys, 2)\big)$$

$$\mathsf{P}_3 \triangleq \mathsf{P}_3'^{\omega} \text{ with } \mathsf{P}_3' \triangleq \Sigma_{zs,t}\, \mathsf{recv}(zs, t) \cdot \mathsf{send}(\mathtt{sort}(zs), t)$$

Observe that P_1 encodes the property of interest as the assertion $?(xs = \langle 4, 5, 6, 7, 8\rangle)$. The validity of this assertion is checked by mCRL2 on the translated model, as

described in the next paragraph. Moreover, `sort` is assumed to be the functional description of a sorting algorithm. Such a description can axiomatically be defined in mCRL2, as shown in Figure 7.4. The `sort` mapping can easily act as a functional specification for the implementation of more intricate sorting algorithms like `ParSort`. Deductive verifiers are generally well-suited to relate such functional specifications to implementations via pre/postcondition reasoning:

```
1 ensures \result = sort(xs);
2 seq⟨nat⟩ ParSort(seq⟨nat⟩ xs) {
3   ...
4 }
```

Step 2: Analysing the process-algebraic model. The composite process P can straightforwardly be translated to mCRL2 input and be analysed. Our translation can be found at the online Git repository. This translation has been done manually, yet it would not be difficult to write a tool that does it mechanically (we are actively working on this).

Notably, assertive processes $?b$ are translated into $\mathsf{check}(b)$ actions; the action $\mathsf{check}(\mathsf{false})$ can be seen as the encoding of $\lightning$. Checking for process safety $\mathsf{P}\checkmark$ can be reduced to checking the μ-calculus formula $\phi = [\mathsf{true}^* \cdot \mathsf{check}(\mathsf{false})]\mathsf{false}$, stating that no $\mathsf{check}(\mathsf{false})$ action can ever be performed, or equivalently that the process is free of faults. mCRL2 can indeed confirm that P is fault-free by checking ϕ, and thus that the asserted property of interest holds. In Step 3 we formally prove that the program adheres to the communication behaviour described by P, which allows projecting these model checking results onto program behaviour.

Another question one might ask is: does Thread 1 always terminate? This can also be checked, using the fixed-point formula $\nu X.\langle\neg\mathsf{check}\rangle X$, meaning "there exists an infinite trace of non-check actions". This formula holds for P, meaning that from Step 3 it also follows that Thread 1 is not guaranteed to terminate. To see the corresponding behaviour on the program level, observe that Threads 2 and 3 may get entangled in a non-terminating communication loop (under unfair scheduling).

Step 3: Connecting processes to program behaviour. The final step is to deductively prove that Figure 7.1's program refines P, with respect to communication behaviour, using CSL. To do this, we extend CSL with predicates of the form $\mathsf{Proc}(P)$, which express that the remaining program will communicate as prescribed by the process P—the program's model. More specifically, the proof system enforces that every **send** (e, e') instruction must be prescribed by a

$\mathsf{Proc}(\mathsf{send}(e, e') \cdot P)$ predicate in the logic, and likewise for **recv**, thereby enforcing that the process-algebraic model can perform a matching send or recv action. These actions are then *consumed* in the logic, while following the structure of the program. Similarly, **query** b annotations must be prescribed by a $\mathsf{Proc}(?b \cdot P)$ predicate, and allow assuming b in the logic as result of Step 2, by which $?b$ is consumed from the Proc predicate.

Figure 7.5 illustrates this, by giving the intermediate steps of the proofs of Threads 1 and 3. An extra **query** annotation has been added in Thread 1's program for obtaining the asserted property from P_1. Moreover, the annotated **invariant** in Thread 3 is a loop invariant that states that $\mathsf{Proc}(\mathsf{P}_3'^{\omega})$ prescribes the communication behaviour of every loop iteration.

Another feature of the logic is that $\mathsf{Proc}(P)$ predicates can be *split* and *merged* along parallel compositions inside P, in the style of CSL. This is used in the top-level proof of the example program, shown in Figure 7.6. The $*$ connective is the *separating conjunction* from separation logic, which now expresses that different threads will use different parts of the model. This makes the approach both modular and compositional, by allowing the programs' top-level proof to be composed out of the individual independent proofs of its threads.

We encoded the program logic into the Viper concurrency verifier and used it to fully mechanise the deductive proof of the example program. The Viper files are available online at [Sup]. This encoding primarily consists of an axiomatic domain for processes, containing constructors for the process-algebraic connectives, supported by standard axioms of process algebra. The Proc assertions are then encoded as unary predicates over these process types. Viper can verify correctness of the example program in under 3 seconds.

We are investigating our verification technique on bigger case studies that involve distributed consensus protocols [Fok13], e.g., classical leader election [LL77, OBH16] and Paxos [Lam98].

7.4 Formalisation

This section discusses the assertion language and entailment rules of the program logic (§7.4.1), the Hoare-triple rules for message passing and querying (§7.4.2), and their soundness (§7.4.3).

7.4.1 Program Logic

The program logic extends intuitionistic concurrent separation logic [Vaf11, HDV11], where the assertion language is defined by the following grammar.

```
1  {Proc(P_1)}
2  {Proc(send(⟨4,7,5⟩,1) · Σ_x recv(x,2) · ?(x = ⟨4,5,6,7,8⟩))}
3  send (⟨4,7,5⟩,1);
4  {Proc(Σ_x recv(x,2) · ?(x = ⟨4,5,6,7,8⟩))}
5  xs := recv 2;
6  {Proc(?(xs = ⟨4,5,6,7,8⟩))}
7  {Proc(?(xs = ⟨4,5,6,7,8⟩) · ε)}
8  query xs = ⟨4,5,6,7,8⟩;
9  {Proc(ε) ∗ xs = ⟨4,5,6,7,8⟩}
10 assert xs = ⟨4,5,6,7,8⟩;
11 {Proc(ε) ∗ xs = ⟨4,5,6,7,8⟩}
```

(a) Proof of Thread 1's program.

```
1  {Proc(P_3)}
2  {Proc(P_3^ω)}
3  while (true) invariant Proc(P_3'^ω) {
4    {Proc(P_3'^ω)}
5    {Proc(P_3' · P_3'^ω)}
6    {Proc(Σ_{zs,t} recv(zs,t) · send(sort(zs),t) · P_3'^ω)}
7    (zs,t) := recv;
8    {Proc(send(sort(zs),t) · P_3'^ω)}
9    zs' := ParSort(zs);
10   {Proc(send(sort(zs),t) · P_3'^ω) ∗ zs' = sort(zs)}
11   {Proc(send(zs',t) · P_3'^ω)}
12   send (zs',t);
13   {Proc(P_3'^ω)}
14 }
15 {Proc(P_3'^ω) ∗ false} // i.e., this point of the code is never reached
```

(b) Proof of Thread 3's program.

Figure 7.5: Proofs for Threads 1 and 3 of the example program. Thread 2 is proven likewise. All the intermediate proof steps are coloured purple.

$$\{\mathsf{Proc}(\mathsf{P}_1 \parallel \mathsf{P}_2 \parallel \mathsf{P}_3)\}$$
$$\{\mathsf{Proc}(\mathsf{P}_1) * \mathsf{Proc}(\mathsf{P}_2) * \mathsf{Proc}(\mathsf{P}_3)\}$$
$$\begin{array}{c|c|c} \{\mathsf{Proc}(\mathsf{P}_1)\} & \{\mathsf{Proc}(\mathsf{P}_2)\} & \{\mathsf{Proc}(\mathsf{P}_3)\} \\ \textit{Thread 1's program} & \textit{Thread 2's program} & \textit{Thread 3's program} \\ \{\mathsf{Proc}(\varepsilon)\} & \{\mathsf{Proc}(\mathsf{P}_2'^{\omega}) * \mathsf{false}\} & \{\mathsf{Proc}(\mathsf{P}_3'^{\omega}) * \mathsf{false}\} \end{array}$$
$$\{\mathsf{Proc}(\varepsilon) * \mathsf{Proc}(\mathsf{P}_2'^{\omega}) * \mathsf{false} * \mathsf{Proc}(\mathsf{P}_3'^{\omega}) * \mathsf{false}\}$$
$$\{\mathsf{false}\}$$

Figure 7.6: The top-level specification and ownership distribution of the example program.

Definition 7.4.1 (Assertions)**.**

$$\mathcal{P}, \mathcal{Q}, \cdots \in \mathit{Assn} \;::=\; b \mid \forall x.\mathcal{P} \mid \exists x.\mathcal{P} \mid \mathcal{P} \vee \mathcal{Q} \mid \mathcal{P} * \mathcal{Q} \mid \mathcal{P} \mathrel{-\!\!*} \mathcal{Q} \mid e \hookrightarrow_\pi e \mid \mathsf{Proc}(P) \mid P \approx Q$$

The assertion $e_1 \hookrightarrow_\pi e_2$ is the *heap ownership assertion* and expresses the knowledge that the heap holds the value e_2 at heap location e_1. Moreover, $\pi \in (0, 1]_{\mathbb{Q}}$ is a *fractional permission* in the style of Boyland [Boy03] and determines the type of ownership: write access to e_1 is provided in case $\pi = 1$, and read access is provided in case $0 < \pi < 1$.

The $*$ connective is the *separating conjunction* from separation logic. The assertion $\mathcal{P} * \mathcal{Q}$ expresses that the ownerships captured by $\mathcal{P}$ and $\mathcal{Q}$ are *disjoint*, e.g., it is disallowed that both express write access to the same heap location. The $-\!\!*$ connective is known as the *magic wand* and describes hypothetical modifications of the current state.

The assertion $\mathsf{Proc}(P)$ expresses the ownership of the right to send and receive messages as prescribed by the process P. Here P may contain free variables and may be replaced by any process bisimilar to P. To handle such replacements, the assertion $P \approx Q$ can be used, which expresses that P and Q are bisimilar in the current context. To give an example, one might wish to deduce that $\mathsf{Proc}(0 < x : P) * x = 2$ entails $\mathsf{Proc}(P)$. Even though $0 < x : P$ has free variables, it is used in a context where x equals 2, and therefore $0 < x : P \approx P$ can be established. The next paragraph discusses the entailment rules for these deductions.

7.4.1.1 Proof Rules

Figure 7.7 shows an excerpt of the proof rules, which are given as sequents of the form $\vdash \mathcal{P}$ and $\mathcal{P} \vdash \mathcal{Q}$. A complete overview is given in Appendix B. The notation

$\hookrightarrow$-SPLIT-MERGE
$e_1 \hookrightarrow_{\pi_1+\pi_2} e_2 \dashv\vdash e_1 \hookrightarrow_{\pi_1} e_2 * e_1 \hookrightarrow_{\pi_2} e_2$

Proc-SPLIT-MERGE
$\mathsf{Proc}(P \parallel Q) \dashv\vdash \mathsf{Proc}(P) * \mathsf{Proc}(Q)$

Proc-$\approx$
$\mathsf{Proc}(P) * P \approx Q \vdash \mathsf{Proc}(Q)$

$\approx$-BISIM
$$\frac{P \cong Q}{\vdash P \approx Q}$$

$\approx$-REFL
$\vdash P \approx P$

$\approx$-SYMM
$P \approx Q \vdash Q \approx P$

$\approx$-TRANS
$P \approx Q * Q \approx R \vdash P \approx R$

$\approx$-COND-TRUE
$b \vdash b : P \approx P$

$\approx$-COND-FALSE
$b \vdash \neg b : P \approx \delta$

Figure 7.7: An except of the entailment rules of the program logic.

$\mathcal{P} \dashv\vdash \mathcal{Q}$ is shorthand for $\mathcal{P} \vdash \mathcal{Q}$ and $\mathcal{Q} \vdash \mathcal{P}$. All proof rules are sound in the standard sense.

The $\hookrightarrow$-SPLIT-MERGE rule expresses that heap ownership predicates can be *split* (in the left-to-right direction) and *merged* (right-to-left) along π, allowing heap ownership to be distributed over different threads. Likewise, Proc-SPLIT-MERGE allows process predicates to be split and merged along the parallel composition, to allow different threads to communicate as prescribed by the different parts of the process-algebraic model. Process terms inside Proc predicates may be replaced by bisimilar ones via Proc-$\approx$. This rule can be used to rewrite process terms to a canonical form used by some other proof rules. The $\approx$ connective enjoys properties similar to $\cong$: it is an equivalence relation with respect to $*$, as shown by $\approx$-REFL, $\approx$-SYMM and $\approx$-TRANS, and is a congruence for all process-algebraic connectives. Finally, $\approx$ allows using contextual information to resolve guards, via $\approx$-COND-TRUE and $\approx$-COND-FALSE.

Example 7.4.1 (Using $\approx$ to eliminate conditional processes)**.**

$$\dfrac{\dfrac{\dfrac{}{b \vdash b : P \approx P}\,(4)}{\mathsf{Proc}(b : P) * b \vdash \mathsf{Proc}(b : P) * b : P \approx P}\,(3) \quad \dfrac{}{\mathsf{Proc}(b : P) * b : P \approx P \vdash \mathsf{Proc}(P)}\,(2)}{\mathsf{Proc}(b : P) * b \vdash \mathsf{Proc}(P)}\,(1)$$

Rule application (1) is transitivity; (2) is an application of Proc-$\approx$*; (3) is monotonicity with respect to the left operand of the* $*$*; and (4) is exactly* $\approx$-COND-TRUE*.*

7.4.1.2 Semantics of Assertions

The semantics of assertions is given as a modelling relation $\iota, \sigma, P \models \mathcal{P}$, where the models are abstractions of program states (these can also be seen as partial

program states). These state abstractions consist of three components, the first being a *permission heap*. Permission heaps $\iota \in \mathit{PermHeap} \triangleq \mathit{Val} \rightharpoonup_{\mathsf{fin}} (0,1]_{\mathbb{Q}} \times \mathit{Val}$ extend normal heaps by associating a fractional permission to each occupied heap cell. The second component is an ordinary store σ, and the last component is a *closed* process P that determines the state of the process-algebraic model that may be described by $\mathcal{P}$.

The semantics of assertions relies on the notions of disjointness and disjoint union of permission heaps. Two permission heaps, ι_1 and ι_2, are said to be *disjoint*, written $\iota_1 \perp \iota_2$, if they agree on their contents and the pairwise addition of the fractional permissions they store are again valid fractional permissions. Furthermore, the *disjoint union* of ι_1 and ι_2, which is written $\iota_1 \uplus \iota_2$, is defined to be the pairwise union of all their disjoint heap cells.

Definition 7.4.2 (Disjointness of permission heaps)**.**

$$\iota_1 \perp \iota_2 \triangleq \forall v \in \mathit{dom}(\iota_1) \cap \mathit{dom}(\iota_2) \,.\, \iota_1(v) \perp_{\mathit{cell}} \iota_2(v), \text{ where}$$
$$(\pi_1, v_1) \perp_{\mathit{cell}} (\pi_2, v_2) \triangleq v_1 = v_2 \wedge \pi_1 + \pi_2 \in (0,1]_{\mathbb{Q}}$$

Definition 7.4.3 (Disjoint union of permission heaps)**.**

$$\iota_1 \uplus \iota_2 \triangleq \lambda v \,.\, \begin{cases} \iota_1(v) & \text{if } v \in \mathsf{dom}(\iota_1) \setminus \mathsf{dom}(\iota_2) \\ \iota_2(v) & \text{if } v \in \mathsf{dom}(\iota_2) \setminus \mathsf{dom}(\iota_1) \\ (\pi_1 + \pi_2, v') & \text{if } \iota_1(v) = (\pi_1, v') \wedge \iota_2(v) = (\pi_2, v') \wedge \\ & \qquad \pi_1 + \pi_2 \in (0,1]_{\mathbb{Q}} \end{cases}$$

As one would expect, $\uplus$ is associative and commutative, and $\perp$ is symmetric. Most importantly, if $\iota_1 \perp \iota_2$, then $\iota_1 \uplus \iota_2$ does not lose information with respect to ι_1 and ι_2.

Lemma 7.4.1. *The operations $\perp$ and $\uplus$ satisfy the following properties:*

1. $\iota_1 \uplus (\iota_2 \uplus \iota_3) = (\iota_1 \uplus \iota_2) \uplus \iota_3$ *and* $\iota_1 \uplus \iota_2 = \iota_2 \uplus \iota_1$.
2. *If* $\iota_1 \perp \iota_2$, *then* $\iota_2 \perp \iota_1$.
3. *If* $\iota_1 \perp \iota_2$ *and* $\iota_1 \uplus \iota_2 \perp \iota_3$, *then* $\iota_2 \perp \iota_3$ *and* $\iota_1 \perp \iota_2 \uplus \iota_3$.

The semantics of assertions also relies on a closure operation for closing process terms. Given any process P, the *σ-closure* of P is defined to be $P[\sigma] \triangleq P[x/\sigma(x)]_{\forall x \in \mathsf{fv}(P)}$.

Definition 7.4.4 (Semantics of assertions)**.** *The interpretation of assertions, written $\iota, \sigma, P \models \mathcal{P}$, is defined by structural induction on $\mathcal{P}$ provided that P is closed,*

in the following way:

$$
\begin{array}{lll}
\iota, \sigma, P \models b & \textit{iff} & [\![b]\!]\sigma \\
\iota, \sigma, P \models \forall x.\mathcal{P} & \textit{iff} & \forall v \,.\, \iota, \sigma[x \mapsto v], P \models \mathcal{P} \\
\iota, \sigma, P \models \exists x.\mathcal{P} & \textit{iff} & \exists v \,.\, \iota, \sigma[x \mapsto v], P \models \mathcal{P} \\
\iota, \sigma, P \models \mathcal{P} \vee \mathcal{Q} & \textit{iff} & \iota, \sigma, P \models \mathcal{P} \vee \iota, \sigma, P \models \mathcal{Q} \\
\iota, \sigma, P \models \mathcal{P} * \mathcal{Q} & \textit{iff} & \exists \iota_1, P_1, \iota_2, P_2 \,.\, \iota_1 \perp \iota_2 \wedge \iota = \iota_1 \uplus \iota_2 \wedge P \cong P_1 \,\|\, P_2 \wedge \\
 & & \quad \iota_1, \sigma, P_1 \models \mathcal{P} \wedge \iota_2, \sigma, P_2 \models \mathcal{Q} \\
\iota, \sigma, P \models \mathcal{P} \mathrel{-\!\!*} \mathcal{Q} & \textit{iff} & \forall \iota', P' \,.\, (\iota \perp \iota' \wedge \iota', \sigma, P' \models \mathcal{P}) \Rightarrow \iota \uplus \iota', \sigma, P \,\|\, P' \models \mathcal{Q} \\
\iota, \sigma, P \models e_1 \hookrightarrow_\pi e_2 & \textit{iff} & \exists \pi' \,.\, \iota([\![e_1]\!]\sigma) = (\pi', [\![e_2]\!]\sigma) \wedge \pi \leq \pi' \\
\iota, \sigma, P \models \mathsf{Proc}(Q) & \textit{iff} & \exists Q' \,.\, P \cong Q[\sigma] \parallel Q' \\
\iota, \sigma, P \models Q_1 \approx Q_2 & \textit{iff} & Q_1[\sigma] \cong Q_2[\sigma]
\end{array}
$$

All assertions are interpreted intuitionistically in the standard sense [Vaf11], except for the last two cases, which cover the process-algebraic extensions. Both cases rely on σ-closures to resolve any free variables that may have been introduced by some other proof rules (e.g., the Hoare logic rule for **recv** instructions may do this). Process ownership assertions $\mathsf{Proc}(Q)$ are satisfied if there exists a (necessarily closed) process Q', which is the "framed" process that is maintained by the environmental threads, such that P is bisimilar to $Q[\sigma] \parallel Q'$. The intuition here is that P must have at least the behaviour that is abstractly described by Q. Finally, $Q_1 \approx Q_2$ is satisfied if Q_1 and Q_2 are bisimilar with respect to the current state.

Lemma 7.4.2. *The interpretation of assertions has the following properties:*

1. *If $\iota, \sigma, P \models \mathcal{P}$ and $P \cong Q$, then $\iota, \sigma, Q \models \mathcal{P}$.*
2. *$\iota, \sigma, P \models \mathcal{P}[x/e]$ if and only if $\iota, \sigma[x \mapsto [\![e]\!]\sigma], P \models \mathcal{P}$.*
3. *If $\iota_1, \sigma, P_1 \models \mathcal{P}$ and $\iota_1 \perp \iota_2$, then $\iota_1 \uplus \iota_2, \sigma, P_1 \parallel P_2 \models \mathcal{P}$.*

Law 1 states that the interpretation of assertions is insensitive to replacing the process components by bisimilar ones, inside the models of the logic. Law 2 relates substitution in assertions with the evaluation of expressions. Law 3 is known as *weakening* (or monotonicity), and is a crucial property in intuitionistic separation logic. Its intuitive meaning is that extending the heap or the process-algebraic model only makes the property "more true".

