Few is Just Enough!

Small Model Theorem for Parameterized Verification and Shape Analysis

FRÉDÉRIC HAZIZA
Dissertation presented at Uppsala University to be publicly examined in ITC 2446, Lägerhyddsvägen 2, Uppsala, Wednesday, 18 November 2015 at 13:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Prof. Ing. Tomáš Vojnar.

Abstract

This doctoral thesis considers the automatic verification of parameterized systems, i.e. systems with an arbitrary number of communicating components, such as mutual exclusion protocols, cache coherence protocols or heap manipulating programs. The components may be organized in various topologies such as words, multisets, rings, or trees.

The task is to show correctness regardless of the size of the system and we consider two methods to prove safety: (i) a backward reachability analysis, using the well-quasi ordered framework and monotonic abstraction, and (ii) a forward analysis which only needs to inspect a small number of components in order to show correctness of the whole system. The latter relies on an abstraction function that views the system from the perspective of a fixed number of components. The abstraction is used during the verification procedure in order to dynamically detect cut-off points beyond which the search of the state-space need not continue.

Our experimentation on a variety of benchmarks demonstrate that the method is highly efficient and that it works well even for classes of systems with undecidable property. It has been, for example, successfully applied to verify a fine-grained model of Szymanski’s mutual exclusion protocol. Finally, we applied the methods to solve the complex problem of verifying highly concurrent data-structures, in a challenging setting: We do not a priori bound the number of threads, the size of the data-structure, the domain of the data to store nor do we require the presence of a garbage collector. We successfully verified the concurrent Treiber’s stack and Michael & Scott’s queue, in the aforementioned setting.

To the best of our knowledge, these verification problems have been considered challenging in the parameterized verification community and could not be carried out automatically by other existing methods.

Keywords: program verification, model checking, parameterized systems, infinite-state systems, reachability, approximation, safety, tree systems, shape analysis, small model properties, view abstraction, monotonic abstraction

Frédéric Haziza, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Frédéric Haziza 2015

ISSN 1651-6214
ISBN 978-91-554-9366-0
urn:nbn:se:uu:diva-264171 (http://urn.kb.se/resolve?urn=nbn:se:uu:diva-264171)
Pour Papa, Maman, Franck et Alexandre
List of papers

This thesis is based on the following papers, which are referred in the text by their roman numerals.

I  All for the Price of Few
Parosh A. Abdulla, Frédéric Haziza, and Lukás Holík.
In *Verification, Model Checking, and Abstract Interpretation*, 2013.

II An Integrated Specification and Verification Technique for Highly Concurrent Data Structures
Parosh A. Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine.

III Block Me If You Can! (Context-Sensitive Parameterized Verification)
Parosh A. Abdulla, Frédéric Haziza, and Lukás Holík.
In *Static Analysis Symposium*, 2014.

IV Monotonic Abstraction for Programs with Dynamic Memory Heaps
Parosh A. Abdulla, Ahmed Bouajjani, Jonathan Cederberg, Frédéric Haziza, Ahmed Rezine.

V Parameterized Tree Systems
Parosh A. Abdulla, Noomene Ben Henda, Giorgio Delzanno, Frédéric Haziza, Ahmed Rezine.

VI Model Checking Race-Freeness
Parosh A. Abdulla, Frédéric Haziza, and Mats Kindahl.

VII Parameterized Systems through View Abstraction
Parosh A. Abdulla, Frédéric Haziza, and Lukás Holík.

Reprints were made with permission from the publishers.
Acknowledgements

I have only two words to describe how I feel: **Tack Parosh!**

Turn the page to see what I mean.
FINALLY! It has been long and even tough at times, but I made it! We all know that I would not be writing this section if Parosh had not been there. I hope he understands how grateful I am to count him as my supervisor. But let me save the best for last and let me start by thanking everyone else who contributed to my journey that leads to this thesis. I must of course start with my favorite czech companion Lukáš Holík. Lukáš is a great researcher with tremendous motivation and skills. He was patient to listen to my ideas, which we both know must first make their way through a thick layer of verbose ramblings. It is nevertheless a great pleasure to work with him.

I enjoyed extremely much working with Bengt Jonsson, and I wish we had more projects in common. He made me quickly feel at ease and confident, even though I felt intimidated at first. This was really impressive to witness such a great mind at work. I would like to thank Noomene Ben Henda, Ahmed Rezine, Lisa Kaati, Mayank Sanjana, Johan Deneux, Sven Sandberg during my early days as a PhD student and lately Jonathan Cederberg, Othmane Rezine, Faouzi Atig, Trinh Cong Quy, Carl Leonardsson, Jari Stenman, Yunyun Zhu and Muneeb Khan. From outside the IT department of Uppsala University, I would like to mention Tomas Vojnar, Yufang Chen and Giorgio Delzanno who showed me how research works in other universities and how lucky I was at Uppsala University with Parosh. From Uppsala University, my deepest thank you goes to Ivan Christoff. Ivan always have our well-being at heart. It is a truly remarkable quality that leaves me with a profound gratitude. I would like to thank Mats Daniels, Wang Yi, Joachim Parrow, Ulrika Jaresund, Ulrika Andersson, Mattias Wigberg, Karl Marklund, to always have the time to lend me a hand or a complicit hear. A special note goes to my previous supervisors Björn Victor and Erik Hagersten who introduced me to the PhD program.
I developed a special profile as a PhD student, since I was given the chance to be the main teacher of several courses. Thank you Mats for trusting me with it. This was way more fun and challenging than a teaching assistant position. Along with teaching skills, I also had the opportunity to learn a lot culturally about Sweden. This is mainly due to my participation in local dance schools. First as a student (thank you Johanna Berglund to drag me into that world) and then as a teacher. Yes, you read it right, I am a dancing nerd. So it is now time to mention everyone else who contributed to my special time in Sweden. This time is probably not over but I still want to thank them. So here I shoot. In Uppsala, thank you Junior, Marina, Fabien, Chris, Sepideh, Paul, Camilla, Paola, Lilly, Lina, Carlos, Alexander K, Patrick, Alexander H, Alexander Ki, Linda, Yuri, Lisa K, Tanja, Anna G, Lisa-Marie, Ronak, Sasha, Amanda, Annika, Becky, Eva, Carmen, Eva-Lena, Wendelin, Isabel, John, Maele, Maggie, Maria, Malou, Rebecca, Tim, Anna C, Erik I, Matild, Tove, Mia, Erika and Chiqui. Special thanks to my tennis partners Nadim and Sebastian. In Stockholm, thank you Kristofer, Marina, Alexander, Adela, Mina, Andreas, Behnaz, Egle, Camilla C, Eshtar, Hind, Jaime, Jennifer, Jo- cal, Tofik, Lalla, Pas- hanna E, and every- one else that I will soon meet. And of course, let’s not forget Lisa and Liza. So, finally, we come to the main acknowledgment. This thesis starts with Parosh on my side and I would like to finish that section with Parosh A. Abdulla. This has not been easy and despite my stubbornness, he was willing to put efforts into helping me. He backed me up at difficult times and trusted me at some other times. He gave me a chance to travel, trusted me with conference presentations and showed me that I maybe know more than it looked like. It is almost too good to be true. Thank you Parosh.


**Säkerhet: Givet en specifikation, kan systemet hamna i en fel konfiguration?**


Många system med ett oändligt antal tillstånd kan faktiskt karakteriseras av en familj som består av system med ett ändligt antal tillstånd, och en parameter
(eller flera) med värde från en obegränsad domän. För varje värde på parame-
tern, innehåller systemet ett ändligt antal tillstånd. Parametern kan exempelvis
vara antalet processer som är aktiva i en viss session av ett protokoll, antalet
noder i ett nät, eller hur komponenterna av ett program kommunikerar med
varandra. Hur som helst, system som innehåller en preliminärt okänd param-
eter bör ha ett korrekt beteende oavsett värdet på parametern. De betraktas
därför som system med ett oändligt antal tillstånd och kallas för parametris-
erade system. I denna avhandling presenterar vi två metoder för att verifiera
vissa säkerhetsegenskaper hos sådana parametriserade system.

Den första metoden körs baklänges. Den startar från dem felaktiga tillstån-
den, och beräknar vilka andra tillstånd som skulle kunna leda till ett fel. Med
andra ord upptäcker metoden alla konfigurationer som direkt eller indirekt är
felaktiga. Om initialtillstånden av programmet inte tillhör dem sistnämnda,
anses programmet vara korrekt. Vi måste såklart först se till att approximatio-
nen av modellen motsvarar det ursprungliga programmet.

Den andra metoden startar från initialtillstånden. Den begränsar sig till små
värden av parametern, och härleder ett tröskelvärde, efter vilket metoden inte
behöver fortsätta: den har faktiskt all nödvändig information för att dra slutsat-
sen att det inte finns några felkonfigurationer för större värden på parametern
över denna tröskel. För att förenkla, bryter metoden ner varje konfiguration
i små bitar av en viss storlek, och rekombinerar bitarna på alla möjliga sätt,
just för att skapa konfigurationer av större storlek (litegrann som Legobitar).
Tanken är att samla alla dem små bitarna och se till att ingen rekombination
matchar någon felaktig konfiguration. I så fall är tröskeln hittad. Annars börjar
man om med lite större bitar.

Slutligen är det intressant att undra vilka av villkoren som behövs för att
metodens beräkningar inte ska fortsätta för evigt. Problemet klassas som
oavgörbart, det vill säga att det inte finns någon generell metod som kan lösa
alla instanser av problemet. Däremot har dessa två metoder trots allt visat sig
vara väldigt effektiva. Där andra metoder tog timmar, kunde dessa två metoder
faktiskt verifiera vissa program inom sekunder, vilket var målet. I vissa fall
kan man även ge garanti på att antalet beräkningar är begränsad.
Résumé en Français

Pour vous simplifier beaucoup de tâches longues et ardues, vous avez certainement déjà utilisé un ordinateur et en étiez très satisfaits. Cependant, de temps à autre, l’ordinateur ne fait pas ce que vous lui demandez: vous cliquez sur le bouton “Faire ça” et il ne le fait pas. Ou pire, l’ordinateur se bloque et ne répond plus à aucune commande. D’importantes heures de travail viennent peut-être de s’envoler. Vous gardez votre sang-froid, redémarrer l’ordinateur et tout semble à nouveau marcher comme prévu. L’erreur ne vient visiblement pas de l’ordinateur lui-même, mais elle semble venir du logiciel qui contrôle ce dernier. Vous vous demandez pourquoi cette erreur n’a pas été corrigée, et qui plus est, pourquoi on ne s’est pas assuré dès le départ que l’erreur ne survient pas.

Les différentes composantes électriques de la machine peuvent tomber en panne. Historiquement, ces erreurs sont appelées des bugs – le terme anglais pour cafards – car il est dit qu’un vrai cafard est entré un soir dans le châssis d’une machine qu’un informaticien fabriquait dans son garage, détruisant au passage quelques morceaux du circuit électrique. En se réveillant le lendemain, il s’est rendu compte que la machine manifestait un comportement bizarre. Le terme est resté et est maintenant utilisé pour qualifier toutes sortes d’erreurs de programmation en général.

Étant donné la complexité des ordinateurs de nos jours, il n’est pas étonnant qu’ils soient sujets à de nombreuses erreurs. Ils nécessitent de prendre en charge une multitude de paramètres, sans oublier les erreurs humaines introduites lors de la phase de programmation. C’est pourquoi la tendance est de construire ces machines en composant plusieurs unités plus réduites. Chaque unité est de ce fait plus facile à contrôler, mais ces unités peuvent communiquer les unes avec les autres à n’importe quel moment. Il devient alors très difficile de prendre en compte tous les scénarios possibles et de prédire le comportement général du programme, à cause de ce caractère imprévisible.

Les entreprises fabriquant ces logiciels n’ont pas intérêt à y laisser des bugs, puisqu’une erreur peut engendrer d’autres erreurs en cascade. Celles-ci passent du temps, et de l’argent, à traquer ces bugs et s’il y en a trop, il n’est simplement plus rentable ou même envisageable de les éliminer. Les entreprises mettent donc en place, dans le cycle de développement de chaque projet, une phase essentielle de contrôle de qualité. Cela dit, certains bugs sont plus urgents à résoudre que d’autres. Ce n’est pas très grave si on ne peut pas décrocher son téléphone lors d’un appel, ou si l’éditeur de texte perd les derniers ajouts à
notre document. C’est peut-être très ennuyeux mais le pronostic vital n’est pas engagé! On s’en remettra, en attendant la mise à jour du logiciel incriminé. Par contre, ce n’est pas le cas pour ces systèmes, dits *critiques*, pour lesquels la sûreté est primordiale. Toutes les erreurs doivent être éliminées, qu’elles soient au niveau logiciel ou au niveau de la machine. Il n’est pas acceptable, par exemple, qu’un pacemaker s’arrête de fonctionner quand on passe le portique dans le métro, que l’avion parte en chute libre quand l’équipage allume le signal d’interdiction de fumer à bord, ou encore que deux trains entrent en collision parce que les feux ont mal fonctionné. Il est nécessaire de concevoir des techniques pour détecter ce genre d’erreurs.

Pour améliorer la qualité des logiciels, il existe une technique prédominante: la méthode qui vise à soumettre le programme à une série de tests et d’en observer les résultats. Ces scénarios de tests sont soigneusement conçus afin de couvrir un maximum d’exécution possible du programme. Dans la même catégorie, il est possible d’extraire un *modèle* du programme et de l’utiliser pour simuler le programme. Cela dit, lorsque le programme est de plus en plus grand et complexe, que l’on utilise un modèle ou le programme lui-même, il devient impossible de garantir que toutes les exécutions soient prises en compte par une batterie de scénarios. C’est même tout bonnement impossible lorsque le programme contient un paramètre qui varie sur un domaine non borné, comme par exemple, un entier. Ces deux méthodes sont utiles pour découvrir rapidement de simples erreurs, mais les erreurs subtiles, comme celles concernant le timing, restent non détectées. Dans le cas des systèmes qui requièrent une sûreté maximale, il n’est pas concevable d’utiliser un programme qui pourrait contenir une erreur, quand bien même il ait passé tous ses tests.

Comment donc s’assurer que toutes les exécutions d’un programme aient été prises en compte? On ne peut certainement pas faire tourner le programme et se contenter d’un ensemble restreint d’exécutions ou de valeurs pour les paramètres du programme. La technique d’analyse statique offre une couverture complète d’un programme, sans l’exécuter. Elle inspecte toutes les exécutions *faisables*, c’est-à-dire celles que le programme peut effectuer et non celles qu’il va effectuer. Le code source du programme y est sous la loupe, qu’il soit lisible par un humain ou seulement par une machine. Chaque combinaison est prise en compte ce qui rend cette méthode rapidement ingérable. Cette thèse se concentre autour du problème suivant: concevoir une méthode qui garantisse qu’aucune erreur ne reste inaperçue, tout en restant dans des proportions raisonnables.

Pour pouvoir vérifier qu’un programme soit correct, il s’agit d’abord de définir ce à quoi il doit se conformer —appelé formellement sa spécification. En listant l’ensemble des configurations à éviter, en listant tous les comportements souhaitables (ou une combinaison des deux), on décrit les propriétés que le programme doit satisfaire. Il en existe deux catégories: les propriétés de sûreté et les propriétés de vivacité (Safety et Liveness en anglais). Par exemple, “le pacemaker ne s’arrête jamais”, “l’airbag ne doit pas prendre plus
de $x$ millisecondes pour s’ouvrir”, ou encore “aucun processus ne peut bloquer tous les autres” sont des propriétés de sûreté. On doit s’assurer que le programme ne soit jamais dans une mauvaise configuration. À l’inverse, “le facteur livre le courrier à son destinataire”, “le système ne stagne pas” ou encore “le serveur prend en charge une requête internet” sont des propriétés de vivacité, où l’on s’intéresse aux bonnes configurations du programme, associées à certaines probabilités. C’est à nous de définir ce que la spécification doit inclure, pour s’accorder avec la vérification souhaitée. Un système d’antiblocage de freins (ABS), par exemple, calcule le freinage adéquate. Toutefois, si ce calcul ne se fait pas dans le temps imparti, le système n’est pas considéré comme un comportement désirable, et est donc défectueux. Dans cette thèse, nous nous concentrons sur les propriétés de sûreté.

**Sûreté:** Étant donnée une spécification, le système peut-il se retrouver dans une mauvaise configuration?

Plutôt que de tester le programme dans des scénarios particuliers, ou d’en analyser le code source, on utilise un cadre mathématique, dit de vérification formelle, qui permette de prouver qu’un programme soit conforme à sa spécification. Plus particulièrement, on cherche à prouver l’absence d’erreur, de manière automatique, c’est-à-dire sans intervention de l’utilisateur. On commence par extraire un modèle qui correspond aux comportements du programme originel, tout en en éliminant les parties sans rapport au vu de la propriété à vérifier. Toutefois, comment faire lorsque le programme manipule des entités non bornées? On parle alors de système à états infinis et on en construit une approximation qui respecte malgré tout l’essence même du programme.

De nombreux systèmes à états infinis peuvent en fait être caractérisés par une famille de systèmes à états finis, avec un paramètre (ou plus) variant dans un domaine non borné. Pour chaque valeur du paramètre, le système est à états finis. Le paramètre peut par exemple être le nombre de processus associés à une session donnée d’un protocole, le nombre de nœuds dans les mailles d’un réseau, ou encore l’agencement des différentes composantes d’un programme (et implicitement comment elles communiquent entre elles). Les systèmes qui contiennent un paramètre a priori inconnu doivent se comporter correctement quelle que soit la valeur du paramètre. Ils sont ainsi considérés comme des systèmes à états infinis, et on les appelle des systèmes paramétrés. Dans cette thèse, nous présentons deux méthodes pour vérifier certaines propriétés de sûreté de ces systèmes paramétrés.

La première méthode fait machine arrière: elle part des états erronés du système en général, et calcule à quoi ressembleraient les états du système qui pourraient mener à une erreur. Autrement dit, elle trouve l’ensemble des configurations qui sont directement et indirectement mauvaises. Si les états initiaux du système originel ne font pas partie de cet ensemble, le programme
peut être considéré comme correct. Il faut bien sûr d’abord s’assurer que l’approximation du modèle extraite du programme de départ corresponde à ce dernier.

La deuxième méthode démarre des états initiaux. Elle se limite à de petites valeurs du paramètre, et en déduit un seuil pour lequel il n’est pas nécessaire de continuer les calculs: la méthode a en effet toutes les données nécessaires pour conclure qu’il n’existe pas de mauvaises configurations pour les valeurs du paramètre plus grandes que ce seuil. En simplifiant, la méthode décompose chaque configuration en petits morceaux d’une certaine taille, et se charge de recombiner ces morceaux de toutes les manières possibles, en particulier en configurations de toutes tailles (un peu comme des briques de Lego). L’idée est de collecter tous les petits morceaux, et de s’assurer qu’aucune des recombinations ne correspond à une mauvaise configuration. Si c’est le cas, le seuil est trouvé. Si ce n’est pas le cas, on recommence avec des morceaux de taille un peu plus grande.

En dernier lieu, il est intéressant de se demander dans quelles conditions la méthode termine. Le problème est classé dans la catégorie des problèmes indécidables, c’est-à-dire qu’il n’y a pas de méthode générale pour solutionner toutes les instances du problème. Cela dit, pour certaines instances, ces deux méthodes s’avèrent être efficaces et peuvent même parfois être garanties de terminer. En effet, alors que d’autres méthodes prenaient des heures, ces méthodes permettent de vérifier certains programmes en quelques secondes, ce qui était le but recherché. Cette thèse démontre notamment l’efficacité de ces méthodes en s’attaquant au problème complexe d’analyse de formes, c’est-à-dire aux programmes concurrents manipulant des listes liées.
## Contents

List of papers .................................................................................................................. v

Acknowledgements (= Thank you, Parosh!) ................................................................. vii

Summary ........................................................................................................................... xi
  in Swedish ....................................................................................................................... xi
  in French ......................................................................................................................... xv

How to read this thesis ................................................................................................. xxi

1 The Field, the Area, the Problem ................................................................................ 23
  1.1 Formal Verification .................................................................................................. 25
  1.2 Model Checking ...................................................................................................... 26
  1.3 Parameterized Verification .................................................................................... 30
  1.4 Theoretical Limitations … not! ............................................................................ 32

2 Parameterized Systems .............................................................................................. 35
  2.1 Features .................................................................................................................. 35
  2.2 Formal Definition ................................................................................................... 37
  2.3 Non-Atomic Global Conditions .......................................................................... 38
  2.4 The Reachability Problem .................................................................................... 39
  2.5 A Challenging Example ....................................................................................... 40

3 Monotonic Abstraction ............................................................................................... 45
  3.1 Upward-Closed Sets ............................................................................................ 45
  3.2 Monotonicity .......................................................................................................... 46
  3.3 Backward Reachability ........................................................................................ 48
  3.4 Procedure and Termination ................................................................................ 49
  3.5 Applications .......................................................................................................... 51

4 View Abstraction ........................................................................................................ 59
  4.1 Abstract Domain ................................................................................................... 61
  4.2 Method .................................................................................................................... 62
  4.3 Contexts .................................................................................................................. 66
  4.4 No Atomicity .......................................................................................................... 71

5 Shape Analysis ........................................................................................................... 77
  5.1 Program Model ....................................................................................................... 77
5.2 Specification for a Concurrent Data Structure ....................................... 80
5.3 Data Independence ................................................................................... 82
5.4 Observers .................................................................................................. 83
5.5 Bounded Abstract Domain | No Garbage Collection ............................... 84
5.6 Verification procedure ............................................................................. 87

6 Case Studies ................................................................................................. 93
6.1 Burns’ mutual exclusion .......................................................................... 93
6.2 Szymanski’s mutual exclusion ................................................................. 93
6.3 Dijsktra’s mutual exclusion ..................................................................... 94
6.4 Gribomont-Zenner’s mutual exclusion .................................................... 95
6.5 Parosh’s mutual exclusion ....................................................................... 95
6.6 Bakery mutual exclusion ......................................................................... 96
6.7 MOSI Cache Coherence Protocol ............................................................ 96
6.8 German Cache Coherence Protocol .......................................................... 97
6.9 Tree Protocols .......................................................................................... 100
6.10 Agreement protocol on a Ring ................................................................. 104
6.11 Critical section guarded by a lock ............................................................ 106
6.12 Light Control ............................................................................................ 106
6.13 Priority Allocator ..................................................................................... 107
6.14 Simple Barrier ......................................................................................... 108
6.15 List with Counter Automata ..................................................................... 109
6.16 Michael & Scott’s Lock-Free Queue ....................................................... 109
6.17 Treiber’s Lock-Free Stack ....................................................................... 110

My Contributions .............................................................................................. 111

Conclusion and Future Work ............................................................................. 113

References ....................................................................................................... 117

Index ................................................................................................................. 121
How to read this thesis

This document is a comprehensive summary.