7.4.2 Program Judgments

Judgments of programs are of the usual form $\mathcal{I} \vdash \{\mathcal{P}\}\, C \,\{\mathcal{Q}\}$ and indicate partial correctness of the program C, where $\mathcal{I} \in \mathit{Assn}$ is known as the *resource*

$$\text{HT-SEND}\quad \mathcal{I} \vdash \{\mathsf{Proc}(\mathsf{send}(e_1, e_2) \cdot P)\}\ \mathbf{send}\ (e_1, e_2)\ \{\mathsf{Proc}(P)\}$$

$$\text{HT-RECV}\quad \frac{x \notin \mathsf{fv}(\mathcal{I}) \cup \mathsf{fv}(P) \qquad y \notin \mathsf{fv}(e)}{\mathcal{I} \vdash \{\mathsf{Proc}(\Sigma_y\, \mathsf{recv}(y, e) \cdot P)\}\, x := \mathbf{recv}\ e\, \{\mathsf{Proc}(P[y/x])\}}$$

$$\text{HT-RECV-WILDCARD}\quad \frac{x_1, x_2 \notin \mathsf{fv}(\mathcal{I}) \cup \mathsf{fv}(P) \qquad \{x_1, y_1\} \cap \{x_2, y_2\} = \emptyset}{\mathcal{I} \vdash \{\mathsf{Proc}(\Sigma_{y_1, y_2}\, \mathsf{recv}(y_1, y_2) \cdot P)\}\, (x_1, x_2) := \mathbf{recv}\, \{\mathsf{Proc}(P[y_1/x_1][y_2/x_2])\}}$$

$$\text{HT-QUERY}\quad \mathcal{I} \vdash \{\mathsf{Proc}(?b \cdot P)\}\ \mathbf{query}\ b\ \{\mathsf{Proc}(P) * b\}$$

Figure 7.8: An excerpt of the proof rules for program judgments.

invariant [Bro07]. Their intuitive meaning is that, starting from an initial state satisfying $\mathcal{P} * \mathcal{I}$, the invariant $\mathcal{I}$ is maintained throughout execution of C, and any final state upon termination of C will satisfy $\mathcal{Q} * \mathcal{I}$.

Figure 7.8 gives an overview of the new proof rules that are specific to handling processes. All other proof rules are standard in CSL and are therefore deferred to Appendix B. The HT-SEND rule expresses that, as a precondition, any **send** command in the program must be prescribed by a matching send action in the process-algebraic model. Furthermore, it reduces the process term by ensuring a $\mathsf{Proc}(P)$ predicate, with P the leftover process after the performance of send. The HT-RECV rule is similar in the sense that any $x :=$ **recv** e instruction must be matched by a $\mathsf{recv}(y, e)$ action, but now y can be any message. Process-algebraic summation is used here to quantify over all possible messages to receive, and in the post-state of HT-RECV this message is bound to x—the received message. For wildcard receives (HT-RECV-WILDCARD) both the message and the tag are quantified over using summation. Finally, HT-QUERY allows "query"-ing for properties that are verified during process-algebraic analysis. Recall that the main objective of process-algebraic analysis is to verify that every reachable assertive process $?b$ is satisfied. If this is the case, then b can be used in the program logic, since the send/receive behaviour of the program has been proven to adhere to the process-algebraic specification of its communication behaviour, as a result of the proof rules for **send** and **recv**.

$$\frac{P \xrightarrow{\mathit{send}(v_1,v_2)} P' \qquad (C,h,\sigma) \xrightarrow{\mathit{send}(v_1,v_2)} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{send}(v_1,v_2)}\!\!\!\!\rightarrow (C',P',h',\sigma')} \;\twoheadrightarrow\text{-SEND}$$

$$\frac{P \xrightarrow{\mathit{recv}(v_1,v_2)} P' \qquad (C,h,\sigma) \xrightarrow{\mathit{recv}(v_1,v_2)} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{recv}(v_1,v_2)}\!\!\!\!\rightarrow (C',P',h',\sigma')} \;\twoheadrightarrow\text{-RECV}$$

$$\frac{P \xrightarrow{\mathit{comm}(v_1,v_2)} P' \qquad (C,h,\sigma) \xrightarrow{\mathit{comm}(v_1,v_2)} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{comm}(v_1,v_2)}\!\!\!\!\rightarrow (C',P',h',\sigma')} \;\twoheadrightarrow\text{-COMM}$$

$$\frac{P \xrightarrow{\mathit{assn}} P' \qquad (C,h,\sigma) \xrightarrow{\mathit{qry}} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{qry}}\!\!\!\!\rightarrow (C',P',h',\sigma')} \;\twoheadrightarrow\text{-ASSN}$$

$$\frac{(C,h,\sigma) \xrightarrow{\mathit{cmp}} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{cmp}}\!\!\!\!\rightarrow (C',P,h',\sigma')} \;\twoheadrightarrow\text{-CMP}$$

$$\frac{P \cong P' \qquad (C,P',h,\sigma) \xrightarrow{l}\!\!\!\!\rightarrow (C',Q',h',\sigma') \qquad Q' \cong Q}{(C,P,h,\sigma) \xrightarrow{l}\!\!\!\!\rightarrow (C',Q,h',\sigma')} \;\twoheadrightarrow\text{-STRUCT}$$

Figure 7.9: The instrumented operational semantics that executes programs in lock-step with processes.

7.4.3 Soundness

The soundness statement of the logic relates axiomatic judgments of programs (§7.4.2) to the operational meaning of programs (§7.2.1). This soundness argument guarantees freedom of data-races, memory safety, and compliance of pre- and postconditions, for any program for which a proof can be derived. The proof rules of Figure 7.8 ensure that every proof derivation encodes that the program synchronises with its process-algebraic model. To formulate the soundness statement, this axiomatic notion of synchronisation thus needs to have a matching operational notion of synchronisation. This is defined in terms of an *instrumented semantics* that executes programs in lock-step with their process-algebraic models. The transition rules are shown in Figure 7.9 and are expressed as a labelled binary reduction relation $\cdot \xrightarrow{l}\!\!\!\!\rightarrow \cdot$ between program configurations, extended with a (closed) process component.

The semantics of program judgments is defined in terms of an auxiliary predicate $\mathsf{safe}(C, \iota, \sigma, P, \mathcal{I}, \mathcal{Q})$, stating that C (i) executes safely for any number of execution

steps with respect to the abstract program state (ι, σ, P); (ii) will preserve the invariant $\mathcal{I}$ throughout its execution; and (iii) will satisfy the postcondition $\mathcal{Q}$ upon termination. To elaborate on (i), a *safe execution of* C means that C is data-race free, memory-safe, and synchronises with P with respect to $\dot{\longrightarrow\!\!\!\!\!\rightarrow}$.

To relate abstract program state to concrete program state, the following concretisation operation is used.

Definition 7.4.5 (Permission heap concretisation)**.** *The concretisation operation* $\lfloor \cdot \rfloor : \mathit{PermHeap} \to \mathit{Heap}$ *is used, for permission heaps is defined as follows:*

$$\lfloor \iota \rfloor \triangleq \lambda v \in \mathit{Val}\,.\, \begin{cases} v' & \text{if } \iota(v) = (\pi, v') \\ \mathit{undefined} & \text{if } \iota(v) \text{ is undefined} \end{cases}$$

Definition 7.4.6 (Execution safety)**.** *The* safe *predicate is coinductively defined so that, if* $\mathsf{safe}(C, \iota, \sigma, P, \mathcal{I}, \mathcal{Q})$ *holds, then*

- *If* $C = \mathbf{skip}$*, then* $\iota, \sigma, P \models \mathcal{Q}$*.*
- C *cannot perform a* data-race *or* memory violation *from the current state (the exact formal meaning of these notions are likewise to the ones defined in Chapter 5, for the program models in the shared-memory setting).*
- *For any* $\iota_I, \iota_F, P_I, C', h', \sigma'$ *and* l*, if*

  1. $\iota \perp \iota_I$ *and* $\iota \uplus \iota_I \perp \iota_F$*, and*
  2. $\neg\mathsf{locked}(C)$ *implies* $\iota_I, \sigma, P_I \models \mathcal{I}$*, and*
  3. $(P \parallel P_I)\checkmark$ *and* $(C, \lfloor \iota \uplus \iota_I \uplus \iota_F \rfloor, \sigma) \xrightarrow{l} (C', h', \sigma')$*,*

  then there exists $\iota', \iota'_I, P', P'_I$ *such that*

  i. $\iota' \perp \iota'_I$ *and* $\iota' \uplus \iota'_I \perp \iota_F$ *and* $h' = \lfloor \iota' \uplus \iota'_I \uplus \iota_F \rfloor$*, and*
  ii. $\neg\mathsf{locked}(C')$ *implies* $\iota'_I, \sigma, P'_I \models \mathcal{I}$*, and*
  iii. $(P' \parallel P'_I)\checkmark$ *and* $(C, P \parallel P_I, \lfloor \iota \uplus \iota_I \uplus \iota_F \rfloor, \sigma) \dot{\xrightarrow{l}\!\!\!\!\!\rightarrow} (C', P' \parallel P'_I, h', \sigma')$*, and*
  iv. $\mathsf{safe}(C', \iota', \sigma', P', \mathcal{I}, \mathcal{Q})$*.*

The above definition is based on the similar well-known inductive notion of *configuration safety* of Vafeiadis [Vaf11]. Our definition however is coinductive rather than inductive, as this matches more naturally with the coinductive definitions of bisimilarity and process safety. Moreover, it encodes that the program refines the process with respect to send/receive behaviour: any execution step of the program (*3.*) must be matched by the process-algebraic model (*iii.*), and vice versa by definition of $\dot{\longrightarrow\!\!\!\!\!\rightarrow}$. The $\mathsf{locked}(C)$ predicate determines whether C is locked. Any program is defined to be *locked* if it executes an atomic (sub)program.

Definition 7.4.7 (Semantics of program judgments)**.**

$$\mathcal{I} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \triangleq \forall \iota, \sigma, P\,.\, P\checkmark \implies \iota, \sigma, P \models \mathcal{P} \implies \mathsf{safe}(C, \iota, \sigma, P, \mathcal{I}, \mathcal{Q}).$$

Theorem 7.4.3 (Soundness)**.** $\mathcal{I} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \implies \mathcal{I} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$.

The soundness proof has been fully mechanised using Coq. The Coq development can be found on the online Git repository that comes with this thesis.

7.5 Extensions

So far the presented approach only deals with synchronous message passing. However, the principles of the approach as discussed in Section 7.4 allow for easy extensions to also reason about asynchronous message passing, message loss, message duplication, and collective operations like barriers and broadcasts in the style of MPI [Mes].

The semantics of asynchronous message passing is that **send**s do not block while waiting for a matching **recv**, but instead push the message onto a message queue that is accessible by both threads. The specification language of mCRL2 is rich enough to model such queues, for example as a separate process $\mathsf{Queue}(\eta)$ with η some data-structure that stores messages in order (e.g., a mapping). Then, rather than letting send and recv communicate directly, they should instead communicate with Queue to push and pop messages into η. So in order to lift the verification approach to programs with an asynchronous communication semantics, one only has to make minor changes to the mCRL2 translation of processes.

Message loss can be integrated in a similar way, by introducing an extra process that "steals" pending messages. For example, one could analyse the process $P \parallel (\Sigma_{x,t}\, \mathsf{recv}(x,t))^\omega$ to reason about P's behaviour with the possibility of message loss. Message duplication can be modelled likewise as an extra process that sends multiple copies of any message it receives. Collective operations like barriers and broadcasts may be slightly more involved, as they require some extra bookkeeping, e.g., to administer which threads have already received a broadcasted message. However, all collective operations can be implemented using only (non-blocking) sends and receives [LZS17], and thus for verification purposes can be handled as such. This means that the approach also extends well to collective communication.

To reason about shared-memory concurrency in combination with message passing, one has to include the state and behaviour of the heap into the process-algebraic models. Consider for example the program $x := [e];\ \mathbf{send}\ (x, 1)$ that reads a value from the heap at location e and sends it with tag 1. This program is difficult to reason about in the current setting, as the heap may contain any value at

e, which complicates modelling on the process algebra level. However, observe that the heap can be seen as just another distributed system, where heap reading $x := [e]$ amounts to receiving data from the heap, and heap writing $[e] := e'$ amounts to sending data to the heap. Thus on the level of process algebra these can be modelled explicitly, e.g., in terms of $\mathsf{read}(x, e)$ and $\mathsf{write}(e, e')$ actions that communicate with a separate process $\mathsf{Heap}(\cdot)$ that maintains the shared state. One could then upgrade the proof rules for $x := [e]$ and $[e] := e'$ to also require matching $\mathsf{Proc}(\Sigma_x\, \mathsf{read}(x, e) \cdot P)$ and $\mathsf{Proc}(\mathsf{write}(e, e') \cdot P)$ predicates in their pre-state, like in the proof rules of Figure 7.8.

Finally, the current biggest limitation of our approach is that mCRL2 is primarily an explicit-state model checker, which limits its ability to reason symbolically about send/receive behaviour. Nonetheless, mCRL2 also comes with a symbolic back-end [NWG18] that, at the time of writing, can handle specifications of limited complexity [lps]. We already have some preliminary results on reasoning symbolically about process-algebraic models, and are actively collaborating with the developers of mCRL2 to improve this.

7.6 Related Work

There are many modern concurrency logics [DYDG+10, SB14, SBP13, TDB13, NLWSD14, Fen09, RPDYG14] that provide protocol-like specification mechanisms, to formally describe how shared state is allowed to evolve over time. Notably, Sergey et al. [SWT17] employ this notion in a distributed setting, by using state-transition systems combined with invariants as abstractions for the communication behaviour of distributed programs. All these program logics are however purely theoretical, or can at best semi-automatically be applied via a shallow embedding in Coq. Our approach distinguishes itself by focusing on usability rather than expressivity and targets automated concurrency verifiers instead, like the combination of mCRL2 and Viper.

Francalanza et al. [FRS11] propose a separation logic for message passing programs, where the assertion language has primitives for expressing the contents of communication channels. However, their approach circumvents the need to reason about different thread interleavings by targeting deterministic code, thereby sidestepping an important issue that we address: most problems in realistic distributed programs are the result of intricate thread interleavings. Lei et al. [LZ14] propose a separation logic for modular verification of message passing programs. They achieve modularity via assume-guarantee reasoning, but thereby require users of the logic to manually specify thread interference, which is often non-trivial and non-intuitive. Villard et al. [VLC09] propose a similar approach also based on separation logic, but here the main focus is on transferring heap ownership

between threads, using message passing techniques.

Also the field of session types is related [HVK98, GVR03, HMM+12, HHN+14, CDCPY15], which are a well-studied type discipline for describing protocols of message passing interaction between processes over communication channels (i.e., sessions). Likewise to our approach, these protocols are specifications of the communication behaviour of processes, and are usually expressed using process algebra, most often (variants of) the π-calculus. However, our approach has a slightly different aim: it uses process algebra not only to structure the communication behaviour, but also to reason about it, and to combine this reasoning with well-known deductive techniques for concurrency verification (viz. concurrent separation logic) in a sound and practical manner.

This chapter builds upon our earlier work [OBG+17, OBH16, ZS15] (presented in Chapters 4 and 5), in which process-algebraic abstractions are used to describe how the heap evolves over time in shared-memory concurrent programs (befitting the notion of protocols given earlier). However, in this chapter the abstractions have a different purpose: they instead capture message passing behaviour in a distributed setting. Our abstraction language is therefore different, for example by supporting summation and primitives for communication. Furthermore, in contrast to earlier work, this approach allows to use the result of process-algebraic analysis at intermediate points in the proof derivation of a program, via the novel **query** program annotation.

7.7 Conclusion

This chapter demonstrates how a combination of complementary verification techniques can be used to reason effectively about distributed applications, by naturally combining data-oriented reasoning via deductive verification, and temporal reasoning using algorithmic techniques. The approach is illustrated on a small, but intricate example. Our technique uses CSL to reason about data-centric properties of message passing programs, which are allowed to have shared state, and combines this with standard process-algebraic reasoning to verify properties of inter-thread communication. This combination of approaches is proven sound using the Coq theorem prover, and can easily be extended, e.g., to handle asynchronous- and collective communication (like barriers and broadcasts), as well as message loss and duplication.

7.7.1 Future Work

As future work, we plan to extend process-algebraic reasoning to deal with a reasonable subset of MPI [Mes], with the goal to develop a comprehensive verifi-

cation framework that targets real-world programming languages. This includes handling replicated processes [PPE04]. Moreover, we are actively collaborating with the mCRL2 developers to improve support for symbolic reasoning about process-algebraic models. We also plan to investigate the possibility to automatically extract models from program code. Finally, we will apply our approach on bigger, industrial case studies.

CHAPTER 8

Conclusions and Perspectives

Ensuring the behavioural correctness of parallel, concurrent and distributed software is notoriously difficult. This is because the number of possible system behaviours of such software is typically exponential in the number of concurrent sub-systems (e.g., threads, or distributed agents). It is therefore essential that developers of such complex systems are aided by formal methods, supported by mechanical tools, that are able to manage all these possible concurrent behaviours.

This thesis contributes towards such formal methods and tools, and in particular towards deductive verification. Deductive verifiers consider a behavioural specification of the software system, written in a program logic, and aim to prove that the code implementation adheres to this specification. The work in this thesis builds on a particular program logic that is specific for reasoning about concurrent heap-manipulating programs, named Concurrent Separation Logic (CSL).

In particular, this thesis addresses the challenge of verifying *global behavioural properties of concurrent software, in a reliable and practical manner*, which is, despite tremendous progress in recent years on both the theory and practice of CSL-based concurrency verification, still an open and active challenge.

Part I of this thesis contributes the first machine-checked CSL proof of a real-world parallel graph algorithm, called parallel nested depth-first search (NDFS); an algorithm used for multi-core LTL model checking, for example by LTSmin. Parallel NDFS is a parallel algorithm for finding accepting cycles in automata: cycles that contain at least one state that is marked as being "accepting". We verified soundness (only true accepting cycles are reported) and completeness (the algorithm returns an accepting cycle if one exists) of parallel NDFS, using the VerCors concurrency verifier, for *any* input automaton and with *any* number of threads. This verification opens-up new possibilities for the mechanical verification of parallel graph algorithms: the same approach may be reused to verify other algorithms,

like partial-order reduction or algorithms for parallel SCC decomposition.

The global behavioural properties of parallel NDFS could all be specified as global invariants. However, in many practical scenarios the properties-of-interest take the form of transition systems (locking protocols for example, or the message flow through distributed agents), which are difficult to specify in terms of global, first-order logic invariants. Part II of this thesis contributes an abstraction approach that allows specifying such properties more easily, in the context of shared-memory concurrent programs. This approach allows specifying concurrent program behaviour globally, as a process-algebraic model, and to thread-modularly prove that the code implementation adheres to this global behavioural specification. This shared-memory abstraction approach is (i) formalised and proven sound using the Coq proof assistant, (ii) implemented as part of VerCors, and is (iii) demonstrated on various verification examples, including a leader election protocol, as well as a bigger case study from industry, concerning the formal verification of a safety-critical traffic tunnel control system that is currently in use in Dutch traffic.

Process algebra have extensively been used over the last decades for the specification and verification of communicating systems. Part III of this thesis therefore investigates how our abstraction approach can be adapted to specify and verify distributed software, instead of shared-memory concurrent software. Distributed systems typically deal with computation (data) as well as communication (control-flow). We contribute a verification technique that combines deductive verification (of data properties) with model checking (of control-flow properties), which are, by nature, complementary verification techniques. More specifically, the communication behaviours of distributed threads are specified as a process algebra, while their computational behaviours are specified using CSL. Our approach allows combining the results of deductive verification and model checking, and thereby makes a step in bridging the program logic and model checking communities. The approach has been formalised and proven sound using the Coq proof assistant, and has been demonstrated on a small, yet intricate, verification example.

8.1 Contributions

Altogether, this thesis makes a major step forward towards the *practical and reliable* verification of global behavioural properties of real-world concurrent and distributed software. The techniques proposed in this thesis are: *reliable*, by having mechanically proven correctness results in Coq; are *expressive*, as they are modular, compositional, and build on mathematically elegant structures; and are *practical*, by being implemented in automated concurrency verifiers.

The exact contributions are listed below, explicitly, and are connected to the three

research questions given earlier, in the introduction.

8.1.1 Automated Verification of Parallel NDFS

Chapter 3 of this thesis answers ***RQ 1***:

> *How can concurrent separation logic be used to specify and mechanically verify parallel graph-based algorithms for model checking?*

This question is answered in terms of the following contributions:

- We investigated the possibility to mechanically prove soundness and completeness of the parallel Nested Depth-First Search (NDFS) graph algorithm. While doing this, we reformulated the original correctness proof of this algorithm, to a format that can be encoded and proven with CSL.
- We encoded the graph algorithm and its specification in VerCors—a concurrency verifier that uses CSL as its logical foundation—and mechanically proved correctness of the algorithm. To the best of our knowledge, this is the first mechanically verified parallel graph algorithm.
- We used this mechanised correctness proof to easily prove various optimisations of PNDFS, namely: the all-red and early cycle detection extensions.

8.1.2 Abstractions for Shared-Memory Concurrency

Chapters 4–6 of this thesis answer ***RQ 2***:

> *How can global behavioural correctness properties of shared-memory concurrent programs be specified and mechanically verified, by means of abstraction?*

This question is answered in terms of the following contributions:

- We developed an abstraction technique for specifying complex behaviours of concurrent programs, at a global level (see Chapter 4). This technique builds on CSL, and uses its separating conjunction to allow to thread-modularly prove that a program refines its process-algebraic specification.
- We have proven soundness of this abstraction approach, and presented the formalisation in Chapter 5. This soundness argument has been fully mechanised using the Coq proof assistant, to increase the confidence of its correctness.

- We have illustrated the practical applicability of our approach on various verification examples and case studies. Chapter 4 demonstrates it, among other examples, on (i) the classical Owicki–Gries example, (ii) a concurrent locking protocol, (iii) a program that calculates the GCD of two positive integers concurrently, and (iv) on a classical leader election protocol that is implemented on a shared-memory system of message passing. Moreover, Chapter 6 demonstrates the approach on an industrial, safety-critical case study, concerning the formal verification of a traffic tunnel control system.

8.1.3 Abstractions for Message-Passing Concurrency

Chapter 7 of this thesis answers ***RQ 3***:

> *How can the strengths of deductive and algorithmic verification soundly be combined, to specify and mechanically verify global behavioural properties of distributed message-passing software?*

This question is answered in terms of the following contributions:

- We developed an abstraction technique for specifying message passing behaviour of distributed programs with send/recv primitives (see Chapter 7). This abstraction technique extends CSL with logical constructs that allow to deductively prove that a message passing program soundly refines its abstract model. The models are specified as process algebra—a widely-used language for specifying distributed program behaviour.
- We combined this abstraction approach with model checking, by allowing to use a model checker, mCRL2 in this case, to analyse the models, to indirectly reason globally about the communication behaviour of the modelled program. The deductively established refinement relation between the program and its model allows the results of this reasoning to be used in the deductive proof of the program (see Chapter 7 for details).
- We have mechanically proven soundness of our abstraction technique, by encoding all proof steps in the Coq proof assistant. See Chapter 7 for the formalisation. We have, moreover, demonstrated the approach on a simple but intricate verification example.

8.2 Discussion and Future Directions

We see several extensions and improvements of the current work for future research.

8.2.1 Verifying Parallel Graph Algorithms

This thesis contributes the first automated deductive verification of a parallel model checking algorithm. The verification has been done using VerCors, by encoding the algorithm in PVL, which is a Java-like object-oriented toy language designed by the developers of VerCors. It would be interesting to investigate whether the verification can be brought closer to an actual real-world programming language, for example by verifying the efficient C implementation of parallel NDFS that LTSmin uses. Of course, this would require the automated concurrency verifier to be able to deal with all the intricacies of the semantics of (concurrent) C, e.g., weak/relaxed memory models, which is relatively unexplored at the time of writing. Exploring these would also be an interesting direction of research.

One could also explore the verification of other parallel algorithms for graph exploration or model checking. An interesting candidate may be the multi-core, on-the-fly SCC algorithm of Bloemen et al. [BLP16, BBDL+18, Blo19], based on the classical sequential SCC algorithms of Tarjan [Tar71] and Munro [Mun71]. Another candidate could be the extension of parallel NDFS with subsumption for timed automata [LOD+13]. Compared to the other suggestions of future work in this section, this particular direction of research could be picked up relatively easily by interested researchers, since it may benefit from the approach discussed in Chapter 3 as well as reuse several of our verification components.

It may also be interesting to extend our parallel NDFS verification to the verification of a full-fledged (executable) parallel LTL model checker. Even though several *sequential* model checkers have been verified, no full verification of a parallel model checker currently exists. One possible way to realise this, is to do the actual code verification with an automated verifier like VerCors or VeriFast, and prove the underlying meta-theory in an interactive proof assistant like Coq or Isabelle.

Finally, in this direction of research, it would also be worthwhile to construct a verification stack/library that contains common subtasks for verifying parallel graph algorithms. Such a library could for example contain generic methods for proving termination, for thread coordination and cooperation, for graph manipulations and representations, and for standard search methods over graphs, like BFS and DFS. Most of these components are currently implemented using auxiliary ghost state, as discussed in Chapter 3. To construct a more general verification framework, one could for example automatically generate the required ghost state. Wang et al. [WCMH19] already propose general techniques in Coq for verifying sequential graph-manipulating programs in a separation logic setting. Perhaps these techniques can be lifted to a concurrent setting, and be applied in automated concurrency verifiers like VeriFast and VerCors.

8.2.2 Program Abstractions

Parts II and III of this thesis contribute abstraction techniques for specifying and verifying shared-memory and message passing concurrent programs. This line of work can be extended in various ways.

Firstly, the process-algebraic models currently have to be defined manually. It would be a valuable contribution to this work, to be able to automatically extract program models from the code, or at least partially. In order to achieve this, it might be worthwhile to investigate the related line of work on Session Types. Castro et al. [CHJ+19], for example, are able to automatically extract process-algebraic communication protocol descriptions (i.e., sessions) out of distributed Go programs. Session Types, on the other hand, currently do not allow to analyse such protocol descriptions, to conclude data-oriented properties over the program; something that our work *is* able to do. The combination of our approach with Session Types would therefore be an interesting combination to further explore.

Secondly, the presented abstraction approach only supports safety properties at the moment. This is because the current approach uses process algebra as a specification language to give an over-approximation of the program behaviour. In order to preserve liveness properties, one would have to establish that every trace in the process-algebraic model has a corresponding behaviour in the modelled program. This involves proving termination on the program level, which in combination with concurrency is an interesting, but also a very challenging, field of research [RPDYGS16].

Thirdly, the main contribution of our abstraction approach is its ability to construct a refinement relation between a program and its model. These refinement relations are proven thread-modularly, in a deductive manner, by using reasoning techniques that extend CSL. Even though this approach allows analysing the models algorithmically, e.g., using a model checker, such analysis is currently not done in a modular way. mCRL2 would, for example, still need to linearise these models in order to be able to analyse them. Even though model checkers generally come with state space minimisation techniques to combat state space explosions (e.g., confluence reduction, bisimulation minimisation, etc.), it would still be interesting to investigate how our work combines with work on compositional/modular model checking [CLM89, KV98].

Fourthly, one may argue that our approach is a fairly "heavyweight" approach to software verification, by requiring heavy machinery, like CSL, model checking and process algebra, for specifying and analysing program behaviour. It would be very interesting to investigate how such heavyweight verification approaches could blend more naturally with lightweight approaches, like for example unit testing and integration testing, which are used structurally, every day, by software engineers

in industry. One example of research in this direction, is the generation of test cases from JML specifications, as is done by JMLUnitNG [ZN11]. Following these ideas, the work in this thesis could potentially be combined with model-based testing, by using JTorX [Bel10] for example, to derive test cases from process-algebraic models (JTorX already has an mCRL2 front-end). This would allow one to use deductive verification wherever needed (say, to verify safety-critical software components), and derive test cases for the less safety-critical parts of the system, in a more lightweight manner. Compared to the other suggestions of future work in this section, this direction of research could be picked up relatively easily.

Yet another possible direction of future research would be to investigate alternative modelling formalisms for the program abstractions. Penninckx et al., for example, use petri-nets to model the input/output behaviour of *sequential* programs [PJP15], in the context of VeriFast (the theory of this approach has later been lifted to the concurrent setting [PTJ19], via an encoding into the Iris framework). Our abstraction approach may potentially extend well to petri-nets (e.g., because the mCRL2 language allows, in theory, a faithful translation of petri nets [GMR+07]).

Finally, it would be interesting to investigate whether our refinement technique can be used to derive/generate source code from program models [HJ18]. Such an extension would bring our verification approach closer to model-based design; a software development discipline in which one starts with an abstract model of the software system, that is slowly refined or synthesised into more concrete models, until eventually the actual code implementation is derived. Our verification technique might fit well into such a discipline, which may make the overall approach more lightweight, as well as more accessible to actual industrial software developers.

8.2.3 Recommendations for Software Engineers

Let us finish by giving recommendations for software engineers/operators for applying or incorporating the proposed techniques into their operational processes.

In practice it often happens that software is intended to follow certain patterns, protocols or state machines, like in the traffic tunnel control system discussed in Chapter 6. Such protocols or state machines are typically specified informally during the software design phase, and are then hard-coded in the programs' source code during the software development phase. This raises two questions, which are particularly important in the case of safety-critical software: (i) Is the underlying protocol consistent? And (ii) does the software correctly implement the intended protocol? The techniques proposed in this thesis allows one to address both.

More specifically, the proposed techniques allow to make formal specifications of protocols, for example by using process algebra, and allow to relate these to the programs' source code. For example, in Chapter 6 we formalise the state machine description of a traffic tunnel control system in mCRL2, and relate this formal mCRL2 model to the source code using the techniques of Chapters 2, 4 and 5. This helped to identify intricate, problematic behaviour that would otherwise be very hard to identify, due to the high complexity of the underlying protocol.

However, at the moment the proposed techniques cannot be applied without expertise or prior knowledge in formal verification, process algebra and program logics. Follow-up research is needed to make automated deductive verification accessible to software engineers without such prior knowledge, for example in generating program annotations, inference of contracts (pre/post-conditions), and automatically deriving formal specifications out of documents of informal requirements (resulting from the requirement engineering phase). Nevertheless, in projects concerning safety-critical software, it may be worthwhile to hire an engineer in formal software verification, as formal techniques can help to precisely specify the software systems' behaviour, and to handle the large complexity of these specifications.

8.3 Outlook

Roughly twenty years ago, software verification was a pen-and-paper activity [Fil11], that was only feasible for very simple, sequential programs. In contrast, nowadays we have formal tools and techniques that are mature enough to be able to verify complex concurrent programs and algorithms, as is demonstrated by this thesis.

Even though deductive verification is not yet a standard, integral part of industrial software development, by still requiring expert knowledge in formal methods, such techniques do come closer to real-world programming languages and are able to reason about increasingly complex language features, like concurrency.

There are many promising opportunities to bring deductive verification techniques closer to the industrial routine of software development, and thereby to increasing the reliability of future software systems that we will use every day, either directly or indirectly, sometimes consciously and more often unconsciously.

Part IV

Appendices

APPENDIX **A**

Auxiliary Definitions for Chapter 5

A.1 Processes

This section provides auxiliary definitions and extra information regarding process-algebraic models, accompanying Section 5.3 of Chapter 5.

A.1.1 Syntax

Definition A.1.1 (Free variables in process expressions)**.** *The set of free variables in a process expression is defined in terms of a function* $\mathsf{fv}_e(\cdot) : \mathit{ProcExpr} \to 2^{\mathit{ProcVar}}$*, in the following way:*

$$\begin{aligned}
\mathsf{fv}_e(m) &\triangleq \emptyset \\
\mathsf{fv}_e(x) &\triangleq \{x\} \\
\mathsf{fv}_e(e_1 + e_2) &\triangleq \mathsf{fv}_e(e_1) \cup \mathsf{fv}_e(e_2) \\
\mathsf{fv}_e(e_1 - e_2) &\triangleq \mathsf{fv}_e(e_1) \cup \mathsf{fv}_e(e_2)
\end{aligned}$$

Definition A.1.2 (Free variables in process conditions)**.** *The set of free variables in a process condition is defined in terms of a function* $\mathsf{fv}_b(\cdot) : \mathit{ProcCond} \to 2^{\mathit{ProcVar}}$*, in the following way:*

$$\begin{aligned}
\mathsf{fv}_b(\mathsf{true}) &\triangleq \emptyset \\
\mathsf{fv}_b(\mathsf{false}) &\triangleq \emptyset \\
\mathsf{fv}_b(\neg b) &\triangleq \mathsf{fv}_b(b) \\
\mathsf{fv}_b(b_1 \wedge b_2) &\triangleq \mathsf{fv}_b(b_1) \cup \mathsf{fv}_b(b_2) \\
\mathsf{fv}_b(e_1 = e_2) &\triangleq \mathsf{fv}_e(e_1) \cup \mathsf{fv}_e(e_2)
\end{aligned}$$

$$\mathsf{fv}_b(e_1 < e_2) \triangleq \mathsf{fv}_e(e_1) \cup \mathsf{fv}_e(e_2)$$

Definition A.1.3 (Free variables in processes)**.** *The set of free variables in a process is defined in terms of the function* $\mathsf{fv}_p(\cdot) : \mathit{Proc} \to 2^{\mathit{ProcVar}}$*, in the following way:*

$$\begin{aligned}
\mathsf{fv}_p(\varepsilon) &\triangleq \emptyset \\
\mathsf{fv}_p(\delta) &\triangleq \emptyset \\
\mathsf{fv}_p(a) &\triangleq \mathsf{fv}_b(\mathsf{pre}(a)) \cup \mathsf{fv}_b(\mathsf{post}(a)) \\
\mathsf{fv}_p(P \cdot Q) &\triangleq \mathsf{fv}_p(P) \cup \mathsf{fv}_p(Q) \\
\mathsf{fv}_p(P + Q) &\triangleq \mathsf{fv}_p(P) \cup \mathsf{fv}_p(Q) \\
\mathsf{fv}_p(P \parallel Q) &\triangleq \mathsf{fv}_p(P) \cup \mathsf{fv}_p(Q) \\
\mathsf{fv}_p(P \between Q) &\triangleq \mathsf{fv}_p(P) \cup \mathsf{fv}_p(Q) \\
\mathsf{fv}_p(P^*) &\triangleq \mathsf{fv}_p(P)
\end{aligned}$$

A.1.2 Semantics

Let $[\![+]\!], [\![-]\!], [\![=]\!], [\![<]\!], \ldots$ be the meaning of the operators $+, -, =, <, \ldots$ that are used in the expression language of processes, in the domain of values (Val).

Definition A.1.4 (Denotational semantics of process expressions)**.** *The evaluation function* $[\![\cdot]\!]_e \cdot : \mathit{ProcExpr} \to \mathit{ProcStore} \to \mathit{Val}$ *defines the denotational semantics of expressions in the following way.*

$$\begin{aligned}
[\![m]\!]_e\sigma &\triangleq m \\
[\![x]\!]_e\sigma &\triangleq \sigma(x) \\
[\![e_1 + e_2]\!]_e\sigma &\triangleq [\![+]\!]\big([\![e_1]\!]_e\sigma, [\![e_2]\!]_e\sigma\big) \\
[\![e_1 - e_2]\!]_e\sigma &\triangleq [\![-]\!]\big([\![e_1]\!]_e\sigma, [\![e_2]\!]_e\sigma\big)
\end{aligned}$$

Definition A.1.5 (Denotational semantics of process conditions)**.** *The evaluation function* $[\![\cdot]\!]_b \cdot : \mathit{ProcCond} \to \mathit{ProcStore} \to \mathit{Prop}$ *defines the denotational semantics of conditions in the following way.*

$$\begin{aligned}
[\![\mathsf{true}]\!]_b\sigma &\triangleq \mathsf{true} \\
[\![\mathsf{false}]\!]_b\sigma &\triangleq \mathsf{false} \\
[\![\neg b]\!]_b\sigma &\triangleq \neg[\![b]\!]_b\sigma
\end{aligned}$$

$$\begin{aligned}
[\![b_1 \wedge b_2]\!]_{\mathsf{b}}\sigma &\triangleq [\![b_1]\!]_{\mathsf{b}}\sigma \wedge [\![b_2]\!]_{\mathsf{b}}\sigma \\
[\![e_1 = e_2]\!]_{\mathsf{b}}\sigma &\triangleq [\![=]\!]\big([\![e_1]\!]_{\mathsf{e}}\sigma, [\![e_2]\!]_{\mathsf{b}}\sigma\big) \\
[\![e_1 < e_2]\!]_{\mathsf{b}}\sigma &\triangleq [\![<]\!]\big([\![e_1]\!]_{\mathsf{e}}\sigma, [\![e_2]\!]_{\mathsf{b}}\sigma\big)
\end{aligned}$$

We often simply use the notations $[\![e]\!]\sigma$ and $[\![b]\!]\sigma$ instead of $[\![e]\!]_{\mathsf{e}}\sigma$ or $[\![b]\!]_{\mathsf{b}}\sigma$, respectively. Moreover, the notation $[\![e_0, \ldots, e_n]\!]\sigma$ abbreviates $[\![e_0]\!]\sigma, \ldots, [\![e_n]\!]\sigma$.

A.2 Programs

A.2.1 Syntax of Programs

A.2.1.1 Free Variables

Definition A.2.1 (Free variables of expressions)**.** *The set of free variables in (program) expressions is defined in terms of the function* $\mathsf{fv}_{\mathsf{E}}(\cdot) : \mathit{Expr} \to 2^{\mathit{Var}}$ *as follows.*

$$\begin{aligned}
\mathsf{fv}_{\mathsf{E}}(n) &\triangleq \emptyset \\
\mathsf{fv}_{\mathsf{E}}(X) &\triangleq \{X\} \\
\mathsf{fv}_{\mathsf{E}}(E_1 + E_2) &\triangleq \mathsf{fv}_{\mathsf{E}}(E_1) \cup \mathsf{fv}_{\mathsf{E}}(E_2) \\
\mathsf{fv}_{\mathsf{E}}(E_1 - E_2) &\triangleq \mathsf{fv}_{\mathsf{E}}(E_1) \cup \mathsf{fv}_{\mathsf{E}}(E_2)
\end{aligned}$$

Definition A.2.2 (Free variables in conditions)**.** *The set of free variables in (program) conditions is defined in terms of the function* $\mathsf{fv}_{\mathsf{B}}(\cdot) : \mathit{Cond} \to 2^{\mathit{Var}}$ *as follows.*

$$\begin{aligned}
\mathsf{fv}_{\mathsf{B}}(\mathsf{true}) &\triangleq \emptyset \\
\mathsf{fv}_{\mathsf{B}}(\mathsf{false}) &\triangleq \emptyset \\
\mathsf{fv}_{\mathsf{B}}(\neg B) &\triangleq \mathsf{fv}_{\mathsf{B}}(B) \\
\mathsf{fv}_{\mathsf{B}}(B_1 \wedge B_2) &\triangleq \mathsf{fv}_{\mathsf{B}}(B_1) \cup \mathsf{fv}_{\mathsf{B}}(B_2) \\
\mathsf{fv}_{\mathsf{B}}(E_1 = E_2) &\triangleq \mathsf{fv}_{\mathsf{E}}(E_1) \cup \mathsf{fv}_{\mathsf{E}}(E_2) \\
\mathsf{fv}_{\mathsf{B}}(E_1 < E_2) &\triangleq \mathsf{fv}_{\mathsf{E}}(E_1) \cup \mathsf{fv}_{\mathsf{E}}(E_2)
\end{aligned}$$

We often use the notations $\mathsf{fv}(E)$ and $\mathsf{fv}(B)$ instead of $\mathsf{fv}_{\mathsf{E}}(E)$ or $\mathsf{fv}_{\mathsf{B}}(B)$, respectively, thereby overloading the notation $\mathsf{fv}(\cdot)$. Moreover, the set of free variables $\mathsf{fv}_{\mathsf{AB}}(\Pi)$ in abstraction binders Π is defined in the following way.