We first describe the domain of formal verification, where this thesis belongs, and approach the problem in a top down fashion. Chapter 1 narrows down the problem to parameterized systems and Chapter 2 reveals the main characteristics of the challenge. We present two techniques in Chapter 3 and 4 to prove safety properties for a wide range of programs (listed in Chapter 6). Finally, we dedicate Chapter 5 to the problem of shape analysis. It deals with a class of programs that manipulate memory heaps concurrently. To finish, we give some conclusions and potential directions for future research topics.
1. The Field, the Area, the Problem

You have surely used a computer and were very happy that it mechanically simplified many of the tasks (if not all) that would otherwise be slow and tedious to complete. But occasionally, it does not exactly behave as expected: You click on a button that says “do that” and it doesn’t do it, or so it seems. Even worse, the computer freezes and becomes completely unresponsive. You may have lost some important work. You hold your composure and restart the machine. It then appears to be all fine again. The error is obviously not caused by the machine itself, but it seems to originate from the program that controls the machine. You wonder why they didn’t fix the problem and, furthermore, that they should ensure that there is no problem.

The various electrical components of the machine (the hardware) can fail. Those errors are historically called bugs, as it is said that a real insect entered the compound of a machine one night and destroyed some part of the circuit board. The designers woke up the next day and noticed that the machine manifested an unexpected behaviour. The term stayed and is now also coined when the error is logical, that is, when the program (the software) is not a correct solution to the intended problem.

Given the complexity of today’s programs, it is not surprising to find them error-prone. They take unknown parameters as input and involve many features, which makes them difficult to design. Moreover, there is a human factor not to neglect in the design phase. The trend is now to build programs by composing many independent units that can interact with each other with an unpredictable timing. The latter aspect causes the complexity to increase even further. It can even happen that a program behaves as expected on some machine, but incorrectly on another machine, because the characteristics of the machines are different. We can easily observe this trend in a few examples. Considering our modern mobile phones and their multiple features, it is hard to find one that only makes and receives calls! The newest cars contain more than one hundred processors. Desktop machines allow for example to read an article online, while listening to music or checking emails in the background.

It is not in the interest of software companies to leave bugs undetected in any project they spawn, as those might in fact lead to subsequent bugs and a cascade of other problems. After some point, it becomes too costly to solve them or simply too daunting to attempt to. Therefore, companies include a phase of quality assurance in each project. Even with talented programmers on board earlier project phases, the quality phase takes a substantial amount of resources, both in terms of time and manpower.
Some bugs are less important to solve than others, though. It is of little relevance to our well-being, when we can’t pick up a phone call or when our document gets lost when the word processor crashes. While it is annoying, we usually manage to live with these occasional errors and wait for the next software release, hoping that it would solve the problem. However, in the case of critical systems, safety is of utmost importance and we cannot afford any bugs in either the software nor the hardware. We wouldn’t want a pacemaker to stop working every time the neighbor receives a phone call, the plane to start a free fall when the captain turns off the seatbelt sign or two trains to collide because the signaling system was malfunctioning! It is therefore primordial to devise techniques that discover bugs. It is perhaps even possible to recover from them or avoid them in the first place. Restrictions can be imposed on the programming language, which make programs avoid certain bugs by construction, but that would not eliminate the human factor. Systems can also be designed to be fault-tolerant, i.e. when facing an error, they have the ability to recognize it and recover from it. These domains are beyond this thesis.

The predominant method to improve software quality is testing. It is a dynamic analysis which consists in running a program under specific conditions, so-called test cases, and checking whether the result with a given input matches the expected output. The test cases are carefully crafted scenarios aiming at covering a maximum number of program executions. Similarly, a model of the program can be extracted, while removing all parts that are irrelevant for the tests, and can be used to simulate the executions. Simulation can be advantageously applied prior to the concrete build phase of the program. However, unless all possible input combinations are taken into account, there is no guarantee to cover all possible executions. This is in fact seldom the case and already impossible if any input parameter ranges over an unbounded domain as, say, integers or strings. Both techniques are useful to quickly discover obvious bugs but subtle errors, such as timing-related ones, may remain undetected. In the case of safety-critical systems, it is not satisfactory to have a program that may still contain errors after the test/simulation phase.

How can we achieve a full coverage of program executions? We can surely not run the program and content ourselves with a subset of all possible executions or only some values for the input parameters. One technique that offers full coverage is static analysis, where the code of the program is under scrutiny, in either a human-readable form or a machine-readable form. This technique provides a way, without executing the program, to inspect every feasible execution, i.e. one that the program could perform, as opposed to will perform. Naively taking into account every detail of the whole program, at the cost of potentially considering parts of the code that will never be executed, often turns out to be too heavy for the method. Slicing techniques must be used, i.e. cutting out program statements to locate source of errors more easily. However, finding out what could be ignored is certainly as hard as finding out what is important to keep.
To ascertain a complete coverage of program behaviours, any method will have the task to fully explore the state-space, i.e. to inspect a collection of all snapshots of the system for every step of the program. When the domain of the input parameters is large or if the program is complex, the task becomes clearly intractable. Such methods are then said to suffer from the state-space explosion problem.

The topic of this thesis is centered around the following question: Can we design a method that would guarantee that no error is left undetected and that does not suffer from state-space explosion?

1.1 Formal Verification

Formal verification is a subfield of software engineering where the goal is to ensure that the software meets the predefined requirements. There is a wide variety of properties that a program could try to satisfy, and they fall into two categories: safety and liveness properties.

Liveness properties state that some good behaviour eventually happens, like "the postman delivers the letter to the recipient", "the system makes progress" or "the web server handles a request". Specifying liveness properties requires to describe traces of events, often involving temporal logics, statistics, and probabilities. Checking a liveness property amounts to the repeated reachability of good situations.

On the contrary, ensuring that a safety property is satisfied boils down to checking that the system never encounters a bad situation [48]. It is enough to inspect every state, i.e. the result following every move of the system, without remembering how the system got there.

A specification is a formal description of the properties that a program should satisfy. We can specify programs by describing the situations to avoid, all safe behaviors or a combination of both. In this thesis though, we only encounter programs where the specification describes the bad behaviors. The latter are often easier to list and describe in a concise manner. For example, it can state that "no process blocks all the others", "the pacemaker never stops", "the music player is not too loud", "the airbag should not take more than x milliseconds to open", or finally that, a stack must behave as a "data-structure ensuring that the latest data to enter is the first to exit, without losing, creating nor duplicating data". Note that specifications can be partial or loosely defined. It is up to the designers to make them sufficiently precise for the verification problem at hand. If an Anti-lock Braking System (ABS), for example, computes the proper amount of braking to apply on the wheels but sometimes too late, it is not considered a desired behavior and therefore not a functioning system.

This thesis will concentrate on safety properties and revolve around the following statement:

Safety: Given a specification, does the system reach a bad configuration?
A program is said to be correct if and only if it respects its specifications. Formal verification uses mathematical methods to prove that a program is correct. It is a rigorous framework that determines the absence of errors in the program or a model of it, unlike testing, simulation, static analysis and also simple debugging techniques, such as inserting assertions and print statements in the source code, which show the presence of software defects.

The first approach to formal verification is deductive reasoning, where the program analysis mechanizes the burden of proof: starting from the ground instructions of the programming language, it uses a theorem prover to show that every construction implemented in the program can be derived using mathematical formulas. This approach of logical inference exposes every subtle details of the program that could otherwise be overlooked by a human. Given the complexity of some systems, it is usually the case that the proofs are long, take time to derive and experts are required to supply the formulas, potentially monitoring if extra ones are needed along the way. This limits the applicability of such methods to only specialized software.

1.2 Model Checking

The other approach, and focus of this thesis, is model-checking. This method proves automatically whether a model of the program satisfies a targeted property. It was introduced separately by Emerson and Clarke [23] and by Queille and Sifakis [54]. It was first applied, successfully and fully automatically, to hardware verification.

The method requires a program and a property as input. It first either extracts a model from the program, while omitting details that appear irrelevant for the given property,\(^1\) or it is directly given the model as input rather than the program itself. Notice that, similarly to loosely specifying the desired property, the model or the extraction can be wrong or unfaithful and therefore introduce errors in the verification process… The method then computes and returns either “yes” when the property is satisfied by the program, or “no” when it is not, in which case it can eventually also explain the reason by giving a counterexample. Due to the aspect of automation, model checking is sometimes given the name of “push-button technology”.

A state in the model captures any necessary information about the program to make itself distinct from another state, such as the values of the program variables, or which resources the program is currently holding. Alongside all the states of the system, the model depicts the transitions, i.e. how to move from one state to another state. Every behaviour of the system is represented as a succession of transitions, starting from some initial states. States and transitions together describe the operational semantics, that is, how every step

---

\(^1\)A difficult and intellectually interesting challenge on its own, and not the focus of this thesis.
of the system takes place in the model. We limit ourselves, here only, to a somewhat informal description. The formal notations will be introduced in the next chapters, tailored to the class of system we consider in this thesis.

It is sometimes possible to model the whole system with a finite number of states and transitions, which is then called a finite-state system. Many systems are finite-state, such as a hardware circuit with a few inputs. A simple calculator, for example, can not handle every integer. Despite the high number of possible inputs, its state-space is still finite (yet huge). Model-checking aims at a full coverage of program executions and must therefore explore the state-space entirely. It is done through a procedure that exhausts every transition of the system, starting from some initial states. Inspecting a transition from some state can lead to either the discovery of a new state, or a state that was already examined. The issue is when the state-space is of large size. It grows in-fact exponentially with the number of parameters or the size of their domain. Researchers have investigated several methods to tackle the state-space explosion problem.