Definition A.2.3 (Free variables in abstraction binders)**.**

$$\mathsf{fv}_{AB}(\Pi) \triangleq \bigcup_{i=0}^{n} \mathsf{fv}_E(E_i) \quad \textit{for} \quad \Pi = \{x_0 \mapsto E_0, \cdots, x_n \mapsto E_n\}$$

Definition A.2.4 (Free variables in commands)**.** *The set of* free variables *of a command is defined in terms of the function* $\mathsf{fv}_C(\cdot) : \mathit{Cmd} \to 2^{\mathit{Var}}$*, as follows.*

$$\begin{aligned}
\mathsf{fv}_C(\textbf{skip}) &\triangleq \emptyset \\
\mathsf{fv}_C(X := E) &\triangleq \{X\} \cup \mathsf{fv}_E(E) \\
\mathsf{fv}_C(X := [E]) &\triangleq \{X\} \cup \mathsf{fv}_E(E) \\
\mathsf{fv}_C([E_1] := E_2) &\triangleq \mathsf{fv}_E(E_1) \cup \mathsf{fv}_E(E_2) \\
\mathsf{fv}_C(C_1; C_2) &\triangleq \mathsf{fv}_C(C_1) \cup \mathsf{fv}_C(C_2) \\
\mathsf{fv}_C(X := \textbf{alloc } E) &\triangleq \{X\} \cup \mathsf{fv}_E(E) \\
\mathsf{fv}_C(\textbf{dispose } E) &\triangleq \mathsf{fv}_E(E) \\
\mathsf{fv}_C(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2) &\triangleq \mathsf{fv}_B(B) \cup \mathsf{fv}_C(C_1) \cup \mathsf{fv}_C(C_2) \\
\mathsf{fv}_C(\textbf{while } B \textbf{ do } C) &\triangleq \mathsf{fv}_B(B) \cup \mathsf{fv}_C(C) \\
\mathsf{fv}_C(C_1 \parallel C_2) &\triangleq \mathsf{fv}_C(C_1) \cup \mathsf{fv}_C(C_2) \\
\mathsf{fv}_C(\textbf{atomic } C) &\triangleq \mathsf{fv}_C(C) \\
\mathsf{fv}_C(\textbf{inatom } C) &\triangleq \mathsf{fv}_C(C) \\
\mathsf{fv}_C(X := \textbf{process } p \textbf{ over } \Pi) &\triangleq \{X\} \cup \mathsf{fv}_{AB}(\Pi) \\
\mathsf{fv}_C(\textbf{action } X.a \textbf{ do } C) &\triangleq \{X\} \cup \mathsf{fv}_C(C) \\
\mathsf{fv}_C(\textbf{inact } C) &\triangleq \mathsf{fv}_C(C) \\
\mathsf{fv}_C(\textbf{finish } X) &\triangleq \{X\}
\end{aligned}$$

Definition A.2.5 (Modified variables in commands)**.** *The set of* modified variables *of a command is defined in terms of the function* $\mathsf{mod}(\cdot) : \mathit{Cmd} \to 2^{\mathit{Var}}$*, in the following way.*

$$\begin{aligned}
\mathsf{mod}(\textbf{skip}) &\triangleq \emptyset \\
\mathsf{mod}(X := E) &\triangleq \{X\} \\
\mathsf{mod}(X := [E]) &\triangleq \{X\} \\
\mathsf{mod}([E_1] := E_2) &\triangleq \emptyset \\
\mathsf{mod}(C_1; C_2) &\triangleq \mathsf{mod}(C_1) \cup \mathsf{mod}(C_2)
\end{aligned}$$

$$
\begin{aligned}
\mathsf{mod}(X := \textbf{alloc}\ E) &\triangleq \{X\} \\
\mathsf{mod}(\textbf{dispose}\ E) &\triangleq \emptyset \\
\mathsf{mod}(\textbf{if}\ B\ \textbf{then}\ C1\ \textbf{else}\ C_2) &\triangleq \mathsf{mod}(C_1) \cup \mathsf{mod}(C_2) \\
\mathsf{mod}(\textbf{while}\ B\ \textbf{do}\ C) &\triangleq \mathsf{mod}(C) \\
\mathsf{mod}(C_1 \parallel C_2) &\triangleq \mathsf{mod}(C_1) \cup \mathsf{mod}(C_2) \\
\mathsf{mod}(\textbf{atomic}\ C) &\triangleq \mathsf{mod}(C) \\
\mathsf{mod}(\textbf{inatom}\ C) &\triangleq \mathsf{mod}(C) \\
\mathsf{mod}(X := \textbf{process}\ p\ \textbf{over}\ \Pi) &\triangleq \{X\} \\
\mathsf{mod}(\textbf{action}\ X.a\ \textbf{do}\ C) &\triangleq \mathsf{mod}(C) \\
\mathsf{mod}(\textbf{inact}\ C) &\triangleq \mathsf{mod}(C) \\
\mathsf{mod}(\textbf{finish}\ X) &\triangleq \emptyset
\end{aligned}
$$

Naturally, the set of modified variables in any command C is a subset of the set of free variables in C:

Lemma A.2.1. *For every $C \in Cmd$ it holds that* $\mathsf{mod}(C) \subseteq \mathsf{fv}_C(C)$.

A.2.1.2 Substitution

Definition A.2.6 (Substitution in expressions)**.** *Substitution $E[X/E']$ of a variable X with E' within E is defined as follows, by structural recursion on E:*

$$
\begin{aligned}
n[X/E] &\triangleq n \\
Y[X/E] &\triangleq \begin{cases} E & \text{if } X = Y \\ Y & \text{if } X \neq Y \end{cases} \\
(E_1 + E_2)[X/E] &\triangleq E_1[X/E] + E_2[X/E] \\
(E_1 - E_2)[X/E] &\triangleq E_1[X/E] - E_2[X/E]
\end{aligned}
$$

Definition A.2.7 (Substitution in conditions)**.** *Substitution $B[X/E]$ of a variable X with E within B is defined as follows, by structural recursion on B:*

$$
\begin{aligned}
\mathsf{true}[X/E] &\triangleq \mathsf{true} \\
\mathsf{false}[X/E] &\triangleq \mathsf{false} \\
(\neg B)[X/E] &\triangleq \neg(B[X/E])
\end{aligned}
$$

$$(B_1 \wedge B_2)[X/E] \triangleq B_1[X/E] \wedge B_2[X/E]$$
$$(E_1 = E_2)[X/E] \triangleq E_1[X/E] = E_2[X/E]$$
$$(E_1 < E_2)[X/E] \triangleq E_1[X/E] < E_2[X/E]$$

Definition A.2.8 (Substitution in abstraction binders)**.**

$$\Pi[X/E] \triangleq \{x_0 \mapsto E_0[X/E], \cdots, x_n \mapsto E_n[X/E]\}$$
$$\textit{for}\ \ \Pi = \{x_0 \mapsto E_0, \cdots, x_n \mapsto E_n\}$$

Lemma A.2.2. *The following basic properties hold for substitution:*

1. *For any X, E and E', if $X \notin \mathsf{fv}_E(E)$, then $E[X/E'] = E$.*
2. *For any X, E and B, if $X \notin \mathsf{fv}_B(B)$, then $B[X/E] = B$.*
3. *For any X, E and Π, if $X \notin \mathsf{fv}_{AB}(\Pi)$, then $\Pi[X/E] = \Pi$.*

A.2.1.3 User Programs and Well-Formed Programs

Any command C is a *user command* if C does not contain subprograms of the form **inatom** C' or **inact** C' (for any C'). Recall that these two commands are *runtime syntax*, and only exist for runtime purposes.

Definition A.2.9 (User command)**.** *Any command C is defined to be a* user command*, if C : user, which is inductively defined as follows.*

$$\textbf{skip} : \mathsf{user} \qquad X := E : \mathsf{user} \qquad X := [E] : \mathsf{user} \qquad [E_1] := E_2 : \mathsf{user}$$

$$\frac{C_1 : \mathsf{user} \qquad C_2 : \mathsf{user}}{C_1; C_2 : \mathsf{user}} \qquad X := \textbf{alloc}\ E : \mathsf{user} \qquad \textbf{dispose}\ E : \mathsf{user}$$

$$\frac{C_1 : \mathsf{user} \qquad C_2 : \mathsf{user}}{\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2 : \mathsf{user}} \qquad \frac{C : \mathsf{user}}{\textbf{while}\ B\ \textbf{do}\ C : \mathsf{user}} \qquad \frac{C_1 : \mathsf{user} \qquad C_2 : \mathsf{user}}{C_1 \parallel C_2 : \mathsf{user}}$$

$$\frac{C : \mathsf{user}}{\textbf{atomic}\ C : \mathsf{user}} \qquad X := \textbf{process}\ p\ \textbf{over}\ \Pi : \mathsf{user} \qquad \frac{C : \mathsf{user}}{\textbf{action}\ X.a\ \textbf{do}\ C : \mathsf{user}}$$

$$\textbf{finish}\ X : \mathsf{user}$$

Any command C is defined to be *basic*, if C does not contain atomic subprograms, or specification-only constructs. This notion of basicality is captured by the following definition.

Definition A.2.10 (Basic command)**.** *Any command C is defined to be a* basic command*, if C : basic, which is inductively defined as follows.*

$$\textbf{skip} : \mathsf{basic} \qquad X := E : \mathsf{basic} \qquad X := [E] : \mathsf{basic} \qquad [E_1] := E_2 : \mathsf{basic}$$

$$\frac{C_1 : \mathsf{basic} \qquad C_2 : \mathsf{basic}}{C_1; C_2 : \mathsf{basic}} \qquad X := \textbf{alloc}\ E : \mathsf{basic} \qquad \textbf{dispose}\ E : \mathsf{basic}$$

$$\frac{C_1 : \mathsf{basic} \qquad C_2 : \mathsf{basic}}{\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2 : \mathsf{basic}} \qquad \frac{C : \mathsf{basic}}{\textbf{while}\ B\ \textbf{do}\ C : \mathsf{basic}}$$

$$\frac{C_1 : \mathsf{basic} \qquad C_2 : \mathsf{basic}}{C_1 \parallel C_2 : \mathsf{basic}}$$

Any command C is defined to be *well-formed*, if all C's action sub-programs (i.e., the C''s in subprograms of the form $\textbf{action}\ X.a\ \textbf{do}\ C'$ and $\textbf{inact}\ C'$ inside C) are basic.

Definition A.2.11 (Well-formed command)**.** *Any command C is defined to be a* well-formed command*, if C : wf, which is inductively defined as follows.*

$$\textbf{skip} : \mathsf{wf} \qquad X := E : \mathsf{wf} \qquad X := [E] : \mathsf{wf} \qquad [E_1] := E_2 : \mathsf{wf}$$

$$\frac{C_1 : \mathsf{wf} \qquad C_2 : \mathsf{wf}}{C_1; C_2 : \mathsf{wf}} \qquad X := \textbf{alloc}\ E : \mathsf{wf} \qquad \textbf{dispose}\ E : \mathsf{wf}$$

$$\frac{C_1 : \mathsf{wf} \qquad C_2 : \mathsf{wf}}{\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2 : \mathsf{wf}} \qquad \frac{C : \mathsf{wf}}{\textbf{while}\ B\ \textbf{do}\ C : \mathsf{wf}} \qquad \frac{C_1 : \mathsf{wf} \qquad C_2 : \mathsf{wf}}{C_1 \parallel C_2 : \mathsf{wf}}$$

$$\frac{C : \mathsf{wf}}{\textbf{atomic}\ C : \mathsf{wf}} \qquad \frac{C : \mathsf{wf}}{\textbf{inatom}\ C : \mathsf{wf}} \qquad X := \textbf{process}\ p\ \textbf{over}\ \Pi : \mathsf{wf}$$

$$\frac{C : \mathsf{basic}}{\textbf{action}\ X.a\ \textbf{do}\ C : \mathsf{wf}} \qquad \frac{C : \mathsf{basic}}{\textbf{inact}\ C : \mathsf{wf}} \qquad \textbf{finish}\ X : \mathsf{wf}$$

From the above definitions it follows that the set of basic programs is a subset of the set of all well-formed programs:

Lemma A.2.3. *For any command C, if C : basic, then C : wf.*

A.2.2 Semantics of Programs

A.2.2.1 Denotational Semantics

Definition A.2.12 (Denotational semantics of expressions)**.** *The denotational semantics of expressions is defined as the function* $[\![\cdot]\!]_{\mathsf{E}}\cdot : \mathit{Expr} \to \mathit{Store} \to \mathit{Val}$*, as follows.*

$$\begin{aligned}
[\![n]\!]_{\mathsf{E}}s &\triangleq n \\
[\![x]\!]_{\mathsf{E}}s &\triangleq s(x) \\
[\![E_1 + E_2]\!]_{\mathsf{E}}s &\triangleq [\![+]\!]\big([\![E_1]\!]_{\mathsf{E}}s, [\![E_2]\!]_{\mathsf{E}}s\big) \\
[\![E_1 - E_2]\!]_{\mathsf{E}}s &\triangleq [\![-]\!]\big([\![E_1]\!]_{\mathsf{E}}s, [\![E_2]\!]_{\mathsf{E}}s\big)
\end{aligned}$$

Definition A.2.13 (Denotational semantics of conditions)**.** *The denotational semantics of conditions is defined in terms of the function* $[\![\cdot]\!]_{\mathsf{B}}\cdot : \mathit{Cond} \to \mathit{Store} \to \mathit{Val}$*, as follows.*

$$\begin{aligned}
[\![\mathsf{true}]\!]_{\mathsf{B}}s &\triangleq \mathsf{true} \\
[\![\mathsf{false}]\!]_{\mathsf{B}}s &\triangleq \mathsf{false} \\
[\![\neg B]\!]_{\mathsf{B}}s &\triangleq \neg[\![B]\!]_{\mathsf{B}}s \\
[\![B_1 \wedge B_2]\!]_{\mathsf{B}}s &\triangleq [\![B_1]\!]_{\mathsf{B}}s \wedge [\![B_2]\!]_{\mathsf{B}}s \\
[\![E_1 = E_2]\!]_{\mathsf{B}}s &\triangleq [\![=]\!]\big([\![E_1]\!]_{\mathsf{E}}s, [\![E_2]\!]_{\mathsf{E}}s\big) \\
[\![E_1 < E_2]\!]_{\mathsf{B}}s &\triangleq [\![<]\!]\big([\![E_1]\!]_{\mathsf{E}}s, [\![E_2]\!]_{\mathsf{E}}s\big)
\end{aligned}$$

In this thesis we often use the notations $[\![E]\!]s$ and $[\![B]\!]s$ instead of $[\![E]\!]_{\mathsf{E}}s$ or $[\![B]\!]_{\mathsf{B}}s$, for denoting the evaluations of E and B, respectively.

The structural operational semantics of programs is fully defined on page 139, as Definition 5.3.5, and is therefore not repeated here.

A.3 Assertions

Definition A.3.1 (Free variables in assertions)**.** *The set of free variables in assertions is defined in terms of the function* $\mathsf{fv}_{\mathsf{A}}(\cdot) : \mathit{Assn} \to 2^{\mathit{Var}}$*, by structural recursion on the first argument, in the following way.*

$$\mathsf{fv}_{\mathsf{A}}(B) \triangleq \mathsf{fv}_{\mathsf{B}}(B)$$

$$
\begin{aligned}
\mathsf{fv}_{\mathsf{A}}(\forall X.\mathcal{P}) &\triangleq \mathsf{fv}_{\mathsf{A}}(\mathcal{P}) \setminus \{X\} \\
\mathsf{fv}_{\mathsf{A}}(\exists X.\mathcal{P}) &\triangleq \mathsf{fv}_{\mathsf{A}}(\mathcal{P}) \setminus \{X\} \\
\mathsf{fv}_{\mathsf{A}}(\mathcal{P} \vee \mathcal{Q}) &\triangleq \mathsf{fv}_{\mathsf{A}}(\mathcal{P}) \cup \mathsf{fv}_{\mathsf{A}}(\mathcal{Q}) \\
\mathsf{fv}_{\mathsf{A}}(\mathcal{P} * \mathcal{Q}) &\triangleq \mathsf{fv}_{\mathsf{A}}(\mathcal{P}) \cup \mathsf{fv}_{\mathsf{A}}(\mathcal{Q}) \\
\mathsf{fv}_{\mathsf{A}}(\mathcal{P} \mathrel{-\!\!*} \mathcal{Q}) &\triangleq \mathsf{fv}_{\mathsf{A}}(\mathcal{P}) \cup \mathsf{fv}_{\mathsf{A}}(\mathcal{Q}) \\
\mathsf{fv}_{\mathsf{A}}(*_{i \in I} \mathcal{P}_i) &\triangleq \bigcup_{i \in I} \mathsf{fv}_{\mathsf{A}}(\mathcal{P}_i) \\
\mathsf{fv}_{\mathsf{A}}(E_1 \overset{\pi}{\hookrightarrow}_t E_2) &\triangleq \mathsf{fv}_{\mathsf{E}}(E_1) \cup \mathsf{fv}_{\mathsf{E}}(E_2) \\
\mathsf{fv}_{\mathsf{A}}(\mathsf{Proc}_\pi(X, p, P, \Pi)) &\triangleq \{X\} \cup \mathsf{fv}_{\mathsf{AB}}(\Pi)
\end{aligned}
$$

Definition A.3.2 (Substitution in assertions)**.** *The substitution of a variable X for an expression E inside an assertion $\mathcal{P}$ is written $\mathcal{P}[X/E]$, and is defined by structural recursion on $\mathcal{P}$, as follows.*

$$
\begin{aligned}
(\forall Y.\mathcal{P})[X/E] &\triangleq \begin{cases} \forall Y.\mathcal{P} & \text{if } X = Y \\ \forall Y.(\mathcal{P}[X/E]) & \text{if } X \neq Y \end{cases} \\
(\exists Y.\mathcal{P})[X/E] &\triangleq \begin{cases} \exists Y.\mathcal{P} & \text{if } X = Y \\ \exists Y.(\mathcal{P}[X/E]) & \text{if } X \neq Y \end{cases} \\
(\mathcal{P} \vee \mathcal{Q})[X/E] &\triangleq \mathcal{P}[X/E] \vee \mathcal{Q}[X/E] \\
(\mathcal{P} * \mathcal{Q})[X/E] &\triangleq \mathcal{P}[X/E] * \mathcal{Q}[X/E] \\
(\mathcal{P} \mathrel{-\!\!*} \mathcal{Q})[X/E] &\triangleq \mathcal{P}[X/E] \mathrel{-\!\!*} \mathcal{Q}[X/E] \\
(*_{i \in I} \mathcal{P}_i)[X/E] &\triangleq *_{i \in I}(\mathcal{P}_i[x/E]) \\
(E_1 \overset{\pi}{\hookrightarrow}_t E_2)[X/E] &\triangleq E_1[X/E] \overset{\pi}{\hookrightarrow}_t E_2[X/E] \\
\mathsf{Proc}_\pi(Y, p, P, \Pi)[X/E] &\triangleq \mathsf{Proc}_\pi(Y, p, P, \Pi[X/E])
\end{aligned}
$$

APPENDIX **B**

Auxiliary Definitions for Chapter 7

B.1 Programs

B.1.1 Syntax of Programs

Definition B.1.1 (Free variables of expressions)**.** *The functions* $\mathsf{fv_e}(\cdot) : \mathit{Expr} \to 2^{\mathit{Var}}$ *and* $\mathsf{fv_b}(\cdot) : \mathit{Cond} \to 2^{\mathit{Var}}$ *define the set of free variables of expressions and Boolean conditions, respectively. These two functions are defined as follows.*

$$
\begin{aligned}
\mathsf{fv_e}(v) &\triangleq \emptyset \\
\mathsf{fv_e}(x) &\triangleq \{x\} \\
\mathsf{fv_e}(e_1 + e_2) &\triangleq \mathsf{fv_e}(e_1) \cup \mathsf{fv_e}(e_2) \\
\mathsf{fv_e}(e_1 - e_2) &\triangleq \mathsf{fv_e}(e_1) \cup \mathsf{fv_e}(e_2) \\
\mathsf{fv_b}(\mathsf{true}) &\triangleq \emptyset \\
\mathsf{fv_b}(\mathsf{false}) &\triangleq \emptyset \\
\mathsf{fv_b}(\neg b) &\triangleq \mathsf{fv_b}(b) \\
\mathsf{fv_b}(b_1 \wedge b_2) &\triangleq \mathsf{fv_b}(b_1) \cup \mathsf{fv_b}(b_2) \\
\mathsf{fv_b}(e_1 = e_2) &\triangleq \mathsf{fv_e}(e_1) \cup \mathsf{fv_e}(e_2) \\
\mathsf{fv_b}(e_1 < e_2) &\triangleq \mathsf{fv_e}(e_1) \cup \mathsf{fv_e}(e_2)
\end{aligned}
$$

However, in the thesis we overload the notation $\mathsf{fv}(\cdot)$, by simply writing $\mathsf{fv}(e)$ and $\mathsf{fv}(b)$, instead of $\mathsf{fv_e}(e)$ and $\mathsf{fv_b}(b)$, respectively. Moreover, we sometimes use the notation $\mathsf{fv}(e_1, \ldots, e_n)$ to abbreviate $\cup_{i=0}^{n} \mathsf{fv}(e_i)$. As usual, any expression e or condition b is defined to be *closed* if $\mathsf{fv}(e) = \emptyset$ or $\mathsf{fv}(b) = \emptyset$, respectively.

Definition B.1.2 (Substitution of expressions)**.** *Substitution of a variable x by an expression e within an expression e' or Boolean condition b, written $e'[x/e]$ and $b[x/e]$, respectively, is defined as follows.*

$$
\begin{aligned}
v[x/e] &\triangleq v \\
y[x/e] &\triangleq \begin{cases} e & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \\
(e_1 + e_2)[x/e] &\triangleq e_1[x/e] + e_2[x/e] \\
(e_1 - e_2)[x/e] &\triangleq e_1[x/e] - e_2[x/e] \\
\mathsf{true}[x/e] &\triangleq \mathsf{true} \\
\mathsf{false}[x/e] &\triangleq \mathsf{false} \\
(\neg b)[x/e] &\triangleq \neg(b[x/e]) \\
(b_1 \wedge b_2)[x/e] &\triangleq b_1[x/e] \wedge b_2[x/e] \\
(e_1 = e_2)[x/e] &\triangleq e_1[x/e] = e_2[x/e] \\
(e_1 < e_2)[x/e] &\triangleq e_1[x/e] < e_2[x/e]
\end{aligned}
$$

The notation $e[x_1, \ldots, x_n/e_1, \ldots, e_n]$ abbreviates $e[x_1/e_1] \cdots [x_n/e_n]$, and likewise for (Boolean) conditions.

B.1.2 Denotational Semantics

Definition B.1.3 (Store update)**.** *Given any store σ, the operation $\sigma[x \mapsto v]$ gives a new store that is equal to σ, but with x mapping to v.*

$$
\sigma[x \mapsto v] \triangleq \lambda y \in \mathit{Var}\,.\, \begin{cases} v & \text{if } x = y \\ \sigma(y) & \text{if } x \neq y \end{cases}
$$

The shorthand notation $\sigma[x_1 \mapsto v_1, \ldots x_n \mapsto v_n]$, i.e., a series of store updates, is sometimes used to abbreviate the store $\sigma[x_1 \mapsto v_1] \cdots [x_n \mapsto v_n]$.

Definition B.1.4 (Denotational semantics)**.** *The denotation semantics $[\![e]\!]_e\sigma$ and $[\![b]\!]_b\sigma$, of expressions e and Boolean conditions b, respectively, is defined as follows.*

$$
\begin{aligned}
[\![v]\!]_e\sigma &\triangleq v \\
[\![x]\!]_e\sigma &\triangleq \sigma(x) \\
[\![e_1 + e_2]\!]_e\sigma &\triangleq [\![+]\!]\big([\![e_1]\!]_e\sigma, [\![e_2]\!]_e\sigma\big) \\
[\![e_1 - e_2]\!]_e\sigma &\triangleq [\![-]\!]\big([\![e_1]\!]_e\sigma, [\![e_2]\!]_e\sigma\big)
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{true}]\!]_b\sigma &\triangleq \mathsf{true} \\
[\![\mathsf{false}]\!]_b\sigma &\triangleq \mathsf{false} \\
[\![\neg b]\!]_b\sigma &\triangleq \neg[\![b]\!]_b\sigma \\
[\![b_1 \wedge b_2]\!]_b\sigma &\triangleq [\![b_1]\!]_b\sigma \wedge [\![b_2]\!]_b\sigma \\
[\![e_1 = e_2]\!]_b\sigma &\triangleq [\![=]\!]\big([\![e_1]\!]_e\sigma, [\![e_2]\!]_e\sigma\big) \\
[\![e_1 < e_2]\!]_b\sigma &\triangleq [\![<]\!]\big([\![e_1]\!]_e\sigma, [\![e_2]\!]_e\sigma\big)
\end{aligned}
$$

However, in the thesis, we simply write $[\![e]\!]\sigma$ and $[\![b]\!]\sigma$ instead of $[\![e]\!]_e\sigma$ and $[\![b]\!]_b\sigma$, respectively, thereby overloading the $[\![\cdot]\!]\sigma$ notation. Moreover, we often simply write $[\![\cdot]\!]$ instead of $[\![\cdot]\!]\sigma$, if the expression/condition at the $\cdot$ is closed.

Lemma B.1.1. *The following standard properties hold for substitution. The same properties also hold for substitution within conditions.*

1. *For any x, e and e', if $x \notin \mathsf{fv}(e)$, then $e[x/e'] = e$.*
2. *For any x, e_1, e_2 and σ, it holds that $[\![e_1[x/e_2]]\!]\sigma = [\![e_1]\!]\sigma[x \mapsto [\![e_2]\!]\sigma]$.*

B.1.3 Operational Semantics

Definition B.1.5 (Locked commands)**.** *Any command C is defined to be* locked *if* $\mathsf{locked}(C)$*, where the* ***locked*** *predicate is defined as follows.*

$$
\frac{}{\mathsf{locked}(\mathbf{inatom}\ C)} \qquad \frac{\mathsf{locked}(C_1)}{\mathsf{locked}(C_1; C_2)} \qquad \frac{\mathsf{locked}(C_1)}{\mathsf{locked}(C_1 \parallel C_2)} \qquad \frac{\mathsf{locked}(C_2)}{\mathsf{locked}(C_1 \parallel C_2)}
$$

Definition B.1.6 (Small-step operational semantics of programs)**.**

$$
(\mathbf{skip}; C, h, \sigma) \xrightarrow{\mathit{cmp}} (C, h, \sigma) \qquad \frac{(C_1, h, \sigma) \xrightarrow{l} (C_1', h', \sigma')}{(C_1; C_2, h, \sigma) \xrightarrow{l} (C_1'; C_2, h', \sigma')}
$$

$$
(x := e, h, \sigma) \xrightarrow{\mathit{cmp}} (\mathbf{skip}, h, \sigma[x \mapsto [\![e]\!]\sigma])
$$

$$
(x := [e], h, \sigma) \xrightarrow{\mathit{cmp}} (\mathbf{skip}, h, \sigma[x \mapsto h([\![e]\!]\sigma)])
$$

$$
\frac{[\![e_1]\!]\sigma \in \mathsf{dom}(h)}{([e_1] := e_2, h, \sigma) \xrightarrow{\mathit{cmp}} (\mathbf{skip}, h[[\![e_1]\!]\sigma \mapsto [\![e_2]\!]\sigma], \sigma)}
$$

$$
\frac{[\![b]\!]\sigma}{(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, h, \sigma) \xrightarrow{\mathit{cmp}} (C_1, h, \sigma)}
$$

$$\frac{\neg[\![b]\!]\sigma}{(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2, h, \sigma) \xrightarrow{\mathit{cmp}} (C_2, h, \sigma)}$$

$$(\textbf{while } b \textbf{ do } C, h, \sigma) \xrightarrow{\mathit{cmp}} (\textbf{if } b \textbf{ then } (C; \textbf{while } b \textbf{ do } C) \textbf{ else skip}, h, \sigma)$$

$$\frac{v \notin \mathsf{dom}(h)}{(x := \textbf{alloc } e, h, \sigma) \xrightarrow{\mathit{cmp}} (\textbf{skip}, h[v \mapsto [\![e]\!]\sigma], \sigma[x \mapsto v])}$$

$$(\textbf{dispose } e, h, \sigma) \xrightarrow{\mathit{cmp}} (\textbf{skip}, h - [\![e]\!]\sigma, \sigma) \qquad \frac{\neg\mathsf{locked}(C_2) \quad (C_1, h, \sigma) \xrightarrow{l} (C_1', h', \sigma')}{(C_1 \parallel C_2, h, \sigma) \xrightarrow{l} (C_1' \parallel C_2, h', \sigma')}$$

$$\frac{\neg\mathsf{locked}(C_1) \quad (C_2, h, \sigma) \xrightarrow{l} (C_2', h', \sigma')}{(C_1 \parallel C_2, h, \sigma) \xrightarrow{l} (C_1 \parallel C_2', h', \sigma')} \qquad (\textbf{skip} \parallel \textbf{skip}, h, \sigma) \xrightarrow{\mathit{cmp}} (\textbf{skip}, h, \sigma)$$

$$(\textbf{atomic } C, h, \sigma) \xrightarrow{\mathit{cmp}} (\textbf{inatom } C, h, \sigma)$$

$$\frac{(C, h, \sigma) \xrightarrow{l} (C', h', \sigma')}{(\textbf{inatom } C, h, \sigma) \xrightarrow{l} (\textbf{inatom } C', h', \sigma')}$$

$$(\textbf{inatom skip}, h, \sigma) \xrightarrow{\mathit{cmp}} (\textbf{skip}, h, \sigma)$$

$$(\textbf{send } (e_1, e_2), h, \sigma) \xrightarrow{\mathit{send}([\![e_1]\!]\sigma, [\![e_2]\!]\sigma)} (\textbf{skip}, h, \sigma)$$

$$((x, y) := \textbf{recv}, h, \sigma) \xrightarrow{\mathit{recv}(v, v')} (\textbf{skip}, h, \sigma[x \mapsto v, y \mapsto v'])$$

$$(x := \textbf{recv } e, h, \sigma) \xrightarrow{\mathit{recv}(v, [\![e]\!]\sigma)} (\textbf{skip}, h, \sigma[x \mapsto v])$$

$$\frac{\neg\mathsf{locked}(C_1) \quad (C_1, h, \sigma) \xrightarrow{\mathit{send}(v_1, v_2)} (C_1', h, \sigma) \qquad \neg\mathsf{locked}(C_2) \quad (C_2, h, \sigma) \xrightarrow{\mathit{recv}(v_1, v_2)} (C_2', h, \sigma')}{(C_1 \parallel C_2, h, \sigma) \xrightarrow{\mathit{comm}(v_1, v_2)} (C_1' \parallel C_2', h, \sigma')}$$

$$\frac{\neg\mathsf{locked}(C_1) \quad (C_1, h, \sigma) \xrightarrow{\mathit{recv}(v_1,v_2)} (C_1', h, \sigma') \qquad \neg\mathsf{locked}(C_2) \quad (C_2, h, \sigma) \xrightarrow{\mathit{send}(v_1,v_2)} (C_2', h, \sigma)}{(C_1 \parallel C_2, h, \sigma) \xrightarrow{\mathit{comm}(v_1,v_2)} (C_1' \parallel C_2', h, \sigma')}$$

$$(\textbf{query}\ b, h, s) \xrightarrow{\mathit{qry}} (\textbf{skip}, h, s)$$

B.2 Processes

B.2.1 Syntax of Processes

Definition B.2.1 (Free variables of processes)**.** *The function* $\mathsf{fv}_p(\cdot) : \mathit{Proc} \to 2^{\mathit{Var}}$ *determines the set of free variables in a given process, and is defined as follows.*

$$\begin{aligned}
\mathsf{fv}_p(\varepsilon) &\triangleq \emptyset \\
\mathsf{fv}_p(\delta) &\triangleq \emptyset \\
\mathsf{fv}_p(\mathsf{send}(e_1, e_2)) &\triangleq \mathsf{fv}_e(e_1) \cup \mathsf{fv}_e(e_2) \\
\mathsf{fv}_p(\mathsf{recv}(e_1, e_2)) &\triangleq \mathsf{fv}_e(e_1) \cup \mathsf{fv}_e(e_2) \\
\mathsf{fv}_p(?b) &\triangleq \mathsf{fv}_b(b) \\
\mathsf{fv}_p(b : P) &\triangleq \mathsf{fv}_b(b) \cup \mathsf{fv}_p(P) \\
\mathsf{fv}_p(P \cdot Q) &\triangleq \mathsf{fv}_p(P) \cup \mathsf{fv}_p(Q) \\
\mathsf{fv}_p(P + Q) &\triangleq \mathsf{fv}_p(P) \cup \mathsf{fv}_p(Q) \\
\mathsf{fv}_p(P \parallel Q) &\triangleq \mathsf{fv}_p(P) \cup \mathsf{fv}_p(Q) \\
\mathsf{fv}_p(\Sigma_x\, P) &\triangleq \mathsf{fv}_p(P) \setminus \{x\} \\
\mathsf{fv}_p(P^*) &\triangleq \mathsf{fv}_p(P)
\end{aligned}$$

We overload the fv notation and often simply write $\mathsf{fv}(P)$, instead of $\mathsf{fv}_\mathsf{p}(P)$. Again, $\mathsf{fv}(P_1, \ldots, P_n)$ is written as shorthand for $\cup_{i=0}^{n} \mathsf{fv}(P_i)$.

Definition B.2.2 (Substitution in processes)**.** *The operation* $P[x/e]$ *gives the process* P*, but with every occurrence of* x *substituted for* e*, and is defined as:*

$$\begin{aligned}
\varepsilon[x/e] &\triangleq \varepsilon \\
\delta[x/e] &\triangleq \delta \\
\mathsf{send}(e_1, e_2)[x/e] &\triangleq \mathsf{send}(e_1[x/e], e_2[x/e]) \\
\mathsf{recv}(e_1, e_2)[x/e] &\triangleq \mathsf{recv}(e_1[x/e], e_2[x/e])
\end{aligned}$$

$$
\begin{aligned}
(?b)[x/e] &\triangleq ?(b[x/e]) \\
(b : P)[x/e] &\triangleq b[x/e] : P[x/e] \\
(P \cdot Q)[x/e] &\triangleq P[x/e] \cdot Q[x/e] \\
(P + Q)[x/e] &\triangleq P[x/e] + Q[x/e] \\
(P \parallel Q)[x/e] &\triangleq P[x/e] \parallel Q[x/e] \\
(\Sigma_y P)[x/e] &\triangleq \begin{cases} \Sigma_y P & \text{if } x = y \\ \Sigma_y (P[x/e]) & \text{if } x \neq y \end{cases} \\
P^*[x/e] &\triangleq P[x/e]^*
\end{aligned}
$$

We sometimes write $P[x_1, \ldots, x_n/e_1, \ldots, x_n]$ as a shorthand for $P[x_1/e_1] \cdots [x_n/e_n]$.

B.2.2 Axiomatisation

The following theorem states soundness of standard axioms of our process algebra language.

Theorem B.2.1. *The following bisimulation equivalences hold.*

$$
\begin{aligned}
\varepsilon \cdot P &\cong P \\
P \cdot \varepsilon &\cong P \\
\delta \cdot P &\cong \delta \\
P \cdot (Q \cdot R) &\cong (P \cdot Q) \cdot R \\
P + Q &\cong Q + P \\
P + (Q + R) &\cong (P + Q) + R \\
P + P &\cong P \\
P + \delta &\cong P \\
(P + Q) \cdot R &\cong (P \cdot R) + (Q \cdot R) \\
P \parallel Q &\cong Q \parallel P \\
P \parallel (Q \parallel R) &\cong (P \parallel Q) \parallel R \\
\varepsilon \parallel P &\cong P \\
P \parallel \delta &\cong P \cdot \delta \\
\mathsf{true} : P &\cong P
\end{aligned}
\qquad
\begin{aligned}
\mathsf{false} : P &\cong \delta \\
b_1 : b_2 : P &\cong b_1 \wedge b_2 : P \\
\Sigma_x P &\cong P[x/v] + \Sigma_x P \\
\Sigma_x (P + Q) &\cong \Sigma_x P + \Sigma_x Q \\
(\Sigma_x P) \cdot Q &\cong \Sigma_x (P \cdot Q) \ \textit{if } x \notin \mathsf{fv}(Q) \\
\Sigma_x b : P &\cong b : \Sigma_x P \ \textit{if } x \notin \mathsf{fv}(b) \\
\Sigma_x P &\cong P \ \textit{if } x \notin \mathsf{fv}(P) \\
\varepsilon^* &\cong \varepsilon \\
\delta^* &\cong \delta \\
P^{**} &\cong P^* \\
P^* &\cong (P \cdot P^*) + \varepsilon \\
P^* \cdot P^* &\cong P^* \\
(P + Q)^* &\cong P^* \cdot (Q \cdot P^*)^* \\
P^\omega &\cong P \cdot P^\omega
\end{aligned}
$$

B.3 Assertions

Definition B.3.1 (Free variables in assertions)**.** *The function* $\mathsf{fv}_a(\cdot) : Assn \to 2^{Var}$ *determines the set of free variables in a given assertion, and is defined as:*

$$\begin{aligned}
\mathsf{fv}_a(b) &\triangleq \mathsf{fv}_b(b) \\
\mathsf{fv}_a(\forall x.\mathcal{P}) &\triangleq \mathsf{fv}_a(\mathcal{P}) \setminus \{x\} \\
\mathsf{fv}_a(\exists x.\mathcal{P}) &\triangleq \mathsf{fv}_a(\mathcal{P}) \setminus \{x\} \\
\mathsf{fv}_a(\mathcal{P} \vee \mathcal{Q}) &\triangleq \mathsf{fv}_a(\mathcal{P}) \cup \mathsf{fv}_a(\mathcal{Q}) \\
\mathsf{fv}_a(\mathcal{P} * \mathcal{Q}) &\triangleq \mathsf{fv}_a(\mathcal{P}) \cup \mathsf{fv}_a(\mathcal{Q}) \\
\mathsf{fv}_a(\mathcal{P} \mathrel{-\!\!*} \mathcal{Q}) &\triangleq \mathsf{fv}_a(\mathcal{P}) \cup \mathsf{fv}_a(\mathcal{Q}) \\
\mathsf{fv}_a(e_1 \hookrightarrow_\pi e_2) &\triangleq \mathsf{fv}_e(e_1) \cup \mathsf{fv}_e(e_2) \\
\mathsf{fv}_a(\mathsf{Proc}(P)) &\triangleq \mathsf{fv}_p(P) \\
\mathsf{fv}_a(P \approx Q) &\triangleq \mathsf{fv}_p(P) \cup \mathsf{fv}_p(Q)
\end{aligned}$$

As before, we overload the fv operation and often simply write $\mathsf{fv}(\mathcal{P})$ instead of $\mathsf{fv}_a(\mathcal{P})$.

Definition B.3.2 (Substitution in assertions)**.** *The substitution operation* $\mathcal{P}[x/e]$ *replaces every occurrence of* x *with* e *in the assertion* $\mathcal{P}$*, and is defined as follows.*

$$\begin{aligned}
(\forall y.\mathcal{P})[x/e] &\triangleq \begin{cases} \forall y.\mathcal{P} & \text{if } x = y \\ \forall y.(\mathcal{P}[x/e]) & \text{if } x \neq y \end{cases} \\
(\exists y.\mathcal{P})[x/e] &\triangleq \begin{cases} \exists y.\mathcal{P} & \text{if } x = y \\ \exists y.(\mathcal{P}[x/e]) & \text{if } x \neq y \end{cases} \\
(\mathcal{P} \vee \mathcal{Q})[x/e] &\triangleq \mathcal{P}[x/e] \vee \mathcal{Q}[x/e] \\
(\mathcal{P} * \mathcal{Q})[x/e] &\triangleq \mathcal{P}[x/e] * \mathcal{Q}[x/e] \\
(\mathcal{P} \mathrel{-\!\!*} \mathcal{Q})[x/e] &\triangleq \mathcal{P}[x/e] \mathrel{-\!\!*} \mathcal{Q}[x/e] \\
(e_1 \hookrightarrow_\pi e_2)[x/e] &\triangleq e_1[x/e] \hookrightarrow_\pi e_2[x/e] \\
\mathsf{Proc}(P)[x/e] &\triangleq \mathsf{Proc}(P[x/e]) \\
(P \approx Q)[x/e] &\triangleq P[x/e] \approx Q[x/e]
\end{aligned}$$

B.4 Proof Rules

An overview of the proof rules of the program logic is given below. All proof rules are of the standard form $\mathcal{P} \vdash \mathcal{Q}$. Moreover, $\vdash \mathcal{P} \triangleq \mathsf{true} \vdash \mathcal{P}$ is derived and states

logical validity of $\mathcal{P}$.