The choice of which transition to pick during the exploration can be crucial for the efficiency of the procedure. Let us illustrate the reason with a tiny example: If we want a cup of tea, we put a tea bag and water in a cup and then, we remove the tea bag. Whether we put one first and then the other, the outcome is still the same: a cup of tea. The only constraint is to remove the tea bag after we have put both the tea bag and the water in the cup. Hence, exploring all orderings of events is not necessary, as it only makes algorithms re-inspect states that have already been examined. Partial order techniques aim at detecting and avoiding redundant situations, while retaining important dependencies among actions. They however do not reduce the state-space.

The main approach to combat the state-space explosion problem is to find ways to not represent every state of the system individually in the model, but rather group them together and form sets of states, using a compact and concise notation called symbolic representation. Partial order techniques and symbolic representations are orthogonal techniques that aim at accelerating the state-space exploration. In this thesis, we employ symbolic representations which allow algorithms to make bigger strides. For example, consider a switch with two positions: up and down. The switch is coupled to a light bulb and a counter using a display that shows six digits, that indicates how many times the light was on (i.e. with a maximum value of one million). Initially, the switch is down, the light is off and the counter is zero. When we shift the switch up, the light is turned on, and the counter is incremented. When we shift the switch down, the light is turned off and the counter is reset (to zero) only if it had reached its maximum value. We would like to check that the switch can not be up while the light is off. If we were to model all the information of the system, that is, a state containing the status of the switch, the status of the light bulb and the value of the counter, we would end up with potentially four million states \((2 \times 2 \times 1000000)\). However, it is obvious that the counter does
not play a role for the property at hand, so we could *disregard* that information and group the states into sets that are labeled with only the status of the switch and the light bulb. The initial label is \([\text{down} \mid \text{off}]\) and represents the states \(\langle \text{down}, \text{off}, \text{counter} = 0 \rangle, \langle \text{down}, \text{off}, \text{counter} = 1 \rangle, \ldots, \langle \text{down}, \text{off}, \text{counter} = 999999 \rangle\). The label \([\text{up} \mid \text{on}]\) represents the states \(\langle \text{up}, \text{on}, \text{counter} = 1 \rangle, \langle \text{up}, \text{on}, \text{counter} = 2 \rangle, \ldots, \langle \text{up}, \text{on}, \text{counter} = 1000000 \rangle\). The new symbolic representation allows the exploration to jump potentially through four sets labeled with \([\text{down} \mid \text{off}], \ [\text{down} \mid \text{on}], \ [\text{up} \mid \text{off}]\) and \([\text{up} \mid \text{on}]\). We would like to check if the system encounters a situation where its state is in the set labeled by \([\text{down} \mid \text{on}]\). The system using the symbolic representation moves from \([\text{down} \mid \text{off}]\) to \([\text{up} \mid \text{on}]\) and conversely from \([\text{up} \mid \text{on}]\) to \([\text{down} \mid \text{off}]\). We see that the state-space exploration never visits a state that belongs to the set labeled by \([\text{down} \mid \text{on}]\) so the system is safe. The gain is important: Assume that the exploration of potentially four million states would have taken ten minutes. The algorithm with symbolic representations would now leap through four sets only and run in less than a millisecond! The switch example is of course simple and does not reflect the complexity of today’s software. Symbolic representations are of crucial help to combat the state-space explosion, accelerate the algorithms and get them to terminate in a reasonable amount of time.

**Concrete reachable state-space**

**Abstract state-space using symbolic representation**

Tailoring a state to the given property by *dropping* irrelevant details (as opposed to disregarding them) and representing sets of states with compact notations introduce layers of *abstraction* into the model — an important notion for model-checking, whether in the definition phase or in the extraction phase of the model. It allows to transform a model into another model, called the *abstract model*, such that proving the property for the abstract model is easier and implies the property for the original model. In the switch example, we simply do not represent the value of the counter at all. The abstract model is only a switch and a light bulb. The original state-space has been swapped for another one that is easier and faster to explore, and still allows us to prove that the property holds in the original system.

Systems are however not always finite-state. When a state needs to contain variables that can take values from an unbounded domain or if it represents structures of unknown size, for example, then the list of states is unbounded and
the model is *infinite-state*. This infiniteness renders any exhaustive exploration, in the hope of covering the full state-space without encountering any bad state, obviously impossible! This thesis contributes to solving the problem of model-checking infinite-state systems.

We observe that there is a compelling need to reduce infinite-state systems to finite-state systems. It is done through the use of approximations — another layer of abstraction — where information is distorted in order to derive a finite-state model of the system. There are two types of approximations: over-approximations and under-approximations. In the case of under-approximations, the idea is to ignore some behaviours of the system by discarding some states or transitions, which gives an acceleration of the state-space exploration. This is of course not satisfactory to prove safety, but interesting to quickly detect bugs, as in testing. On the contrary, over-approximations introduce extra states and/or transitions, in order to find judicious symbolic representations. This in fact adds behaviours to the original system (or just modifies them). It is sound to prove safety of the abstract model in order to imply safety for the original system, since all the behaviours of the latter are represented in the former. The challenge is to find over-approximations that do not introduce behaviours that could turn out to be bad. Indeed, the method would return that the property is not satisfied and we would not know whether it comes from the approximation or from the concrete system itself.

To palliate to the imprecision caused by a too coarse over-approximation, it is possible to analyze the returned counter-example and find the origin of the problem. If it turns out to be a real concrete example, the method has in fact found a bug, and the property is surely not satisfied. Otherwise, the counter-example comes from the approximation, that is, there is a step in the sequence of events leading to that counter-example which is not performed by the original system but only by the abstract model. The approximation is be refined by discarding this step and the method should be run anew.

Nevertheless, finding suitable over-approximations is a challenge on its own. This thesis now revolves around the following problem statement.

**Safety**: *Given a specification and an (over-)approximation, does the abstract system reach a bad configuration?*
1.3 Parameterized Verification

Many systems can actually be characterized as a family of finite-state systems with one (or more) parameter ranging over an unbounded domain. For each fixed value of the parameter, the system is finite-state. However, a system that contains an \textit{a priori} unknown parameter must behave properly regardless of the value of the parameter. It is therefore considered infinite-state.

The \textit{parameterized verification} problem is to prove correctness of the system, against some specification, for all values of the parameter. For example, in the case of a program manipulating a list, the parameter could be the length of the list. The system as a whole is a set of finite-state systems, one for each length of the list. For a network protocol, the parameter could be the topology of the network and the system is to be proven safe regardless of how the network nodes are arranged.

We concentrate in this thesis on a specific class of \textit{parameterized systems}, namely, systems consisting of an arbitrary number of processes (see Chapter 2). The size of the system is the parameter of the verification problem. For each value \( n \) of the parameter, the system \( S_n \) is the parallel composition of \( n \) processes, which can interact with each other at any time. Given a specification, we prove safety for the family \( (S_n)_{n \geq 0} \) — as a whole — by showing the non-reachability of some (potentially infinite) set of \textit{bad states}. We use the following strategy:

1. We extract a model from each process in the system. This in turn defines the operational semantics with states and transitions for the entire family.
2. We use an over-approximation to derive an abstract model from the original (infinite-state) model. This defines a new state-space and set of transitions.
3. We determine two sets in the abstract model:
   - the initial states, usually mirroring the initial settings of the program,
   - the bad states, usually along the lines of the targeted property.
4. We finally check whether the bad states are reachable from the initial states in the abstract model.

By construction of the over-approximation, showing that the abstract model is safe, will imply that the original system is also correct with respect to its specification, regardless of the number of processes in the system. We present a formal definition and some examples in the next chapter.
Heap analysis

An important extension in this thesis is the application of parameterized verification to the complex problem of shape analysis. We consider programs that implement data-structures that can be concurrently accessed by an arbitrary number of threads. The data-structures, e.g. stacks and queues, are constructed using singly-linked lists. By following the chains of pointers, we observe that programs leave memory footprints that we coin *shapes* or *heaps*.

The number of threads is one dimension of the parametrization. Shape analysis can also be parameterized along other dimensions, such as the size of the data-structure or the data domain. In Chapter 5, we address the combined challenges of an arbitrary number of threads, an unbounded size of heaps and an unbounded data domain. Moreover, we do not assume the presence of a garbage collector which makes the task even more complex.

The program specification itself is infinite since the programs can manipulate data values from a domain of unbounded size. It is not enough to inspect each shape and conclude that it belongs to a particular set of bad shapes. We need to observe *traces* of events. We therefore pair each shape with a special *observer* whose role is to determine whether a sequence of events might not occur for a particular data-structure. We still apply the above-described strategy and conclude that the program reaches a bad configuration when the observer is in a bad state. We dedicate Chapter 5 to explain this complex problem and the verification method in further details.
1.4 Theoretical Limitations

It is important to understand the type of programs that the methods can handle and with which limitations. A problem is *decidable* if there exists an algorithm to solve *every* instance of the problem. Decidable problems can still be *intractable*, that is, even if a solution exists, it is difficult to compute because it requires too much time or space resources. On the contrary, in the case of undecidable problems, it is sometimes possible to design algorithms that can work well in practice for *some* instances but do not guarantee to compute a solution for all input combinations or even terminate in general. Research challenges regarding the theoretical limitations of algorithms consist of identifying (i) undecidable problems and (ii) classes of systems and specifications for which the verification problem is decidable and designing efficient algorithms for these.

The problem of parameterized verification is well-known to be undecidable in general [3] but for some subclass of systems (see Chapters 3 to 5), the verification problem becomes decidable, yet complex, and we present efficient algorithms to solve it.

We would like to pinpoint that this thesis is *not* about theoretical results on decidability of some class of programs. Rather than providing computational bounds on the time and/or amount of space in memory the verification algorithms require or comparing which algorithm is best, the thesis focuses on finding the suitable abstractions that make the algorithms *practical*, i.e. terminate in a reasonable amount of time. We shall not bother much about the amount of space they require, because we can assume that it is always possible to extend the capacity of the machine and re-run the methods.

In the coming chapters, we will describe the different techniques to make the algorithms usable in practice. This was the challenge and we present the results in this thesis.

*This focus of this thesis is on designing efficient algorithms for the intractable problem of parameterized verification for a specific class of programs.*
What we learned in Chapter 1

**Bugs** make computers crash and some of them are vital to fix.

**Testing** is a classical method to ensure software quality and requires to run the program. However, it does not cover all possible executions and therefore might leave errors undetected.

**Formal Verification** uses, on the other hand, mathematical proofs to derive the absence of errors.

**Properties** that are interesting to verify, usually fall into two categories: safety or liveness properties. We focus on safety properties, which state that the program never encounters a bad situation.