$$\textsf{true}\text{-INTRO}\quad \mathcal{P} \vdash \textsf{true} \qquad \textsf{false}\text{-ELIM}\quad \textsf{false} \vdash \mathcal{P} \qquad \text{COND-DUPL}\quad b \vdash b * b$$

$$\forall\text{-INTRO}\quad \frac{\forall v.(\mathcal{P} \vdash \mathcal{Q}[x/v])}{\mathcal{P} \vdash \forall x.\mathcal{Q}} \qquad \forall\text{-ELIM}\quad \frac{\mathcal{P} \vdash \forall x.\mathcal{Q}}{\mathcal{P} \vdash \mathcal{Q}[x/v]}$$

$$\exists\text{-INTRO}\quad \frac{\mathcal{P} \vdash \mathcal{Q}[x/v]}{\mathcal{P} \vdash \exists x.\mathcal{Q}} \qquad \exists\text{-ELIM}\quad \frac{\mathcal{P} \vdash \exists x.\mathcal{Q}}{\exists v.(\mathcal{P} \vdash \mathcal{Q}[x/v])} \qquad \vee\text{-ASSOC}\quad \mathcal{P} \vee (\mathcal{Q} \vee \mathcal{R}) \dashv\vdash (\mathcal{P} \vee \mathcal{Q}) \vee \mathcal{R}$$

$$\vee\text{-COMM}\quad \mathcal{P} \vee \mathcal{Q} \vdash \mathcal{Q} \vee \mathcal{P} \qquad \vee\text{-ELIM-L}\quad \frac{\mathcal{P} \vdash \mathcal{Q}_1}{\mathcal{P} \vdash \mathcal{Q}_1 \vee \mathcal{Q}_2} \qquad \vee\text{-ELIM-R}\quad \frac{\mathcal{P} \vdash \mathcal{Q}_2}{\mathcal{P} \vdash \mathcal{Q}_1 \vee \mathcal{Q}_2} \qquad *\text{-WEAK}\quad \mathcal{P} * \mathcal{Q} \vdash \mathcal{P}$$

$$*\text{-ASSOC}\quad \mathcal{P} * (\mathcal{Q} * \mathcal{R}) \dashv\vdash (\mathcal{P} * \mathcal{Q}) * \mathcal{R} \qquad *\text{-COMM}\quad \mathcal{P} * \mathcal{Q} \vdash \mathcal{Q} * \mathcal{P} \qquad *\text{-MONO}\quad \frac{\mathcal{P} \vdash \mathcal{P}' \qquad \mathcal{Q} \vdash \mathcal{Q}'}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{P}' * \mathcal{Q}'}$$

$$*\text{-DISJ}\quad \mathcal{P} * (\mathcal{Q} \vee \mathcal{R}) \dashv\vdash (\mathcal{P} \vee \mathcal{Q}) * (\mathcal{P} \vee \mathcal{R}) \qquad -\!\!*\text{-INTRO}\quad \frac{\mathcal{P} * \mathcal{Q} \vdash \mathcal{R}}{\mathcal{P} \vdash \mathcal{Q} -\!\!* \mathcal{R}} \qquad -\!\!*\text{-ELIM}\quad \frac{\mathcal{P} \vdash \mathcal{Q} -\!\!* \mathcal{R}}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{R}}$$

$$\hookrightarrow\text{-SPLIT-MERGE}\quad e_1 \hookrightarrow_{\pi_1+\pi_2} e_2 \dashv\vdash e_1 \hookrightarrow_{\pi_1} e_2 * e_1 \hookrightarrow_{\pi_2} e_2 \qquad \textsf{Proc}\text{-SPLIT-MERGE}\quad \textsf{Proc}(P \parallel Q) \dashv\vdash \textsf{Proc}(P) * \textsf{Proc}(Q)$$

$$\approx\text{-BISIM}\quad \frac{P \cong Q}{\vdash P \approx Q} \qquad \approx\text{-REFL}\quad \vdash P \approx P \qquad \approx\text{-SYMM}\quad P \approx Q \vdash Q \approx P \qquad \approx\text{-TRANS}\quad P \approx Q * Q \approx R \vdash P \approx R$$

$$\approx\text{-CONG-SEQ}\quad P \approx P' * Q \approx Q' \vdash P \cdot Q \approx P' \cdot Q' \qquad \approx\text{-CONG-ALT}\quad P \approx P' * Q \approx Q' \vdash P + Q \approx P' + Q'$$

$$\approx\text{-CONG-PAR}\quad P \approx P' * Q \approx Q' \vdash P \parallel Q \approx P' \parallel Q' \qquad \approx\text{-CONG-SUM}\quad \frac{x \notin \textsf{fv}(P, Q)}{P \approx Q \vdash \Sigma_x P \approx \Sigma_x Q}$$

$$\approx\text{-CONG-COND}\quad P \approx Q \vdash b : P \approx b : Q \qquad \approx\text{-CONG-ITER}\quad P \approx Q \vdash P^* \approx Q^* \qquad \approx\text{-COND-TRUE}\quad b \vdash b : P \approx P \qquad \approx\text{-COND-FALSE}\quad b \vdash \neg b : P \approx \delta$$

$$\textsf{Proc}\text{-}\approx\quad \textsf{Proc}(P) * P \approx Q \vdash \textsf{Proc}(Q)$$

In addition, all the axiom rules in Appendix B.4 are logically valid if one replaces

all $\cong$ equivalences by $\approx$, for example $\vdash \varepsilon \cdot P \approx P$, $\vdash \Sigma_x P \approx P[x/v] + \Sigma_x P$, et cetera.

Let $\mathit{Models} \triangleq \mathit{Heap} \times \mathit{Store} \times \mathit{Proc}$ be the universe set of models of the logic and let the denotation $[\![\mathcal{P}]\!] \triangleq \{(\iota, \sigma, P) \mid \iota, \sigma, P \models \mathcal{P}\} \subseteq \mathit{Models}$ be the set of models of the assertion $\mathcal{P}$. Then all entailment rules shown above are sound in the standard sense:

Theorem B.4.1 (Soundness of the entailment rules)**.**

$$\mathcal{P} \vdash \mathcal{Q} \implies [\![\mathcal{P}]\!] \subseteq [\![\mathcal{Q}]\!]$$

.

B.5 Program Logic

An overview of the Hoare-triple rules of the program logic is given below. These rules are all sound as stated by Theorem 7.4.3 (on page 226). The notation $e \hookrightarrow_\pi -$ is shorthand for $\exists x \,.\, e \hookrightarrow_\pi x$. Furthermore, $\mathsf{mod}(C) \subseteq \mathsf{fv}(C)$ gives the set of free variables that are written to by C.

HT-SKIP
$$\mathcal{I} \vdash \{\mathcal{P}\}\,\mathbf{skip}\,\{\mathcal{P}\}$$

HT-FRAME
$$\frac{\mathcal{I} \vdash \{\mathcal{P}\}\,C\,\{\mathcal{Q}\} \qquad \mathsf{fv}(\mathcal{R}) \cap \mathsf{mod}(C) = \emptyset}{\mathcal{I} \vdash \{\mathcal{P} * \mathcal{R}\}\,C\,\{\mathcal{Q} * \mathcal{R}\}}$$

HT-SEQ
$$\frac{\mathcal{I} \vdash \{\mathcal{P}\}\,C_1\,\{\mathcal{Q}\} \quad \mathcal{I} \vdash \{\mathcal{Q}\}\,C_2\,\{\mathcal{R}\}}{\mathcal{I} \vdash \{\mathcal{P}\}\,C_1; C_2\,\{\mathcal{R}\}}$$

HT-CONSEQ
$$\frac{\mathcal{P} \vdash \mathcal{P}' \qquad \mathcal{I} \vdash \{\mathcal{P}'\}\,C\,\{\mathcal{Q}'\} \qquad \mathcal{Q}' \vdash \mathcal{Q}}{\mathcal{I} \vdash \{\mathcal{P}\}\,C\,\{\mathcal{Q}\}}$$

HT-ITE
$$\frac{\mathcal{I} \vdash \{\mathcal{P} * b\}\,C_1\,\{\mathcal{Q}\} \quad \mathcal{I} \vdash \{\mathcal{P} * \neg b\}\,C_2\,\{\mathcal{Q}\}}{\mathcal{I} \vdash \{\mathcal{P}\}\,\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\,\{\mathcal{Q}\}}$$

HT-WHILE
$$\frac{\mathcal{I} \vdash \{\mathcal{P} * b\}\,C\,\{\mathcal{P}\}}{\mathcal{I} \vdash \{\mathcal{P}\}\,\mathbf{while}\ b\ \mathbf{do}\ C\,\{\mathcal{P} * \neg b\}}$$

HT-ASSIGN
$$\frac{x \notin \mathsf{fv}(\mathcal{I})}{\mathcal{I} \vdash \{\mathcal{P}[x/e]\}\,x := e\,\{\mathcal{P}\}}$$

HT-READ
$$\frac{x \notin \mathsf{fv}(\mathcal{I}, e, e')}{\mathcal{I} \vdash \{\mathcal{P}[x/e'] * e \hookrightarrow_\pi e'\}\,x := [e]\,\{\mathcal{P} * e \hookrightarrow_\pi e'\}}$$

HT-WRITE
$$\mathcal{I} \vdash \{e \hookrightarrow_1 -\}\,[e] := e'\,\{e \hookrightarrow_1 e'\}$$

HT-ALLOC
$$\frac{x \notin \mathsf{fv}(\mathcal{I}, e)}{\mathcal{I} \vdash \{\mathsf{true}\}\,x := \mathbf{alloc}\ e\,\{x \hookrightarrow_1 e\}}$$

HT-DISPOSE
$$\mathcal{I} \vdash \{e \hookrightarrow_1 -\}\ \mathbf{dispose}\ e\ \{\mathsf{true}\}$$

HT-PAR
$$\frac{\mathcal{I} \vdash \{\mathcal{P}_1\}\ C_1\ \{\mathcal{Q}_1\} \qquad \mathsf{fv}(\mathcal{I}, \mathcal{P}_1, C_1) \cap \mathsf{mod}(C_2) = \emptyset \\ \mathcal{I} \vdash \{\mathcal{P}_2\}\ C_2\ \{\mathcal{Q}_2\} \qquad \mathsf{fv}(\mathcal{I}, \mathcal{P}_2, C_2) \cap \mathsf{mod}(C_1) = \emptyset}{\mathcal{I} \vdash \{\mathcal{P}_1 * \mathcal{P}_2\}\ C_1 \parallel C_2\ \{\mathcal{Q}_1 * \mathcal{Q}_2\}}$$

HT-ATOM
$$\frac{\mathsf{true} \vdash \{\mathcal{P} * \mathcal{I}\}\ C\ \{\mathcal{Q} * \mathcal{I}\}}{\mathcal{I} \vdash \{\mathcal{P}\}\ \mathbf{atomic}\ C\ \{\mathcal{Q}\}}$$

HT-SHARE
$$\frac{\mathcal{I} * \mathcal{I}' \vdash \{\mathcal{P}\}\ C\ \{\mathcal{Q}\}}{\mathcal{I} \vdash \{\mathcal{P} * \mathcal{I}'\}\ C\ \{\mathcal{Q} * \mathcal{I}'\}}$$

HT-DISJ
$$\frac{\mathcal{I} \vdash \{\mathcal{P}_1\}\ C\ \{\mathcal{Q}_1\} \\ \mathcal{I} \vdash \{\mathcal{P}_2\}\ C\ \{\mathcal{Q}_2\}}{\mathcal{I} \vdash \{\mathcal{P}_1 \vee \mathcal{P}_2\}\ C\ \{\mathcal{Q}_1 \vee \mathcal{Q}_2\}}$$

HT-EX
$$\frac{\mathcal{I} \vdash \{\mathcal{P}\}\ C\ \{\mathcal{Q}\} \qquad x \notin \mathsf{fv}(C)}{\mathcal{I} \vdash \{\exists x.\mathcal{P}\}\ C\ \{\exists x.\mathcal{Q}\}}$$

HT-SEND
$$\mathcal{I} \vdash \{\mathsf{Proc}(\mathsf{send}(e_1, e_2) \cdot P)\}\ \mathbf{send}\ (e_1, e_2)\ \{\mathsf{Proc}(P)\}$$

HT-RECV
$$\frac{x \notin \mathsf{fv}(\mathcal{I}) \cup \mathsf{fv}(P) \qquad y \notin \mathsf{fv}(e)}{\mathcal{I} \vdash \{\mathsf{Proc}(\Sigma_y\, \mathsf{recv}(y, e) \cdot P)\}\ x := \mathbf{recv}\ e\ \{\mathsf{Proc}(P[y/x])\}}$$

HT-RECV-WILDCARD
$$\frac{x_1, x_2 \notin \mathsf{fv}(\mathcal{I}) \cup \mathsf{fv}(P) \qquad \{x_1, y_1\} \cap \{x_2, y_2\} = \emptyset}{\mathcal{I} \vdash \{\mathsf{Proc}(\Sigma_{y_1, y_2}\, \mathsf{recv}(y_1, y_2) \cdot P)\}\ (x_1, x_2) := \mathbf{recv}\ \{\mathsf{Proc}(P[y_1/x_1][y_2/x_2])\}}$$

HT-QUERY
$$\mathcal{I} \vdash \{\mathsf{Proc}(?b \cdot P)\}\ \mathbf{query}\ b\ \{\mathsf{Proc}(P) * b\}$$

APPENDIX C

Publications by the Author

- W. Oortwijn, D. Gurov, and M. Huisman. Practical Abstractions for Automated Verification of Shared-Memory Concurrency. In D. Beyer and D. Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS. Springer, 2020. To appear
- W. Oortwijn, M. Huisman, S. Joosten, and J. van de Pol. Automated Verification of Parallel Nested DFS. In *Submitted*, 2019
- W. Ahrendt, L. Henrio, and W. Oortwijn. Who is to Blame? Runtime Verification of Distributed Objects with Active Monitors. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 302:32–46, 2019
- W. Oortwijn and M. Huisman. Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System. In W. Ahrendt and S. L. Tapia Tarifa, editors, *integrated Formal Methods (iFM)*, LNCS. Springer, 2019. To appear
- W. Oortwijn and M. Huisman. Practical Abstractions for Automated Verification of Message Passing Concurrency. In W. Ahrendt and S. L. Tapia Tarifa, editors, *integrated Formal Methods (iFM)*, LNCS. Springer, 2019. To appear
- S. Joosten, W. Oortwijn, M. Safari, and M. Huisman. An Exercise in Verifying Sequential Programs with VerCors. In A. Summers, editor, *Formal Techniques for Java-like Programs (FTfJP)*. ACM, 2018
- S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In N. Polikarpova and S. Schneider, editors, *integrated Formal Methods (iFM)*, volume 10510 of *LNCS*, pages 102–110. Springer, 2017
- W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. An Abstraction Technique for Describing Concurrent Program Behaviour. In

A. Paskevich and T. Wies, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 10712 of *LNCS*, pages 191–209, 2017

- W. Oortwijn, T. van Dijk, and J. van de Pol. Distributed Binary Decision Diagrams for Symbolic Reachability. In *SPIN*, pages 21–30. ACM, 2017 **(Best paper award)**
- W. Oortwijn, S. Blom, and M. Huisman. Future-based Static Analysis of Message Passing Programs. In *Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES)*, pages 65–72. Open Publishing Association, 2016
- W. Oortwijn, T. van Dijk, and J. van de Pol. A Distributed Hash Table for Shared Memory. In R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski, and K. Wiatr, editors, *Parallel Processing and Applied Mathematics (PPAM)*, pages 15–24. Springer, 2016

Bibliography

[AAM+17] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. Belanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. In *CoqPL 2017: Third International Workshop on Coq for Programming Languages (2017)*, 2017.

[AB07] A. Appel and S. Blazy. Separation Logic for Small-Step CMINOR. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 5–21. Springer Berlin Heidelberg, 2007.

[AB18] S. Ahmed and M. Bagherzadeh. What Do Concurrency Developers Ask About?: A Large-scale Study Using Stack Overflow. In *Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. ACM, 2018.

[ABB+16] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich. *Deductive Software Verification – The KeY Book*, volume 10001 of *LNCS*. Springer International Publishing, 2016.

[ABC10] A. Aldini, M. Bernardo, and F. Corradini. *A Process Algebraic Approach to Software Architecture Design*. Springer Science & Business Media, 2010.

[ABH14] A. Amighi, S. Blom, and M. Huisman. Resource Protection Using Atomics - Patterns and Verification. In J. Garrigue, editor, *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 255–274. Springer, 2014.

[ACPS15] W. Ahrendt, J. Chimento, G. Pace, and G. Schneider. A Specification Language for Static and Runtime Verification of Data and

Control Properties. In N. Bjørner and F. de Boer, editors, *Formal Methods (FM)*. Springer International Publishing, 2015.

[ACPS17] W. Ahrendt, J. Chimento, G. Pace, and G. Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods in System Design*, 51(1):200–265, 2017.

[ADBH15] A. Amighi, S. Darabi, S. Blom, and M. Huisman. Specification and Verification of Atomic Operations in GPGPU Programs. In R. Calinescu and B. Rumpe, editors, *Software Engineering and Formal Methods (SEFM)*, volume 9276 of *LNCS*. Springer, 2015.

[AHHH15] A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded Java programs. *Logical Methods in Computer Science*, 11(1), 2015.

[AHO19] W. Ahrendt, L. Henrio, and W. Oortwijn. Who is to Blame? Runtime Verification of Distributed Objects with Active Monitors. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 302:32–46, 2019.

[Ami18] A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. PhD thesis, University of Twente, 2018.

[AMRV07] A. Appel, P. Melliès, C. Richards, and J. Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *Principles of Programming Languages (POPL)*, pages 109–122. ACM, 2007.

[App11a] A. Appel. Verified Software Toolchain. In G. Barthe, editor, *European Symposium on Programming (ESOP)*, pages 1–17. Springer, 2011.

[App11b] A. Appel. VeriSmall: Verified Smallfoot Shape Analysis. In J. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs (CPP)*, pages 231–246. Springer, 2011.

[APS16] W. Ahrendt, G. Pace, and G. Schneider. StaRVOOrS — Episode II. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA)*, pages 402–415. Springer International Publishing, 2016.

[ARSCT09] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A compact kernel for the calculus of inductive constructions. *Sadhana*, 34(1):71–144, 2009.

[Bae00] J. Baeten. *Process Algebra with Explicit Termination*. Eindhoven University of Technology, Department of Mathematics and Computing Science, 2000.

[BBDL+18] J. Barnat, V. Bloemen, A. Duret-Lutz, A. Laarman, L. Petrucci, J. van de Pol, and E. Renault. Parallel Model Checking Algorithms for Linear-Time Temporal Logic. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 457–507. Springer International Publishing, 2018.

[BC06] J. Barnat and I. Cerná. Distributed breadth-first search LTL model checking. *Formal Methods in System Design*, 29(2):117–134, 2006.

[BC10] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer, 1 edition, 2010.

[BCO05] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic Execution with Separation Logic. In K. Yi, editor, *Programming Languages and Systems (APLAS)*, pages 52–68. Springer Berlin Heidelberg, 2005.

[BCOP05] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *Principles of Programming Languages (POPL)*, pages 259–270, 2005.

[BCY06] R. Bornat, C. Calcagno, and H. Yang. Variables As Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.

[BDH15] S. Blom, S. Darabi, and M. Huisman. Verification of Loop Parallelisations. In A. Egyed and I. Schaefer, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 9033 of *LNCS*, pages 202–217. Springer, 2015.

[BDHO17] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In N. Polikarpova and S. Schneider, editors, *integrated Formal Methods (iFM)*, volume 10510 of *LNCS*, pages 102–110. Springer, 2017.

[Bee08] R. Beers. Pre-RTL formal verification: An Intel experience. In *ACM/IEEE Design Automation Conference*, pages 806–811. IEEE, 2008.

[Bel10] A. Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 266–270. Springer Berlin Heidelberg, 2010.

[Bey19] D. Beyer. Automatic Verification of C and Java Programs: SV-COMP 2019. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 133–155. Springer, 2019.

[BGK+19] O. Bunte, J.F. Groote, J. Keiren, M. Laveaux, T. Neele, E. de Vink, W. Wesselink, A. Wijs, and T. Willemse. The mCRL2 Toolset for Analysing Concurrent Systems. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 21–39. Springer, 2019.

[BGKB19] A. Bizjak, D. Gratzer, R. Krebbers, and L. Birkedal. Iron: Managing Obligations in Higher-order Concurrent Separation Logic. *Principles of Programming Languages (POPL)*, 3:1–30, 2019.

[BH14a] B. Beckert and R. Hähnle. Reasoning and Verification: State of the Art and Current Trends. *IEEE Intelligent Systems*, 29(1):20–29, 2014.

[BH14b] D. Bjørner and K. Havelund. 40 Years of Formal Methods. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *Formal Methods (FM)*, pages 42–61. Springer International Publishing, 2014.

[BH14c] S. Blom and M. Huisman. The VerCors Tool for Verification of Concurrent Programs. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *Formal Methods (FM)*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.

[BHZS15] S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. History-Based Verification of Functional Behaviour of Concurrent Programs. In R. Calinescu and B. Rumpe, editors, *Software Engineering and Formal Methods (SEFM)*, volume 9276 of *LNCS*, pages 84–98. Springer, 2015.

[BK84] J. Bergstra and J.W. Klop. Process algebra for Synchronous Communication. *Information and Control*, 60(1):109–137, 1984.

[BK08] C. Baier and J.P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BKLL15] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C. In M. Núñez and M. Güdemann, editors, *Formal Methods for Industrial Critical Systems (FMICS)*, pages 15–30. Springer, 2015.

[BL18] J. Brunner and P. Lammich. Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. *Journal of Automated Reasoning*, 60(1):3–21, 2018.

[Blo19] V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. PhD thesis, University of Twente, 2019.

[BLP16] V. Bloemen, A. Laarman, and J. van de Pol. Multi-core On-the-fly SCC Decomposition. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12. ACM, 2016.

[BMR12] N. Bjørner, K. McMillan, and A. Rybalchenko. Program Verification as Satisfiability Modulo Theories. In *SMT*, 2012.

[BO16] S. Brookes and P. O'Hearn. Concurrent Separation Logic. *ACM SIGLOG News*, 3(3):47–65, 2016.