**Model-Checking** is a fully automatic method where a model of the program is extracted and checked against a property. States and transitions characterize how every step of the program takes place in the model. The number of states can be finite or infinite depending on the complexity of the program and its input parameters. This approach has the advantage that it does not require us to run the program but can suffer from state-space explosion.

**State-space explosion** is handled by grouping states together using compact representations and by using approximations.

**Over-approximations** aim to derive an abstract model in which it is easier to determine whether the original program is correct.

**Parameterized systems** consist in an arbitrary number of processes in parallel, communicating with each other at any time. The number of processes is the parameter of the verification problem. Parameterized systems are therefore considered infinite-state — and are the focus of this thesis.

**Parameterized verification.** Many systems can be characterized by a collection of finite-state systems with one (unbounded) parameter. The task is to prove correctness, regardless of the value of the parameter.

**Practical algorithms.** This thesis focuses on making the algorithms practical and not suffer too much from state-space explosion, rather than theoretical results on the decidability of classes of systems.
2. Parameterized Systems

In this thesis, we focus on systems consisting of an arbitrary number of processes. The size of the system is the parameter of the verification problem. For each value $n$ of the parameter, the system $S_n$ is the parallel composition of $n$ processes. Given a specification, we are interested in proving safety for the family $(S_n)_{n \geq 0}$ — as a whole — which is obtained through the non-reachability of some set $B$ of bad configurations, starting from some set $\mathcal{I}$ of initial configurations. Processes are usually, but not necessarily, copies of each other.

$$S_n = P \parallel P \parallel \cdots \parallel P_{n \text{ times}}$$

Parameterized systems arise naturally in the modeling of mutual exclusion algorithms, distributed protocols, or cache coherence protocols. For instance, sensor networks typically consist of thousands of identical nodes, web services must handle millions of requests of the same type. Mutual exclusion must be guaranteed regardless of the number of processes that participate in a given session of the protocol. Cache coherence must be ensured regardless of the number of cache lines or the number of physical processors. Those systems can be handled easily for small values of the parameter, that is, if the system involve a few components. It is interesting to extend the verification to the parameterized case, even though the state-space for the whole family is infinite.

2.1 Features

The parallel composition of several processes can be further characterized with the following features, which make the task of parameterized verification complex and interesting:

- the topology,
- the way the process can communicate with each other and
- the nature of each process itself, whether it is finite-state or not. In this chapter, each component is however restricted to be finite-state.

The topology describes the way the processes are arranged and implicitly how they can refer to each other, without necessarily revealing their identity. For the linear topology, processes are organized in an array and can distinguish between their left and right neighbors. For a ring, they can inspect their
immediate neighbor, while for a tree, they inspect their parent and/or children processes. Finally, the case where there is no particular structure and where processes can refer to any other processes is called a multiset topology.

Processes can interact with each other and perform actions potentially in any order or simultaneously. Those actions are conditioned on the status of the other processes: Before it performs its action, a process can inspect other processes according to the topology, and their status can allow or prevent the action from the process. We refer to these transitions as being guarded by a global condition, or just global transitions. For example, in a linear topology, a process (at position \( i \)) may be able to perform a transition only if all processes to its left (i.e. with index \( j < i \)) satisfy a particular property. Sometimes, it is only required for some rather than all, as depicted in Figure 2.1. On the other hand, it is occasionally not necessary to refer to other processes at all before performing an action. These actions are by opposition called local transitions.

A major issue is that global conditions are not necessarily checked atomically. In case other processes can perform transitions, while a global condition check is carried out, the system must in fact distinguish intermediate states. These interleavings make the state-space grow even further. We will see that the method from Chapter 4 can handle elegantly this complex setting.

Apart from local and global transitions, we complete the list with the other types of transitions, which might depend on the topology or not. For a broadcast transition, a process is forced to change state synchronously with an arbitrary number of processes. For a rendez-vous transition, it is only required from one extra process. For shared variable update, a process communicates the updated value to the other processes. Finally, process creation and deletion make the topology dynamic, but it is often not a major issue, as shown in Paper I.

Figure 2.1. Example of global transition for a linear topology: a process in state \( s \) may change to state \( d \) provided that there exists another process in state \( w \) on its right.
2.2 Formal Definition

To simplify the presentation, we will only focus, in this chapter, on the case where processes are governed by a finite-state automaton and organized in a linear topology. Other topologies are presented in paper I, VI and IV.

A parameterized system is a pair \( P = (Q, \Delta) \) where \( Q \) is a finite set of \textit{local states} of a process and \( \Delta \) is a set of \textit{transition rules} over \( Q \). A transition rule is either \textit{local} or \textit{global}. A local rule is of the form \( \text{src} \rightarrow \text{dst} \), where the process changes its local state from \( \text{src} \) to \( \text{dst} \) independently from the local states of the other processes. A global rule is either \textit{universal} or \textit{existential}. Recall that a global rule depends on the topology, so it is here of the form:

\[
\text{if } Q \ j \sim i : S \text{ then src } \rightarrow \text{ dst}
\]

where \( Q \) is either \( \exists \) or \( \forall \), for existential and universal conditions respectively, where \( \sim \) is either \(<, > \) or \( \neq \), to indicate which processes are concerned, and where \( S \) is a subset of \( Q \), describing the state of the other \textit{witness} processes. We call \( \text{src} \) the \textit{source}, \( \text{dst} \) the \textit{destination}, \( Q \) the \textit{quantifier} and \( \sim \) the \textit{range}. Informally, the \( i^{th} \) process changes its local states (among \( S \)) for some \( i \) and either (i) \( \delta \) is a local rule \( \text{src} \rightarrow \text{dst} \), where the \( i^{th} \) process checks the local states (among \( S \)) of the other processes when it makes the move. We consider, in this section only, a version where each process checks \textit{atomically} the other processes. The more realistic and more difficult case, where the atomicity assumption is dropped, will be introduced in the next section. For instance, the condition \( \forall j < i : S \) means that “every process \( j \), with a lower index than \( i \), should be in a local state that belongs to the set \( S \).” The condition \( \exists j > i : S \) means that “there should be a process \( j \) with higher index than \( i \) with a local state listed in the set \( S \),” etc…

A \textit{configuration} in \( P \) is a word over the alphabet \( Q \), i.e. an array of process states. We use \( C \) to denote the set of all configurations. For a configuration \( c \in C \), we use \( c[i] \) to denote the state at position \( i \) in the array and \(|c|\) for its size. We use \([a,b]\) to denote the set of integers in the interval \([a,b]\) (i.e. \([a,b] = [a,b] \cap \mathbb{N})\).

For a configuration \( c \in C \), a position \( i \leq |c| \), and a rule \( \delta \in \Delta \), we define how to transform the configuration \( c \) into another configuration if we allow the \( i^{th} \) process in the array to perform the transition \( \delta \). Formally, we define the immediate successor of \( c \) under a \( \delta \)-move of the \( i^{th} \) process, such that \( \delta(c,i) = c' \) if and only if \( c[i] = \text{src} \), \( c'[i] = \text{dst} \), \( c[j] = c'[j] \) for all \( j : j \neq i \) and either (i) \( \delta \) is a local rule \( \text{src} \rightarrow \text{dst} \), or (ii) \( \delta \) is a global rule if \( Q \ j \sim i : S \text{ then src } \rightarrow \text{ dst} \), and one of the following two conditions is satisfied:

- \( Q = \forall \) and for all \( j \in [1,|c|] \) such that \( j \sim i \), it holds that \( c[j] \in S \),
- \( Q = \exists \) and there exists some \( j \in [1,|c|] \) such that \( j \sim i \) and \( c[j] \in S \).

Note that \( \delta(c,i) \) is not defined if \( c[i] \neq \text{src} \). We use \( c \rightarrow c' \) when \( c' = \delta(c,i) \) for some \( i \leq |c| \), and \( c \rightarrow c' \) if \( c \rightarrow c' \) for some \( \delta \in \Delta \). We define \( \rightarrow^* \) as the reflexive transitive closure of \( \rightarrow \) (i.e. the result of repeatedly applying \( \rightarrow \)).
2.3 Non-Atomic Global Conditions

We extend our model to handle parameterized systems where global conditions are not checked atomically. We replace both existentially and universally guarded transitions by the following variant of a for-loop rule

\[
\text{if foreach } j \sim i : S \text{ then } \text{src} \rightarrow \text{dst} \text{ else } \text{src} \rightarrow e
\]

where \( e \in Q \) is an escape state and the other labels src, dst, \( \sim \) and \( S \) are as in the previous section. Essentially, for a configuration with linear topology, a process at position \( i \) inspects the state of another process at position \( j \), in-order. Without loss of generality, we will assume that the for-loops iterate through process indices in increasing order. If the state of the process at position \( j \) does not belong to \( S \), process \( i \) escapes to state \( e \). Otherwise, process \( i \) moves on to inspect the process at position \( j + 1 \), unless there is no more process to inspect in which case process \( i \) completes its transition. This construction allows us to emulate both existential and universal transitions by adjusting the set \( S \) and choosing the right combination of src/dst/e states.

The transition systems of the previous section are extended with for-loop rules in the following way: A configuration is now a pair \( c = (q_1 \ldots q_n, \checkmark) \) where \( q_1 \ldots q_n \in Q^+ \) is a word as before and where \( \checkmark : [1,n] \rightarrow [0,n] \) is a total map which assigns to every position \( i \) of \( c \) the last position which has been inspected by the process \( i \). Initially, \( \checkmark(i) \) is assigned 0.

We fix a rule \( \delta = \text{if foreach } j \neq i : S \text{ then } s \rightarrow t \text{ else } s \rightarrow e \) from \( \Delta \), a configuration \( c \) with \(|c| = n \), and \( i \in [1,n] \). We first define the position \( \text{next}(c,i) \) which the process at position \( i \) is expected to inspect next. Formally, \( \text{next}(c,i) = \min\{j \in [1,n] \mid j > \checkmark(i), j \sim i\} \) is the smallest position larger than \( \checkmark(i) \) which satisfies \( \text{next}(c,i) \sim i \). Notice that if process \( i \) has already inspected the right-most position \( j \) which satisfies \( j \sim i \), then (and only then) \( \text{next}(c,i) \) is undefined.

We distinguish three types of \( \delta \)-move on \( c \) by the process at position \( i \): (i) \( \delta^\mu(c,i) \) for a loop iteration, (ii) \( \delta^e(c,i) \) for escaping and (iii) \( \delta^t(c,i) \) for termination. Each type of move is defined only if \( q_i = s \).

![Figure 2.2](image-url)

*Figure 2.2. if foreach \( j \neq i : \neg\{1, 2\} \) then 3 \( \rightarrow \) 7 else 3 \( \rightarrow \) 4*
(i) $\delta^{it}(c, i)$ is defined if $\text{next}(c, i)$ is defined and $q_{\text{next}(c, i)} \in S$. It is obtained from $c$ by updating $\checkmark(i)$ to $\text{next}(c, i)$. Intuitively, process $i$ is only ticking position $\text{next}(c, i)$.

(ii) $\delta^{e}(c, i)$ is defined if $\text{next}(c, i)$ is defined and $q_{\text{next}(c, i)} \notin S$. It is obtained from $c$ by changing the state of the process $i$ to $e$ and resetting $\checkmark(i)$ to 0. Intuitively, process $i$ has found a reason to escape.

(iii) $\delta^{t}(c, i)$ is defined if $\text{next}(c, i)$ is undefined, and it is obtained from $c$ by changing the state of the process $i$ to $t$ and resetting $\checkmark(i)$ to 0. Intuitively, process $i$ has reached the end of the iteration and terminates its transition (i.e. moves to its target state).

2.4 The Reachability Problem

An instance of the reachability problem is defined by

- a parameterized system $\mathcal{P} = (Q, \Delta)$,
- a set $\mathcal{I} \subseteq Q^{+}$ of initial configurations, and
- a set $\mathcal{B} \subseteq Q^{+}$ of bad configurations.

We say that $c \in C$ is reachable if there are $c_0, \ldots, c_m \in C$ such that $c_0 \in \mathcal{I}$, $c_m = c$, and for all $0 \leq n < m$, there are $\delta_n \in \Delta$ and $j \leq |c_n|$ such that $c_{n+1} = \delta_n(c_n, j)$. In other words,

$$c_0 \in \mathcal{I} \rightarrow c_1 \rightarrow \cdots \rightarrow c_{m-1} \rightarrow c_m = c \quad \text{or} \quad c_0 \in \mathcal{I}^{*} \rightarrow c$$

We use $\mathcal{R}$ to denote the set of all reachable configurations (from $\mathcal{I}$). We say that the system $\mathcal{P}$ is safe with respect to $\mathcal{I}$ and $\mathcal{B}$ if no bad configuration is reachable, i.e. the sets $\mathcal{R}$ and $\mathcal{B}$ do not intersect ($\mathcal{R} \cap \mathcal{B} = \emptyset$). The set $\mathcal{I}$ of initial configurations is usually a regular set. The set $\mathcal{B}$ of bad configurations will be defined in Chapter 3.
2.5 A Challenging Example

We illustrate the notion of a parameterized system with the example of Szymanski’s mutual exclusion protocol [57, 58]. Among other properties, this protocol ensures exclusive access to a shared resource in a system consisting of an unbounded number of processes organized in an array. The critical section is the portion of code in the program which threads are allowed to execute one at a time. The source code for each process is presented in Figure 2.4. For Szymanski’s protocol, the critical section is composed of the statements on line 9 and 10.

Each process participating in a session of the protocol is represented at a fixed position in the array with a local variable flag and how far it has proceeded in its execution. We encode the state of the ith process with a number, which reflects the values of the program location and the local variable flag[i].

A configuration of the induced transition system is a word over the alphabet \( \{0, 1, \ldots, 11\} \) of local process states. The size of a configuration is the parameter of the system. The initial configurations characterize the program before any execution, e.g. using the initial values of the program variables. Here, all processes are initially in state 0, i.e. \( I = 0^+ \). The bad configurations are derived from the targeted property. For a mutual exclusion protocol, a configuration is considered to be bad if it contains two occurrences of state 9 or 10, i.e. at least two processes are in their critical section simultaneously. In other words, the bad configurations belong to an infinite set characterized by the following patterns: \( 0 \rightarrow 9 \rightarrow 9 \rightarrow 10 \rightarrow 9 \rightarrow 10 \rightarrow 9 \rightarrow 10 \rightarrow 10 \rightarrow 10 \rightarrow \).

A transition moves a process from a state to another, provided that the other processes respect the global condition. Here, for example, processes move from state 1 to 2, if the other processes are all in state 0, 1, 2, 5 or 6. If not, the process stays in state 1.

The intuition that Szymanski gives is the presence of a “waiting room”, with an entering and exiting door, before processes move into their critical section. The transition rules are depicted in Figure 2.3 using a simple graphical representation, often called a transition diagram: The label on an edge represent the global conditions that other processes must respect in order to perform the transition. There is no label when
the transition is local. For example, consider the edges between the nodes \(3, 4\) and \(7\). They depict the transition as in Figure 2.2

\[
\text{if foreach } j \neq i : \neg\{1, 2\} \text{ then } 3 \rightarrow 7 \text{ else } 3 \rightarrow 4.
\]

The task is to check that the protocol guarantees exclusive access to the shared resource regardless of the number of processes, i.e. to show that the bad configurations are not reachable from the initial ones. Many techniques [8, 7, 10, 22, 50, 11, 21] have been used to verify automatically the safety property of Szymanski’s mutual exclusion protocol but only in restricted settings. They either assume atomicity of the global conditions and/or only consider a more compact variant of the protocol. The full and fine-grained version has been considered a challenge in the verification community. To the best of our knowledge, this thesis presents the first technique to address the challenge of verifying the protocol fully automatically without atomicity assumption.

```c
0 flag[i] = 1;
1 for(j=0; j<N; j++) { if(flag[j] >= 3) goto 1; }
2 flag[i] = 3;
3 for(j=0; j<N; j++) {
   if (flag[j] = 1) {
      flag[i] = 2;
      for(j=0; j<N; j++) { if(flag[j]==4) goto 7; }
      goto 5;
   }
}
4 flag[i] = 4;
5 for(j=0; j<i; j++) { if(flag[j] >= 2) goto 8; }
6 /* Critical Section */
7 for(j=i+1; j<N; j++) {
      if(flag[j]==2 || flag[j]==3) goto 10;
   }
8 flag[i] = 0; goto 0;
```

*Figure 2.4. Implementation of Szymanski’s protocol (for process i)*
What we learned in Chapter 2

Parameterized systems are infinite-state and the focus of this thesis. They are formally defined as a pair $\mathcal{P} = (Q, \Delta)$ where $Q$ is a finite set of process states and $\Delta$ is a set of local or global transitions over $Q$.

Induced Model. The system $\mathcal{P}$ induces an infinite-state model $(\mathcal{C}, \rightarrow)$ which describes the configurations $\mathcal{C}$ and how to update from a configuration to another using the transition relation $\rightarrow$.

Reachability problem is now formally defined using the induced model, the set bad configurations $\mathcal{B}$ and the set of initial configurations $\mathcal{I}$.

Features. Parameterized systems can be designed with several features in mind, especially regarding the topology and the type of communication allowed between processes.
3. Monotonic Abstraction

We now present a technique to solve the reachability problem presented in the previous chapter. This technique has been introduced by Abdulla et al. [1] and is based on the Well-Quasi Ordering framework.

We first introduce the notion of ordering and monotonicity and how we use them to derive an over-approximation. We then present the algorithm to prove the safety property. Finally, we discuss termination and show how we applied the method on a few protocols. While the method has been applied to linear topologies [5], one of the contributions of this thesis is to apply it to multiset topologies, to shape analysis and to extend it to tree topologies.

3.1 Upward-Closed Sets

We introduce a partial ordering between the configurations, namely the subword relation \( \sqsubseteq \), which allows us to compare two configurations (of potentially different size) and determine if one is “smaller” than the other.

Let \( u \) and \( v \) be two configurations of size \( n \) and \( m \) respectively, with \( n \leq m \). Intuitively, we would like to check if the configuration \( u \) can be “injected” inside \( v \) (or that \( v \) can be “reduced” to \( u \)). We first define \( H^m_n \) as the set of strictly increasing injections from \([1,n]\) to \([1,m]\), i.e. for \( h \in H^m_n \), \( 1 \leq i < j \leq n \implies 1 \leq h(i) < h(j) \leq m \). (Recall \([a,b] = [a, b] \cap \mathbb{N} \) is the range of integers in the interval between \( a \) and \( b \)). Then, we write \( u \sqsubseteq v \) if there exists \( h \) in \( H^m_n \) such that \( u[i] = v[h(i)] \) for all \( i \in [1,n] \) (see Figure 3.1). If there is no injection that can be found to compare \( u \) and \( v \), they are simply incomparable.

For a configuration \( c \in C \), we define its upward-closure as the set of all the configurations of any (larger) size, that at least contain \( c \) as a subword (depicted on the right).

\[
[c] = \{ u \in C \mid c \sqsubseteq u \}
\]

Notice that \( c \sqsubseteq c' \) implies \( [c] \supseteq [c'] \), meaning that the smaller a configuration is in the ordering, the larger set of configurations it represents. We abuse the notation and for a set \( A \), we use \( [A] = \cup_{a \in A} [a] = \{ u \in C \mid \exists a \in A : a \sqsubseteq u \} \).
We say that a set $A \subseteq C$ is upward-closed if $A = \lfloor A \rfloor$. Moreover, for an upward-closed set $U$, we define the minimal elements of $U$, that is, the set $M$ such that

(i) [Closure] $\lfloor M \rfloor = U$, i.e. $U$ can be generated by taking the upward-closure of $M$,

(ii) [Minimality] $\forall a, b \in M, a \sqsubseteq b \implies a = b$, i.e. all the elements of $M$ are incomparable with respect to the subword relation $\sqsubseteq$.

The set $M$ is therefore uniquely defined for a given $U$ and denoted $\text{min}(U)$.

Characterizing the bad configurations for the mutual exclusion property of Figure 2.4, is now an easy task since they all follow a simple pattern. The smallest bad configurations for the protocol in Figure 2.3 is the set $B_{\text{min}} = \{9 \ 9, 10 \ 9, 9 \ 10, 10 \ 10\}$. Any bad configuration contains at least one of the elements of $B_{\text{min}}$ as a subword. We can therefore craftily define the set $B$ of all the bad configurations as the upward-closure of $B_{\text{min}}$. The set $B_{\text{min}}$ are the minimal elements of $B$, i.e. $\text{min}(B) = B_{\text{min}}$. We represent the whole set of bad configurations (albeit infinite) using a compact and finite symbolic representation.

$$B = \lceil B_{\text{min}} \rceil = \{c \in C \mid \exists b \in B_{\text{min}} : b \sqsubseteq c\}$$

### 3.2 Monotonicity

Monotonicity is a mathematical notion that comes from calculus,\(^1\) which deals with functions over sets that preserve a given preorder $\leq$. For a monotonic function $f$ over a domain $D$, whenever two elements $a$ and $b$ in $D$ are ordered such that $a \leq b$, it holds that $f(a) \leq f(b)$.

A parameterized system $P = (Q, \Delta)$ is monotonic (with respect to preorder $\sqsubseteq$), if for each rule $\delta \in \Delta$ and configurations $c_1, c_2$ and $c_3$, such that $c_1 \xrightarrow{\delta} c_2$ and $c_1 \sqsubseteq c_3$, then there exists a configuration $c_4$ such that $c_3 \xrightarrow{\delta} c_4$ and $c_2 \sqsubseteq c_4$. That is to say, if we can fire a transition on a configuration, we can also fire it on a larger configuration, and the results are ordered accordingly. This is illustrated in Figure 3.2.