[Boy03] J. Boyland. Checking Interference with Fractional Permissions. In R. Cousot, editor, *Static Analysis (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[BR15] D. Bogdanas and G. Roşu. K-Java: A Complete Semantics of Java. In *Principles of Programming Languages (POPL)*, pages 445–456. ACM, 2015.

[Bro07] S. Brookes. A Semantics for Concurrent Separation Logic. *Theoretical Computer Science*, 375(1–3):227–270, 2007.

[Bur71] R.M. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. In *Machine Intelligence*, pages 23–50, 1971.

[CCD+14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv Symbolic Model Checker. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV)*, pages 334–342. Springer International Publishing, 2014.

[CCL+18] R. Chen, C. Cohen, J. Lévy, S. Merz, and L. Théry. Formal Proofs of Tarjan's Algorithm in Why3, Coq, and Isabelle. *The Computing Research Repository (CoRR)*, 2018.

[CD11] C. Calcagno and D. Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods (NFM)*, pages 459–465. Springer Berlin Heidelberg, 2011.

[CDCPY15] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. A Gentle Introduction to Multiparty Asynchronous Session Types. In M. Bernardo and E. Johnsen, editors, *Formal Methods for Multicore Programming (SFM)*, pages 146–178. Springer, 2015.

[CDD+15] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving Fast with Software Verification. In K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods (NFM)*, pages 3–11. Springer International Publishing, 2015.

[CDDL12] P. Cuoq, D. Delmas, S. Duprat, and V. Lamiel. Fan-C, a Frama-C plug-in for data flow verification. In *Embedded Real Time Software and Systems (ERTS)*, 2012.

[CE82] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, pages 52–71. Springer Berlin Heidelberg, 1982.

[CFT] The CSL Family Tree of Ilya Sergey. `https://ilyasergey.net/other/CSL-Family-Tree.pdf` *(accessed on May 2019).*

[CG12] A. Cimatti and A. Griggio. Software Model Checking via IC3. In P. Madhusudan and S. Seshia, editors, *Computer-Aided Verification (CAV)*, pages 277–293. Springer Berlin Heidelberg, 2012.

[CGP03] J. Cobleigh, D. Giannakopoulou, and C. Păsăreanu. Learning Assumptions for Compositional Verification. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 331–346. Springer, 2003.

[CHJ+19] D. Castro, R. Hu, S. Jongmans, N. Ng, and N. Yoshida. Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures. *Principles of Programming Languages (POPL)*, 3:1–30, 2019.

[Chl10] A. Chlipala. A Verified Compiler for an Impure Functional Language. In *Principles of Programming Languages (POPL)*, pages 93–106. ACM, 2010.

[CHVB18] E. Clarke, T. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

[CK99] S.C. Cheung and J. Kramer. Checking Safety Properties using Compositional Reachability Analysis. *Transactions on Software Engineering and Methodology (TOSEM)*, pages 49–78, 1999.

[CK05] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 108–128. Springer, 2005.

[CKL04] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176. Springer, 2004.

[CKSY05] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 570–574. Springer, 2005.

[Cla08] E. Clarke. The Birth of Model Checking. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking: History, Achievements, Perspectives*, pages 1–26. Springer Berlin Heidelberg, 2008.

[CLM89] E. Clarke, D. Long, and K. McMillan. Compositional Model Checking. In *Logic in Computer Science (LICS)*, pages 353–362. IEEE, 1989.

[CLSE05] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model Variables: Cleanly Supporting Abstraction in Design by Contract: Research Articles. *Software–Practice and Experience*, 35(6):583–599, 2005.

[Cok14] D. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *1st Workshop on Formal Integrated Development Environment*, pages 79–92. Open Publishing Association, 2014.

[Cok18] D. Cok. Java Automated Deductive Verification in Practice: Lessons from Industrial Proof-Based Projects. In T. Margaria and B. Steffen, editors, *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 176–193. Springer, 2018.

[CPV07] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular Safety Checking for Fine-Grained Concurrency. In H. Nielson and G. Filé, editors, *Static Analysis (SAS)*, pages 233–248. Springer, 2007.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2–3):275–288, 1992.

[CWP] INRIA - The Coq webpage. `https://coq.inria.fr` *(accessed on May 2019)*.

[Dar18] S. Darabi. *Verification of Program Parallelization.* PhD thesis, University of Twente, 2018.

[DBG18] M. Dodds, M. Batty, and A. Gotsman. Compositional Verification of Compiler Optimisations on Relaxed Memory. In A. Ahmed, editor, *Programming Languages and Systems (ESOP)*, pages 1027–1055. Springer, 2018.

[DBH17] S. Darabi, S. Blom, and M. Huisman. A Verification Technique for Deterministic Parallel Programs. In C. Barrett, M. Davies, and T. Kahsai, editors, *NASA Formal Methods (NFM)*, volume 10227 of *LNCS*, pages 247–264, 2017.

[DFPV09] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In G. Castagna, editor, *European Symposium on Programming (ESOP)*, volume 5502 of *LNCS*, pages 363–377. Springer, 2009.

[Dij76] E. Dijkstra. *A Discipline of Programming.* Englewood Cliffs: Prentice-Hall, 1976.

[DP15] T. van Dijk and J. van de Pol. Sylvan: Multi-Core Decision Diagrams. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 677–691. Springer Berlin Heidelberg, 2015.

[DYDG+10] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In T. D'Hondt, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *LNCS*, pages 504–528, 2010.

[EC80] E. Emerson and E. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming (ICALP)*, pages 169–181. Springer Berlin Heidelberg, 1980.

[EHMU19] G. Ernst, M. Huisman, W. Mostowski, and M. Ulbrich. VerifyThis – Verification Competition with a Human Factor. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 176–195. Springer, 2019.

[ELN+13] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J. Smaus. A Fully Verified Executable LTL Model Checker. In

N. Sharygina and H. Veith, editors, *Computer-Aided Verification (CAV)*, pages 463–478. Springer, 2013.

[ELPP12] S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved Multi-Core Nested Depth-First Search. In S. Chakraborty and M. Mukund, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 7561 of *LNCS*, pages 269–283. Springer, 2012.

[EM18] M. Eilers and P. Müller. Nagini: A Static Verifier for Python. In H. Chockler and G. Weissenbacher, editors, *Computer-Aided Verification (CAV)*, pages 596–603. Springer, 2018.

[EPY11] S. Evangelista, L. Petrucci, and S. Youcef. Parallel Nested Depth-First Searches for LTL Model Checking. In T. Bultan and P. Hsiung, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 6996 of *LNCS*, pages 381–396. Springer, 2011.

[Fen09] X. Feng. Local Rely-Guarantee Reasoning. In *Principles of Programming Languages (POPL)*, volume 44, pages 315–327. ACM, 2009.

[FFS07] X. Feng, R. Ferreira, and Z. Shao. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In R. De Nicola, editor, *European Symposium on Programming (ESOP)*, pages 173–188. Springer, 2007.

[Fil11] J. Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):397–403, 2011.

[Flo67] R. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–31, 1967.

[Fok13] W. Fokkink. *Distributed Algorithms: An Intuitive Approach*. The MIT Press, 2013.

[FP13] J. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *European Symposium on Programming (ESOP)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[FRS11] A. Francalanza, J. Rathke, and V. Sassone. Permission-based Separation Logic for Message-Passing Concurrency. *Logical Methods in Computer Science*, 7, 2011.

[FZ94] W. Fokkink and H. Zantema. Basic Process Algebra with Iteration: Completeness of its Equational Axioms. *The Computer Journal*, 37(4):259–267, 1994.

[GBC+07] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In Z. Shao, editor, *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007.

[GHPT16] R. van Glabbeek, P. Höfner, M. Portmann, and W. Tan. Modelling and verifying the AODV routing protocol. *Distributed Computing*, 29(4):279–315, 2016.

[GJS+15] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu. What Change History Tells Us About Thread Synchronization. In *Foundations of Software Engineering (FSE)*, pages 426–438. ACM, 2015.

[GM14] J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.

[GMR+07] J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The Formal Specification Language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[GP95] J.F. Groote and A. Ponse. The Syntax and Semantics of mCRL. In *Algebra of Communicating Processes*, pages 26–62. Springer, 1995.

[GRB+15] S. de Gouw, J. Rot, F. de Boer, R. Bubel, and R. Hähnle. OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case. In D. Kroening and C. Pasareanu, editors, *Computer Aided Verification (CAV)*, volume 9206 of *LNCS*, pages 273–289. Springer, 2015.

[GRT18] A. Griggio, M. Roveri, and S. Tonetta. Certifying Proofs for LTL Model Checking. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 225–233. IEEE, 2018.

[GVR03] S. Gay, V. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. In *Technical Report TR-2003-133, Department of Computing Science, University of Glasgow*, 2003.

[GW16] J.F. Groote and A. Wijs. An $O(m \log n)$ Algorithm for Stuttering Equivalence and Branching Bisimulation. In M. Chechik and

J. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 607–624. Springer Berlin Heidelberg, 2016.

[HAN08] A. Hobor, A. Appel, and F. Nardelli. Oracle Semantics for Concurrent Separation Logic. In S. Drossopoulou, editor, *Programming Languages and Systems (ESOP)*, pages 353–367. Springer Berlin Heidelberg, 2008.

[HDV11] C. Hur, D. Dreyer, and V. Vafeiadis. Separation Logic in the Presence of Garbage Collection. *Logic in Computer Science (LICS)*, pages 247–256, 2011.

[HH17] R. Hähnle and M. Huisman. 24 Challenges in Deductive Software Verification. In G. Reger and D. Traytel, editors, *Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements (ARCADE)*, volume 51 of *EPiC Series in Computing*, pages 37–41. EasyChair, 2017.

[HHH08] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In G. Ramalingam, editor, *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 171–187. Springer, 2008.

[HHK+15] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M. Roberts, S. Setty, and B. Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Symposium on Operating Systems Principles (SOSP)*, pages 1–17. ACM, 2015.

[HHL+14] C. Hawblitzel, J. Howell, J. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Operating Systems Design and Implementation (OSDI)*, pages 165–181. USENIX Association, 2014.

[HHN+14] K. Honda, R. Hu, R. Neykova, T. Chen, R. Demangeon, P. Deniélou, and N. Yoshida. Structuring Communication with Session Types. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, pages 105–127. Springer Berlin Heidelberg, 2014.

[HJ18] M. Huisman and S. Joosten. Towards Reliable Concurrent Software. In P. Müller and I. Schaefer, editors, *Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, pages 129–146. Springer International Publishing, 2018.

[HJG11] G. Holzmann, R. Joshi, and A. Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.

[HKK+13] Y. Hwong, J. Keiren, V. Kusters, S. Leemans, and T. Willemse. Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. *Science of Computer Programming*, 78(12):2435–2452, 2013.

[HMM+12] K. Honda, E. Marques, F. Martins, N. Ng, V. Vasconcelos, and N. Yoshida. Verification of MPI Programs using Session Types. In *Recent Advances in the Message Passing Interface (EuroMPI)*, volume 7940 of *LNCS*, pages 291–293. Springer, 2012.

[HMM+19] M. Huisman, R. Monahan, P. Müller, A. Paskevich, and G. Ernst. VerifyThis 2018: A Program Verification Competition. *Technical report, Inria*, 2019.

[HN18] L. Hupel and T. Nipkow. A Verified Compiler from Isabelle/HOL to CakeML. In A. Ahmed, editor, *Programming Languages and Systems (ESOP)*, pages 999–1026. Springer, 2018.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM (CACM)*, 12(10):576–580, 1969.

[Hob08] A. Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.

[HPY96] G. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In *The Spin Verification System*, volume 32 of *DIMACS*, pages 23–32. DIMACS/AMS, 1996.

[HVK98] K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In C. Hankin, editor, *European Symposium on Programming (ESOP)*, pages 122–138. Springer, 1998.

[HW90] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[HYC08] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *Principles of Programming Languages (POPL)*, pages 273–284. ACM, 2008.

[IO01] S. Ishtiaq and P. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *Principles of Programming Languages (POPL)*, pages 14–26. ACM, 2001.

[IYG+08] F. Ivančić, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based Bounded Model Checking for Software Verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[JJKD17] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Principles of Programming Languages (POPL)*, 2:66:1–66:34, 2017.

[JKBD16] R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-Order Ghost State. In *International Conference on Functional Programming (ICFP)*, volume 51, pages 256–269. ACM, 2016.

[JKM+14] U. Juhasz, I. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. Technical report, ETH Zürich, 2014.

[Jon83] C. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[JOSH18] S. Joosten, W. Oortwijn, M. Safari, and M. Huisman. An Exercise in Verifying Sequential Programs with VerCors. In A. Summers, editor, *Formal Techniques for Java-like Programs (FTfJP)*. ACM, 2018.

[JSP+11] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods (NFM)*, pages 41–55. Springer Berlin Heidelberg, 2011.

[JSS+15] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Principles of Programming Languages (POPL)*, pages 637–650. ACM, 2015.

[JVP15] B. Jacobs, F. Vogels, and F. Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, 11(3), 2015.

[KeY] The KeY Project web page. `https://www.key-project.org` *(accessed on June 2019).*

[KJB+17] R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In H. Yang, editor, *European Symposium on Programming (ESOP)*, volume 10201 of *LNCS*, pages 696–723. Springer, 2017.

[KJTOL19] M. Krogh-Jespersen, A. Timany, M. Ohlenbusch, and Birkedal L. Aneris: A Logic for Node-Local, Modular Reasoning of Distributed Systems. *Unpublished*, 2019.

[KKP+15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing (FAOC)*, 27(3):573–609, 2015.

[KLM+15] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 692–707. Springer Berlin Heidelberg, 2015.

[KMNO14] R. Kumar, M. Myreen, M. Norrish, and S. Owens. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM, 2014.

[Koz82] D. Kozen. Results on the propositional μ-calculus. In M. Nielsen and E. Schmidt, editors, *Automata, Languages and Programming (ICALP)*, pages 348–359. Springer Berlin Heidelberg, 1982.

[Kre14] R. Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *Principles of Programming Languages (POPL)*, pages 101–112. ACM, 2014.

[Kre15] R. Krebbers. *The C standard formalized in Coq.* PhD thesis, Radboud University Nijmegen, 2015.

[KSW17] S. Krishna, D. Shasha, and T. Wies. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *Principles of Programming Languages (POPL)*, 2017.

[Küb18] J. Kübler. Comparing Deductive Program Verification of Graph Data-Structures. *Bachelor's thesis, KIT*, 2018.

[KV98] O. Kupferman and M. Vardi. Modular Model Checking. In W. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference (COMPOS)*, pages 381–401. Springer Berlin Heidelberg, 1998.

[Lam98] L. Lamport. The Part-time Parliament. *Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam13] P Lammich. Automatic Data Refinement. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP)*, pages 84–99. Springer, 2013.

[LBR99] G. Leavens, A. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Springer US, 1999.

[Lei98] K.R.M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 144–153. ACM, 1998.

[Lei10] K.R.M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In E. Clarke and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 348–370. Springer, 2010.

[Ler09] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM (CACM)*, 52(7):107–115, 2009.

[LGV16] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming Release-acquire Consistency. In *Principles of Programming Languages (POPL)*, pages 649–662. ACM, 2016.

[LKT+19] A. Lööw, R. Kumar, Y. Tan, M. Myreen, M. Norrish, O. Abrahamsson, and A. Fox. Verified Compilation on a Verified Processor. In *Programming Language Design and Implementation (PLDI)*, pages 1027–1039. ACM, 2019.

[LL77] G. Le Lann. Distributed Systems - Towards a Formal Approach. In *International Federation for Information Processing (IFIP)*, 1977.

[LLF+96] J. Lions, L. Luebeck, J. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5, Flight 501 Failure, Report by the Inquiry Board, 1996.

[LLP+11] A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. Multi-core Nested Depth-First Search. In T. Bultan and P. Hsiung, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 6996 of *LNCS*, pages 321–335. Springer, 2011.

[LMS09] K.R.M. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri,

editors, *Foundations of Security Analysis and Design (FOSAD)*, volume 5705 of *LNCS*, pages 195–222, 2009.

[LN15] P. Lammich and R. Neumann. A Framework for Verifying Depth-First Search Algorithms. In *Certified Programs and Proofs (CPP)*, pages 137–146. ACM, 2015.

[LOD+13] A. Laarman, M. Olesen, A. Dalsgaard, K. Larsen, and J. van de Pol. Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction. In N. Sharygina and H. Veith, editors, *Computer-Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 968–983. Springer, 2013.

[LPC+07] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, 2007. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[lps] Documentation of the `lpssymbolicbisim` tool. `https://www.mcrl2.org/web/user_manual/tools/experimental/lpssymbolicbisim.html` (accessed July 2019).

[LPW11] A. Laarman, J. van de Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods (NFM)*, pages 506–511. Springer Berlin Heidelberg, 2011.

[LQ08] S. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In *Principles of Programming Languages (POPL)*, pages 171–182. ACM, 2008.

[LT93] N. Leveson and C. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.

[LT12] P. Lammich and T. Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving (ITP)*, pages 166–182. Springer, 2012.

[LV15] O. Lahav and V. Vafeiadis. Owicki-Gries Reasoning for Weak Memory Models. In M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Automata, Languages and Programming (ICALP)*, pages 311–323. Springer Berlin Heidelberg, 2015.

[LVK+17] O. Lahav, V. Vafeiadis, J. Kang, C. Hur, and D. Dreyer. Repairing Sequential Consistency in C/C++11. In *Programming Language Design and Implementation (PLDI)*, pages 618–632. ACM, 2017.

[LW19] P. Lammich and S. Wimmer. IMP2 – Simple Program Verification in Isabelle/HOL. *Archive of Formal Proofs*, 2019. `http://isa-afp.org/entries/IMP2.html`, Formal proof development.

[LZ14] J. Lei and Q. Zongyan. Modular Reasoning for Message-Passing Programs. In G. Ciobanu and D. Méry, editors, *Theoretical Aspects of Computing (ICTAC)*, pages 277–294. Springer, 2014.

[LZS17] Z. Luo, M. Zheng, and S. Siegel. Verification of MPI programs using CIVL. In *EuroMPI*. ACM, 2017.

[MB08] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer Berlin Heidelberg, 2008.

[Mes] MPI: A Message-Passing Interface standard. `http://www.mpi-forum.org/docs` (accessed July 2019).

[Mey88] B. Meyer. Eiffel: A Language and Environment for Software Engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[MH81] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Webber and N. Nilsson, editors, *Readings in Artificial Intelligence*, pages 431–450. Kaufmann, M., 1981.

[Mil84] R. Milner. A Complete Inference System for a Class of Regular Behaviours. *Journal of Computer and System Sciences*, 28(3):439–466, 1984.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[MN95] O. Müller and T. Nipkow. Combining model checking and deduction for I/O- automata. In E. Brinksma, W. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 1–16. Springer Berlin Heidelberg, 1995.

[Mol90] F. Moller. The Importance of the Left Merge Operator in Process Algebras. In M. Paterson, editor, *Automata, Languages and Programming (ICALP)*, pages 752–764. Springer, 1990.

[Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38, 1965.

[MSC] mCRL2—Showcases. `https://www.mcrl2.org/web/user_manual/showcases.html` *(accessed on July 2019)*.

[MSS16] P. Müller, M. Schwerhoff, and A. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K.R.M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 41–62. Springer, 2016.

[Mun71] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[Nam01] K. Namjoshi. Certifying Model Checkers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 2–13. Springer, 2001.

[Neu14] R. Neumann. Using Promela in a Fully Verified Executable LTL Model Checker. In D. Giannakopoulou and D. Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 105–114. Springer, 2014.

[NLWSD14] A. Nanevski, R. Ley-Wild, I. Sergey, and G. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In Z. Shao, editor, *European Symposium on Programming (ESOP)*, pages 290–310. Springer Berlin Heidelberg, 2014.

[NTS] Landelijke Tunnelstandaard (National Tunnel Standard). `http://publicaties.minienm.nl/documenten/landelijke-tunnelstandaard` *(accessed on June 2019)*.

[NWG18] T. Neele, T. Willemse, and J.F. Groote. Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting. In K. Bae and P. Ölveczky, editors, *Formal Aspects of Component Software (FACS)*, pages 216–236. Springer, 2018.

[NWP02] T. Nipkow, M. Wenzel, and L. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[OBG+17] W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. An Abstraction Technique for Describing Concurrent Program Behaviour. In A. Paskevich and T. Wies, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 10712 of *LNCS*, pages 191–209, 2017.

[OBH16] W. Oortwijn, S. Blom, and M. Huisman. Future-based Static Analysis of Message Passing Programs. In *Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES)*, pages 65–72. Open Publishing Association, 2016.