In [3], it is shown that for monotonic systems, the upward-closedness is preserved when computing predecessors, that is, if a set is upward-closed, so is its pre-image. This is an important property that we use in the next section.

\(^1\)In calculus, a function $f$ defined on a subset $D$ of the real numbers $\mathbb{R}$ is called monotonic if it is either entirely increasing or decreasing. It is called increasing (resp. decreasing), if for all $x$ and $y$ in $\mathbb{R}$ such that $x \leq y$, it holds that $f(x) \leq f(y)$ (resp. $f(x) \geq f(y)$). So $f$ either preserves the order, or reverses the order, consistently over the domain $D$. 

46
However, systems are not always monotonic. This is for example the case if \( \Delta \) contains an universally quantified global transition. For example, if \( \delta \in \Delta \) is of the form if \( \forall j \neq i : \{w\} \text{ then } s \rightarrow d \), the subword relation is not necessarily preserved by \( \delta \) as depicted in Figure 3.3.

It has been shown that, for monotonic systems, the problem of determining if a set of bad configurations is reachable from some initial configurations is decidable [12], even though the state-space is infinite. Therefore, the strategy is to make the system monotonic by introducing an over-approximation. Existential and local transitions are preserved from the original system as-is, since they are monotonic. Universal transitions, however, are not and we over-approximate them using the scenario depicted in Figure 3.3. Informally, for every rule \( \delta \in \Delta \), we define a new successor function \( \hat{\delta} \) such that \( \hat{\delta} \) coincide with \( \delta \) if \( \delta \) is a local transition or an existential global transition. In the case where \( \delta \) is an universal global transition, we define \( \delta \) in the following manner (here only giving the intuition): We first remove the process states from the configuration, if they violate the guard, i.e. if they don’t respect the global condition from the rule (which effectively disables the transition). This creates a potentially smaller configuration on which we can apply \( \delta \) as usual. We say that we “go down on the ordering” before applying \( \delta \).

The successor functions \( \hat{\delta} \) form a new set \( \hat{\Delta} \) and the abstract parameterized system \( \hat{P} = (Q, \hat{\Delta}) \) induces a transition system \( (C, \leadsto) \) that is now monotonic and preserves the order \( \sqsubseteq \) (as depicted in Figure 3.4). It is decidable to prove that the bad configurations are not reachable from some initial ones in the new abstract transition system \( \hat{P} \). Since it is an over-approximation, it contains the original transition system \( (C, \rightarrow) \) and proving the abstract system safe implies that the original parameterized system \( P \) is also safe.
3.3 Backward Reachability

The procedure that will explore the state-space needs to work on symbolic representations, rather than the configurations themselves, in the spirit of Section 1.2. As we saw in Section 3.1, it is simple, using upward-closure, to represent the set of all bad configurations. Since we also know that a monotonic transition preserves the upward-closedness when computing predecessors, the interesting idea is then to use upward-closed sets as symbolic representations, define their pre-image and run a backward analysis from the bad set (using the abstract rules \( \hat{\Delta} \)).

When computing predecessors, there is an important mathematical detail to catch first: The pre-image of an upward-closed set is not necessarily the same as taking the upward-closure of the pre-image of its minimal elements. For a configuration \( c \in C \) and a transition \( \delta \in \hat{\Delta} \), the set \( \{ u \in C \mid u \delta \rightarrow c \} \) of configurations that can reach \( c \) in one \( \delta \)-step, might be empty (i.e. the pre-image of a given configuration does not exist). The configuration \( c \) is potentially not attainable, but its upward-closure could be. Therefore the pre-image must be a computation over sets of configurations as a whole rather than individual configurations only. So we compute a slightly different set: the configurations that can reach the upward-closure of \( c \) in one \( \delta \)-step (not forcibly \( c \) itself), that is, the set \( \{ u \in C \mid \exists v \in \lfloor c \rfloor : u \delta \rightarrow v \} \).

More generally, for an upward-closed set \( U \subseteq C \), we define its \( \delta \)-pre-image as the set \( \delta^{-1}(U) = \{ u \in C \mid \exists v \in U : u \delta \rightarrow v \} \) and its pre-image as

\[
Pre(U) = \bigcup_{\delta \in \hat{\Delta}} \delta^{-1}(U) = \{ u \in C \mid \exists v \in U : u \rightarrow v \}
\]

By monotonicity, it follows that the pre-image of an upward-closed set \( U \) is also upward-closed. Notice though, that the minimal elements of the pre-image \( (\min(Pre(U))) \) are not necessarily the inverse-image of its minimal elements \( (\bigcup_{c \in \min(U)} \{ u \in C \mid u \delta \rightarrow c \}) \).

Scheme

Given a upward-closed set of (bad) configurations, the procedure is constructed to compute the fixpoint of the function \( X \mapsto X \cup Pre(X) \). Intuitively, the analysis computes the configurations that could reach the bad set, by applying successively any rule from \( \hat{\Delta} \).

More precisely, the procedure computes a sequence of sets \( U_0, U_1, U_2, \ldots \) such that \( U_0 = B = \lfloor B_{\min} \rfloor \) and for all \( i \geq 0 \), \( U_{i+1} = U_i \cup Pre(U_i) \). Every \( U_i \) represents the set of configurations that can reach \( B \) in at most \( i \) steps. By monotonicity, every \( U_i \) is upward-closed.
Observe that the sequence \((U_i)_{i \in \mathbb{N}}\) is increasing, i.e. \(U_i \subseteq U_{i+1}\) for all \(i \geq 0\). If the backward procedure reaches a point \(n\) such that \(U_n \supseteq U_{n+1}\), it follows that \(\forall m \geq n, U_m = U_n\) and the sequence converges. Consequently,
\[
\{ c \in C \mid \exists b \in \mathcal{B} : c \xrightarrow{*} b \} = U_n = \bigcup_{i \in \mathbb{N}} U_i = \text{Pre}^*(\lceil \mathcal{B}_{\text{min}} \rceil)
\]
Furthermore, if \(U_n \cap \mathcal{I} = \emptyset\) holds, the initial configurations cannot reach the set of bad configurations \(\mathcal{B}\) and since \(\hat{\mathcal{P}}\) is an over-approximation of \(\mathcal{P}\), the system \(\mathcal{P}\) is safe. \(^2\)

The procedure is designed as a worklist algorithm, described in Algorithm 1, which manipulates upward-closed sets. It takes as input an upward-closed set of bad configurations \(\mathcal{B}\) and maintains two lists of sets: (i) a list \(\mathbb{W}\) of sets of configurations that have not yet been analyzed, initialized to \(\{\mathcal{B}\}\) and (ii) a list \(\mathbb{V}\), initially empty, that contains information about the sets of configurations that have already been analyzed.

### 3.4 Procedure and Termination

We are facing several issues to implement Algorithm 1. The latter is only a scheme as it manipulates potentially infinite upward-closed sets of configurations. Even though upward-closed sets can be fully characterized using their minimal elements, why would it guarantee that there are finitely many minimal elements for each upward-closed set?

The answer lies in a property of the subword relation: it is a well-quasi ordering (WQO for short). An ordering \(\leq\) over a set \(A\) is said to be a WQO if for any infinite sequence \(a_0, a_1, a_2, \ldots\) of elements of \(A\), there exists \(i\) and \(j\) such that \(i < j\) and \(a_i \leq a_j\). The definition implies that there is no infinite strictly decreasing sequence of elements of \(A\). The subword relation \(\subseteq\) is in fact a WQO (by Higman’s lemma) and in particular, every set of minimal elements is finite. (Recall indeed that minimal elements are incomparable).

Since every upward-closed set is characterized by a finite number of minimal elements, the scheme in algorithm 1 can be adapted to manipulate individual

---

\(^2\)In the opposite case, it is necessary to examine whether it is a real error or if it is an artifact of the abstraction, and therefore a spurious example. For spurious examples, it is possible to automatically refine the abstraction and re-run the procedure, until it finds a suitable abstraction, that proves the system safe, or escapes with a real error [4].
configurations, that are used as symbolic representations for upward-closed sets. The pre-image of such a minimal configuration \( m \) is now defined as:

\[
Pre(m) = \bigcup_{\delta \in \Delta} \min(\delta^{-1}([m]))
\]

The implementation is presented in Algorithm 2. It initializes the worklist to be the set of minimal bad configurations \( B_{\text{min}} \). The list \( V \) and \( W \) now only contain configurations, as symbolic representations of upward-closed sets. Line 10 ensures that the visited configurations are incomparable with each other, i.e. \( V \) is a set of minimal elements, while the configurations from \( W \) are potentially comparable. Line 6 tests whether a configuration should be discarded, because the set it represents already belongs to the visited set. The test on line 5 is usually carried out using automata theoretic constructs, but it is usually simple and not a bottleneck. An illustration of a backward run is depicted in Figure 3.5.

The procedure computes a sequence \( V_0, V_1, V_2, \ldots \) of (visited) sets of minimal configurations such that \( V_0 = B_{\text{min}} \) and for all \( i \geq 0 \), \( [V_{i+1}] \supseteq [V_i] \). Assume that the increasing sequence \( ([V_i])_{i \in \mathbb{N}} \) is not converging. There is then no such point \( m \) for which \( [V_m] = [V_{m+1}] \), and the sequence is strictly increasing. We can pick an element \( c_{m+1} \) in \( V_{m+1} \) that is not in \( V_m \), and therefore, extract an infinite sequence of configurations \( (c_i)_{i \in \mathbb{N}} \) such that those configurations are all incomparable with each other, with respect to the subword relation \( \sqsubseteq \). This is a contradiction with the fact that \( \sqsubseteq \) is a WQO. Consequently, the sequence \( ([V_i])_{i \in \mathbb{N}} \) must converge, say at point \( n \), and then

\[
\{ c \in C \mid \exists b \in B : c \xrightarrow{*} b \} = [V_n] = \bigcup_{i \in \mathbb{N}} [V_i]
\]

The procedure is then guaranteed to terminate if the ordering in use is a WQO. If it furthermore holds that \( [V_n] \cap \mathcal{I} = \emptyset \), then the bad configurations are not reachable from the initial ones and the safety property is proven.

---

3Defining the pre-image as \( Pre(m) = \min(\bigcup_{\delta \in \Delta} \delta^{-1}([m])) \), is only slightly more efficient theoretically since it avoids in the union to carry around comparable elements, but we have yet to see any real difference using the benchmarks from Chapter 6.

4Recall that for two configurations \( u \) and \( v \), \( u \sqsubseteq v \) implies \( |u| \supseteq |v| \) (and reciprocally).
3.5 Applications

The method has been applied on a variety of protocols such as cache coherence protocol or distributed protocols, presented in Chapter 6. In [6, 10], Abdulla et al. apply the method to parameterized systems, in particular with linear topologies and atomic global conditions. We have applied the method to protocols modeled as a Petri Net (i.e. a multiset) and extended it to tree topologies.