[ODP16] W. Oortwijn, T. van Dijk, and J. van de Pol. A Distributed Hash Table for Shared Memory. In R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski, and K. Wiatr, editors, *Parallel Processing and Applied Mathematics (PPAM)*, pages 15–24. Springer, 2016.

[ODP17] W. Oortwijn, T. van Dijk, and J. van de Pol. Distributed Binary Decision Diagrams for Symbolic Reachability. In *SPIN*, pages 21–30. ACM, 2017.

[OG75] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*, 6:319–340, 1975.

[OGH20] W. Oortwijn, D. Gurov, and M. Huisman. Practical Abstractions for Automated Verification of Shared-Memory Concurrency. In D. Beyer and D. Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS. Springer, 2020. To appear.

[O'H07] P. O'Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[O'H19a] P. O'Hearn. Separation Logic. *Communications of the ACM (CACM)*, 62(2):86–95, 2019.

[OH19b] W. Oortwijn and M. Huisman. Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System. In W. Ahrendt and S. L. Tapia Tarifa, editors, *integrated Formal Methods (iFM)*, LNCS. Springer, 2019. To appear.

[OH19c] W. Oortwijn and M. Huisman. Practical Abstractions for Automated Verification of Message Passing Concurrency. In W. Ahrendt and S. L. Tapia Tarifa, editors, *integrated Formal Methods (iFM)*, LNCS. Springer, 2019. To appear.

[OHJP19] W. Oortwijn, M. Huisman, S. Joosten, and J. van de Pol. Automated Verification of Parallel Nested DFS. In *Submitted*, 2019.

[OP99] P. O'Hearn and D. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[O'R08] G. O'Regan. *A Brief History of Computing*. Springer Science & Business Media, 2008.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.

[ORY01] P. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[OW02] T. Ostrand and E. Weyuker. The Distribution of Faults in a Large Industrial Software System. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 55–64. ACM, 2002.

[OWB04] T. Ostrand, E. Weyuker, and R. Bell. Where the Bugs Are. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 86–96. ACM, 2004.

[OYR04] P. O'Hearn, H. Yang, and J. Reynolds. Separation and Information Hiding. In *Principles of Programming Languages (POPL)*, pages 268–280. ACM, 2004.

[Par10] M. Parkinson. The Next 700 Separation Logics. In G. Leavens, P. O'Hearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 169–182. Springer, 2010.

[Pep17] F. Peper. The End of Moore's Law: Opportunities for Natural Computing? *New Generation Computing*, 35(3):253–269, 2017.

[PGS01] D. Peled, D. Gries, and F. Schneider, editors. *Software Reliability Methods*. Springer-Verlag, 2001.

[PJP15] W. Penninckx, B. Jacobs, and F. Piessens. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In J. Vitek, editor, *European Symposium on Programming (ESOP)*, pages 158–182. Springer Berlin Heidelberg, 2015.

[PMP+14] P. Philippaerts, J. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software Verification with VeriFast: Industrial Case Studies. *Science of Computer Programming*, 82:77–97, 2014.

[PMP+16] O. Padon, K. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: Safety Verification by Interactive Generalization. In *Programming Language Design and Implementation (PLDI)*, pages 614–630. ACM, 2016.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Foundations of Computer Science (FOCS)*, pages 46–57. IEEE Computer Society, 1977.

[Pol15] J. van de Pol. Automated Verification of Nested DFS. In M. Núñez and M. Güdemann, editors, *Formal Methods for Industrial Critical*

Systems (FMICS), volume 9128 of *LNCS*, pages 181–197. Springer, 2015.

[PPE04] J. Pang, J. van de Pol, and M. Espada. Abstraction of Parallel Uniform Processes with Data. In *Software Engineering and Formal Methods (SEFM)*, pages 14–23. IEEE, 2004.

[PR12] C. Popeea and A. Rybalchenko. Compositional Termination Proofs for Multi-threaded Programs. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *LNCS*, pages 237–251. Springer, 2012.

[Pra95] V. Pratt. Anatomy of the Pentium Bug. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Theory and Practice of Software Development (TAPSOFT)*, pages 97–107. Springer, 1995.

[PS11] M. Parkinson and A. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. In G. Barthe, editor, *European Symposium on Programming (ESOP)*, pages 439–458. Springer Berlin Heidelberg, 2011.

[PSO18] D. Pym, J. Spring, and P. O'Hearn. Why Separation Logic Works. *Philosophy & Technology*, 2018.

[PTJ19] W. Penninckx, A. Timany, and B. Jacobs. Abstract I/O Specification. *ArXiv*, abs/1901.10541, 2019.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming (Programming)*, pages 337–351. Springer Berlin Heidelberg, 1982.

[RDKP17] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on Parallel Explicit Emptiness Checks for Generalized Büchi Automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 19(6):653–673, 2017.

[REB98] W. de Roever, K. Engelhardt, and K. Buth. *Data Refinement: Model-oriented Proof Methods and their Comparison*, volume 47. Cambridge University Press, 1998.

[Rei85] J. Reif. Depth-First Search is Inherently Sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[Rey00] J. Reynolds. Intuitionistic Reasoning about Shared Mutable Data Structure. *Millennial Perspectives in Computer Science*, 2(1):303–321, 2000.

[Rey02] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[RGNS16] E. Ruijters, D. Guck, M. van Noort, and M. Stoelinga. Reliability-centered maintenance of the Electrically Insulated Railway Joint via Fault Tree Analysis: A practical experience report. In *Dependable Systems and Networks (DSN)*, pages 662–669. IEEE Computer Society, 2016.

[RHH+01] W. de Roever, U. Hanneman, J. Hooiman, Y. Lakhneche, P. Mannes, J. Zwiers, and F. de Boer. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[RHVG16] A. Raad, A. Hobor, J. Villard, and P. Gardner. Verifying Concurrent Graph Algorithms. In A. Igarashi, editor, *Programming Languages and Systems (APLAS)*, pages 314–334. Springer International Publishing, 2016.

[RPDYG14] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In R. Jones, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 207–231. Springer, 2014.

[RPDYG15] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Steps in Modular Specifications for Concurrent Modules. In *Mathematical Foundations of Programming Semantics (MFPS)*, 2015.

[RPDYGS16] P. da Rocha Pinto, T. Dinsdale-Young, P. Gardner, and J. Sutherland. Modular Termination Verification for Non-blocking Concurrency. In P. Thiemann, editor, *European Symposium on Programming (ESOP)*, pages 176–201. Springer Berlin Heidelberg, 2016.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured Specifications and Interactive Proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications: Volume II: Systems and Implementation Techniques*, pages 13–39. Springer Netherlands, 1998.

[SAB+16] R. e Silva, N. Arai, L. Burgareli, J. de Oliveira, and J. Pinto. Formal Verification With Frama-C: A Case Study in the Space Software Domain. *IEEE Transactions on Reliability*, 65(3):1163–1179, 2016.

[SB14] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In Z. Shao, editor, *European Symposium on Programming (ESOP)*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.

[SBP13] K. Svendsen, L. Birkedal, and M. Parkinson. Modular Reasoning about Separation of Concurrent Data Structures. In M. Felleisen and P. Gardner, editors, *European Symposium on Programming (ESOP)*, pages 169–188. Springer, 2013.

[Sch16] M. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution.* PhD thesis, ETH Zürich, 2016.

[SE05] S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.

[Sha18] N. Shankar. Combining Model Checking and Deduction. In *Handbook of Model Checking*, pages 651–684. Springer, 2018.

[SJP09] J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In S. Drossopoulou, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 148–172. Springer, 2009.

[SJP12] J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames. *Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):1–58, 2012.

[SM16] A. Summers and P. Müller. Actor Services – Modular Verification of Message Passing Programs. In P. Thiemann, editor, *European Symposium on Programming (ESOP)*, pages 699–726. Springer, 2016.

[SM18] A. Summers and P. Müller. Automating Deductive Verification for Weak-Memory Programs. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 190–209. Springer, 2018.

[SNB15a] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized Verification of Fine-Grained Concurrent Programs. In *Programming Language Design and Implementation (PLDI)*, pages 77–87. ACM, 2015.

[SNB15b] I. Sergey, A. Nanevski, and A. Banerjee. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In J. Vitek, editor, *European Symposium on Programming (ESOP)*, volume 9032 of *LNCS*, pages 333–358. Springer, 2015.

[SOP12] R. e Silva, J. de Oliveira, and J. Pinto. A Case Study on Model Checking and Deductive Verification Techniques of Safety-Critical Software. In *Brazilian Symposium on Formal Methods (SBMF)*. Federal University of Campina Grande, 2012.

[Spr98] C. Sprenger. A Verified Model Checker for the Modal μ-calculus in Coq. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1384 of *LNCS*, pages 167–183. Springer, 1998.

[Sup] Supplementary material for this thesis. `https://github.com/wytseoortwijn/SupplementaryMaterialsThesis`.

[SWT17] I. Sergey, J. Wilcox, and Z. Tatlock. Programming and Proving with Distributed Protocols. *Principles of Programming Languages (POPL)*, 2, 2017.

[SZL+15] S. Siegel, M. Zheng, Z. Luo, T. Zirkel, A. Marianiello, J. Edenhofner, M. Dwyer, and M. Rogers. CIVL: The Concurrency Intermediate Verification Language. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 61. ACM, 2015.

[Tar71] R. Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 114–121. IEEE, 1971.

[TDB13] A. Turon, D. Dreyer, and L. Birkedal. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-Order Concurrency. In *International Conference on Functional Programming (ICFP)*, pages 377–390. ACM, 2013.

[Tec] The Technolution webpage. `https://www.technolution.eu` *(accessed on June 2019)*.

[TGSM19] A. Ter-Gabrielyan, A. Summers, and P. Müller. Modular Verification of Heap Reachability Properties in Separation Logic. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 3:1–28, 2019.

[Uri00] T. Uribe. Combinations of Model Checking and Theorem Proving. In H. Kirchner and C. Ringeissen, editors, *Frontiers of Combining Systems (FroCoS)*, pages 151–170. Springer, 2000.

[Use02] Y. Usenko. *Linearization in μCRL*. Technische Universiteit Eindhoven, 2002.

[Vaf10a] V. Vafeiadis. Automatically Proving Linearizability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer-Aided Verification (CAV)*, pages 450–464. Springer Berlin Heidelberg, 2010.

[Vaf10b] V. Vafeiadis. RGSep Action Inference. In G. Barthe and M. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 345–361. Springer Berlin Heidelberg, 2010.

[Vaf11] V. Vafeiadis. Concurrent separation logic and operational semantics. In *Mathematical Foundations of Programming Semantics (MFPS)*, volume 276 of *ENTCS*, pages 335–351, 2011.

[VBC+15] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *Principles of Programming Languages (POPL)*, pages 209–220. ACM, 2015.

[VLC09] J. Villard, É. Lozes, and C. Calcagno. Proving Copyless Message Passing. In Z. Hu, editor, *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 194–209. Springer, 2009.

[VP07] V. Vafeiadis and M. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In L. Caires and V. Vasconcelos, editors, *International Conference on Concurrency Theory (CONCUR)*, pages 256–271. Springer Berlin Heidelberg, 2007.

[VSC] The VerifiedSCION Project Page. `http://www.pm.inf.ethz.ch/research/verifiedscion.html` *(accessed on May 2019).*

[vVZN+11] J. Ševčik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory Concurrency and Verified Compilation. In *Principles of Programming Languages (POPL)*, pages 43–54. ACM, 2011.

[VW86] M. Vardi and P. Wolper. Automata-Theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Sciences (JCSS)*, 32(2):183–221, 1986.

[WCMH19] S. Wang, Q. Cao, A. Mohan, and A. Hobor. Certifying Graph-manipulating C Programs via Localizations Within Data Structures. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 3:1–30, 2019.

[Why] Why3 gallery of formally verified programs. `http://toccata.lri.fr/gallery/graph.en.html` *(accessed on July 2019).*

[Wig07] J.E. Wiggelinkhuizen. Feasibility of formal model checking in the Vitatron environment. Master's thesis, Eindhoven University of Technology, 2007.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems (AVMFSS)*, pages 68–80. Springer, 1989.

[WL18] S. Wimmer and P. Lammich. Verified Model Checking of Timed Automata. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10805 of *LNCS*, pages 61–78. Springer, 2018.

[XRH97] Q. Xu, W. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing (FAOC)*, 9(2):149–174, 1997.

[YB12] B. Yakobowski and R. Bonichon. Frama-C's Mthread plug-in. *Report, Software Reliability Laboratory*, 2012.

[ZN11] D. Zimmerman and R. Nagmoti. JMLUnit: The Next Generation. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 183–197. Springer Berlin Heidelberg, 2011.

[ZRL+15] M. Zheng, M. Rogers, Z. Luo, M. Dwyer, and S. Siegel. CIVL: Formal Verification of Parallel Programs. In *Automated Software Engineering (ASE)*, pages 830–835. IEEE, 2015.

[ZS15] M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs*. PhD thesis, University of Twente, 2015.

Samenvatting

Software is diep geïntegreerd in de moderne samenleving, niet alleen voor alledaags gemak en vermaak, maar ook voor taken waarbij veiligheid cruciaal is. Door de toenemende afhankelijkheid van software wordt het steeds belangrijker dat softwaresystemen zowel betrouwbaar als correct geïmplementeerd zijn met betrekking tot het beoogde gedrag. Het is echter zeer moeilijk om enige garanties te geven op de betrouwbaarheid en correctheid van software, omdat softwaresystemen door mensen worden ontwikkeld die inherent fouten maken. Het kunnen geven van zulke garanties wordt nog moeilijker gemaakt door de toenemende noodzaak om meerdere berekeningen tegelijk uit te kunnen voeren. Het tegelijk uitvoeren van berekeningen wordt *concurrency* genoemd en zorgt voor veel potentiële problemen. Het aantal verschillende manieren en volgordes waarop complexe berekeningen uitgevoerd kunnen worden is namelijk meestal astronomisch. Software-ontwikkelaars hebben daarom formele technieken en gereedschap nodig om te helpen in het begrijpen en bevatten van al het mogelijke systeemgedrag.

Dit proefschrift presenteert formele technieken die het mogelijk maken om concurrent systeemgedrag beter te begrijpen. We richten ons zich in het bijzonder op *deductieve verificatie*, een formele softwareverificatietechniek gebaseerd op wiskundige logica. In deductieve verificatie wordt het beoogde softwaregedrag gespecificeerd in een *programmalogica*. Deze specificaties worden vervolgens ingelezen door verificatiesystemen, die automatisch kunnen bevestigen dat een softwaresysteem inderdaad, in elk mogelijk scenario, het gespecificeerde gedrag toont.

In dit proefschrift richten we ons specifiek op *concurrent separatielogica (CSL)*, een gespecialiseerde programmalogica voor het beschrijven van, en redeneren over, concurrent programmagedrag. In de afgelopen jaren is er veel vooruitgang gemaakt in zowel de theorie van CSL-gebaseerde verificatietechnieken als automatische verificatiesystemen die deze technieken ondersteunen. Desalniettemin zijn er nog veel open uitdagingen. De open uitdaging die centraal staat in dit proefschrift is:

Hoe kan het functionele programmagedrag van concurrent software worden gespecificeerd en geverifieerd op een correcte, degelijke en praktische manier?

Dit proefschrift bestaat uit drie delen, die elk een ander aspect van de bovengenoemde uitdaging behandelen.

In Deel I (hoofdstukken 2–3) onderzoeken we hoe CSL gebruikt kan worden om de correctheid van parallele model-checking algoritmes mechanisch te verifiëren. Model-checking is een alternatieve softwareverificatietechniek die niet is gebaseerd op wiskundige logica, maar op het grondig algoritmisch doorzoeken van alle mogelijke systeemgedragingen op potentiële fouten. Deze algoritmes maken veelal gebruik van parallellisme om sneller door alle systeemgedragingen te kunnen zoeken, waardoor ze foutgevoeliger worden. Het is echter essentieel dat deze parallelle model-checking algoritmes zelf correct zijn om een bedrieglijk gevoel van veiligheid te voorkomen. Hoofdstuk 3 beschrijft de eerste mechanische verificatie van een parallel model-checking algoritme, dat *nested depth-first search (NDFS)* heet. Deze verificatie is uitgevoerd met behulp van VerCors: een automatisch verificatieprogramma dat CSL gebruikt als onderliggende logische basis. We beschrijven eveneens hoe dit mechanische bewijs gebruikt kan worden om eenvoudig de correctheid van verschillende optimalisaties van parallel NDFS te bevestigen.

Deel II (hoofdstukken 4–6) draagt een praktische abstractietechniek bij voor het verifiëren van globale gedragseigenschappen van concurrent softwaresystemen waarbij de verschillende subsystemen communiceren via een gedeeld geheugen. Onze abstractietechniek is ontwikkeld op basis van het inzicht dat concurrent programmagedrag moeilijk gespecificeerd kan worden op het niveau van programmacode. Dit is hoofdzakelijk omdat realistische programmeertalen weinig algebraïsche eigenschappen hebben, bijvoorbeeld vanwege geavanceerde taakconstructies om concurrency te kunnen implementeren. Onze abstractietechniek maakt het echter mogelijk om het programmagedrag te beschrijven als een wiskundig model met een elegante algebraïsche structuur. We gebruiken hiervoor *proces algebra* als modelleertaal, waarbij zogeheten *acties* gebruikt worden voor het abstraheren van schrijfinstructies voor het aanpassen van gedeeld geheugen. Daarnaast breiden we CSL uit met logische primitieven die het mogelijk maken te bewijzen dat een programma een proces-algebraïsch model verfijnd. Onze abstractietechniek is correct bewezen met behulp van de bewijsassistent Coq en is geïmplementeerd in VerCors. We demonstrereren onze abstractietechniek op verschillende verificatievoorbeelden, waaronder een klassiek 'leader election' protocol, alsmede een industriële case study betreffende de formele verificatie van een centraal besturingscomponent van een Nederlandse verkeerstunnel.

Deel III (hoofdstukken 7–8) onderzoekt hoe onze abstractietechniek kan worden aangepast voor het verifiëren van gedistribueerde systemen, waarbij verschillende subsystemen communiceren door middel van het uitwisselen van berichten. Deze

aanpassing gebruikt proces algebra voor het abstraheren van berichtuitwisselingen, in plaats van aanpassingen aan een gedeeld geheugen. Daarnaast onderzoeken we hoe deductieve verificatie gecombineerd kan worden met model-checking. De motivatie hiervoor is dat deductieve verificatie en model-checking complementaire technieken zijn. Deductieve verificatie is gespecialiseerd in het verifiëren van datagedreven gedrag, terwijl model-checking gespecialiseerd is in het verifiëren van controle-georiënteerd gedrag. Deze combinatie is gunstig voor het redeneren over gedistribueerde systemen, omdat ze zowel computaties als communicatie uitoefenen. Onze abstractietechniek is correct bewezen met behulp van Coq en is geïmplementeerd als een handmatige codering in Viper.

Alles samengenomen maakt dit proefschrift een stap voorwaarts richting *praktische, expressieve en betrouwbare* verificatie van globale gedragseigenschappen van concurrent en gedistribueerde software. De technieken in dit proefschrift zijn *betrouwbaar* omdat ze ondersteund worden door mechanische correctheidsbewijzen in Coq; zijn *expressief* omdat ze compositioneel zijn en voortbouwen op wiskundige modellen met een gunstige algebraïsche structuur; en zijn *praktisch* omdat ze ondersteund worden door automatische verificatieprogramma's.

Titles in the IPA Dissertation Series since 2016

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via*

Model Checking. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science,

Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems.* Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming.* Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi.* Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems.* Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces.* Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network.* Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems.* Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations.* Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries.* Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-*

Specific Languages. Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow.* Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning.* Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management.* Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers.* Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems.* Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning.* Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities.* Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis.* Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing.* Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

This thesis contributes formal techniques for verifying global behavioural properties of real-world concurrent software in a sound and practical manner.

The first part of this thesis discusses how Concurrent Separation Logic (CSL) can be used to mechanically verify the parallel nested depth-first search (NDFS) model checking algorithm. This verification has been performed using VerCors. We also demonstrate how our mechanised correctness proof allows verifying various optimisations of parallel NDFS with only little extra effort.

The second part contributes an abstraction technique for verifying global behavioural properties of shared-memory concurrent software. This abstraction technique allows specifying program behaviour as a process-algebraic model, with an elegant algebraic structure. Furthermore, we extend CSL with logical primitives that allow one to prove that a program refines its process-algebraic specification. This abstraction technique is proven sound using Coq and is implemented in VerCors. We demonstrate our approach on various examples, including a real-world case study from industry that concerns safety-critical code.

In part three, we lift our abstraction technique to the distributed case, by adapting it for verifying message passing concurrent software. This adaption uses process-algebraic specifications to abstract the communication behaviour of distributed agents. We also investigate how model checking of these specifications can soundly be combined with the deductive verification of the specified program.