3.5.1 Multisets …and in particular Petri Nets

In this section, we apply the technique of monotonic abstraction and symbolic backward reachability to the problem of Race Detection. In particular, we focus on race-freeness, that is, the absence of race conditions (also known as data races) in shared-variable pthreaded programs. Such programs can be modelled as parameterized systems that consist of an arbitrary number of identical finite-state processes, competing for a global resource and communicating through a finite set of variables. The systems must satisfy mutual exclusion, that is, the
safety property stating that at most one process may hold the global resource at a time.

Race conditions can lead to devious bugs that are hard to track, due to non-determinism and limited reproducibility. Errors caused by race conditions are very subtle and often manifest themselves in the form of corrupted or incorrect variable data. Unfortunately, this often means that the error will not harm the system immediately, but only when some other code is executed, which relies on the data to be correct. This makes the process of locating the original race condition even more difficult.

Detecting a race condition, for a particular shared variable in the code, is akin to finding places where two threads (or more) are accessing the variable and one of them is performing a write operation. The timing of the read/write or write/write operations is critical to determining the value of the variable that the threads read. A program cannot depend on such imprecision as it could compute wrong results only due to “unfortunate” interleavings of the different thread operations.

Most programmers guard themselves against such situations by synchronizing the threads, which should allow only some interleavings to happen, in a controlled manner. They introduce locks around the shared resource they want to protect and make sure that all threads follow the locking discipline they impose. Many race detection tools (e.g. [55]) investigate therefore, in a dynamic fashion, program executions to detect violations of the locking discipline. They however do not guarantee a full coverage (see Chapter 1). Moreover, some programs require extra annotations along the code, to state some form of specification (or contract) that code should follow in order to behave as the programmer intended. Given that industrial-size concurrent programs is becoming increasingly important, this approach reaches its limit and the analysis should, as much as possible, be automatic.

The technique presented in the section detects the race conditions themselves for a particular class of pthreaded C programs. There is no need for annotations and works also in the absence of explicit locking mechanisms. We first extract a model from the code in the form of a Petri Net (see, e.g. Figure 3.6). The extraction is already a challenge and an over-approximation is introduced to
cope with many C intricacies. We do not cover it in this section, but the details of the model and how it is extracted are presented in paper VI.

We then verify the Petri Net model. Transition rules are characterized by the usual Petri Net flow relation, with input and output places. Informally, each place in the Petri Net (or a set of places) represents an instruction from the program code and the number of tokens in a place represent the number of threads currently executing that instruction. Firing transitions moves tokens from the input places to the output places of the transition only if there were “enough” tokens in the input places.

The key aspect of the model is that a shared variable $v$ is associated with two places, $Read_v$ and $Write_v$, in the Petri Net (in green in the figures). A process places a token in $Read_v$ (resp. $Write_v$) if it is currently accessing $v$ for reading (resp. writing). That way, we can distinguish situations where a read and a write assignment on some shared variable can happen simultaneously, effectively detecting a race condition.

A configuration in the system is a multiset, i.e a valuation of the tokens in each place of the Petri Net, often called a marking. The subword relation is replaced by a relation on multisets such that, for a Petri Net with markings $PN_1$ and $PN_2$, we write $PN_1 \sqsubseteq PN_2$ if and only if, intuitively, tokens can be removed from $PN_2$ to obtain $PN_1$ (see Figure 3.7). The set of bad configurations is characterized by the upward-closure of the (minimal) elements depicted on the right. The multiset ordering is a WQO, so procedure 2 is guaranteed to terminate and proves the race-freeness of the system.

3.5.2 Shape analysis

This section focuses on the verification of sequential iterative programs manipulating dynamic memory heaps, only briefly since we dedicate a full chapter (Chapter 5 on page 77) to shape analysis, using yet another abstraction technique, which handles the concurrent case. More precisely, heap structures are built using heap cells which contain one next-selector, i.e. programs are manipulating (possibly circular and shared) singly-linked lists.
We model heaps by labeled graphs, where labels correspond to positions of program variables. In fact, heap graphs are symbolic representations to characterize sets of heaps instead of a single one. The main issue is to define a preorder $\sqsubseteq$ on heap graphs. We introduce the following preorder: Given two graphs $g_1$ and $g_2$, we have $g_1 \sqsubseteq g_2$ if $g_1$ can be obtained from $g_2$ by a sequence of transformations consisting of either deleting an edge, a variable, or an isolated vertex, or of contracting segments (i.e., sequence of vertices) without sharing in the graph. The transition system, abstraction and the procedure to check the entailment relation $\sqsubseteq$ are described in details in paper IV.

The analysis allows to check properties such as absence of null dereferencing as well as absence of garbage creation. Moreover, it allows to check “shape” properties over heaps, such as “well-formedness”, where, for instance, the output is always a list without sharing. We show that these kinds of verification problems can be reduced to the problem of reaching sets of bad configurations corresponding to the existence in the heap graph of some minimal bad patterns. For instance, the set of configuration with garbage can be represented by minimal graphs containing all programs variables plus one isolated vertex.

We applied the monotonic abstraction method to verify such sequential programs fully-automatically (see Paper IV). Furthermore, the graph relation $\sqsubseteq$ is proven to be a WQO so procedure 2 is guaranteed to terminate.

### 3.5.3 Tree topologies

Finally, we extend the method to parameterized systems, here organized according to a tree topology. The topology and the communication primitives define the behaviour of the system, which is modeled similarly as in Section 2.2. In the linear case, global conditions could mention the state of processes on the right or left of a given process. With tree topologies, it is no longer the case, but we see a closely related notion: pattern matching.

A configuration of the system is represented by a tree over a finite set of local process states $Q$. The behaviour of the system is induced by a set of transitions $\Delta$ conditioned by the local states of neighboring processes, i.e. the parent and children processes. In this topology, a transition is a rewriting rule which may change the states of all involved processes. The arity or the order of the children might be relevant. For example, in Figure 3.8, a process in state $a_1$ can change state if the parent is in state $p_1$ and the children are in state $b_1$ and $c_1$. In such a case, the process changes state to $a_2$ and the parent and the two children change state to $p_2$, $b_2$, $c_2$, respectively. In other words, process $a_1$ fits the pattern of the rewriting rule, when $p_1$, $b_1$, $c_1$ are its neighbors.

The subword relation is replaced by a tree embedding relation. Intuitively, a tree $t$ is embedded in tree $t'$ if it is possible to obtain $t$ by removing nodes from $t$. Removing nodes is a complex operation that re-attaches the subtrees of a node to its parent. The embedding relation is presented in details in paper V.
Figure 3.8. A tree transition $\delta$: If the pattern $p_1, a_1, b_1, c_1$ is found in the tree, the rule is applied and the nodes change their local state to $p_2, a_2, b_2, c_2$ respectively.

We over-approximate the behaviour of the system by modifying the semantics of the transitions, such that a rule is applied to a node and two nodes in its left and right subtrees (rather than its immediate left and right children). Nodes “violating” the pattern are removed, i.e. we “go down in the ordering” before we apply the rule (see Figure 3.9). The resulting abstract transition system is then monotonic with respect to the tree embedding relation on configurations — larger configurations are able to perform the same transitions as smaller ones with results ordered accordingly.

Figure 3.9. An abstract tree transition: the pattern to apply the rule $\hat{\delta}$ is searched in the subtree, rather than the immediate children or parent.

Since the abstract transition relation is monotonic and the tree embedding relation is a WQO (by Kruskal’s theorem \[42\]), it follows that we can apply the backward reachability analysis of Algorithm 2, with the guaranty that it terminates.

Upward-closed sets of configurations are symbolically (and finitely) represented with trees, which allows the reachability analysis to be performed by computing predecessors of trees, simply and efficiently — more than applying transducer relations on general tree regular languages \[9\]. Based on the method, we have implemented a prototype which works well on several tree-based protocols such as the percolate, leader election, tree-arbiter, and the IEEE 1394 Tree identity protocols (see Paper V).
What we learned in Chapter 3

**Upward-closed sets.** Monotonic abstraction relies on a preorder which allows us to symbolically and compactly represent sets of configurations, rather than individual configurations.

**Monotonic systems** enjoy the property that firing a transition on a configuration can always be done on a bigger configuration (w.r.t. the preorder) and the images are ordered accordingly. Moreover, reachability of some sets is decidable for such systems.

**Monotonic Over-Approximation** is used to transform non-monotonic systems into abstract monotonic systems. This allows us to define the pre-image that preserves the upward-closedness of sets.

**Bad configurations** often follow a simple pattern and the set of all bad configurations turn out to be upward-closed.

**Backward Reachability** is interesting to compute the predecessors of the bad set.

**Well-Quasi Orderings** imply that there is no infinite sequence of incomparable elements. Therefore, any upward-closed set can be fully-characterized by a finite set of minimal elements. This allows us to concretely implement a fully-automatic and efficient backward procedure that is guaranteed to terminate.

**Applications.** The technique derives over-approximation for systems arranged as linear topologies and we showed that it can easily be applied to multisets (with the important class of Petri Nets), sequential heap programs manipulating singly-linked lists and that it can be extended to tree topologies.
4. View Abstraction

We recall the definitions introduced in Chapter 2: a parameterized system $P = (Q, \Delta)$ represents the parallel composition of several processes, whose local states are in $Q$, and gives rise to a transition system $(C, \rightarrow)$, where $C$ is a set of configurations and $\rightarrow$ is the post-operator which transforms a configuration into another. The configurations have different sizes and are organized according to some predefined topology, such as arrays, trees, rings and multisets. The reachability problem is to determine, for each size of configurations, whether the set $B$ of bad configurations is reachable from the set $I$ of initial configurations. We use $R$ to denote the set of all reachable configurations from $I$ (and $R_i$ for the set of configurations in $R$ of size $i$). The system is hence characterized by an infinite family of reachable configurations and is proven safe when $R \cap B = \emptyset$. Here, the parameter of the verification problem is the size of the configurations.

We present, in this chapter, another technique to solve the reachability problem. The technique is a forward reachability analysis using yet another abstraction. It has been introduced originally to solve the problem of Shape Analysis (presented in Chapter 5), but it applies surprisingly well to the settings of parameterized systems. To simplify the presentation, we focus, in this chapter only, on parameterized systems with a linear topology. The set of states $Q$ is finite and the transitions from $\Delta$ are size-preserving. Extensions and other topologies can be found in Paper I.

The key insight of the method is to take advantage of the fact that the first instances of the system (i.e. for configurations of small sizes) give enough information to derive the behaviour of the system in general. We can see the small configurations as patterns that will only be repeated and intertwined in configurations of larger sizes, as illustrated in Figure 4.1. Moreover, bad patterns, if existing, are often detected already when only a few processes are involved. Consider, for example, the configurations with six processes, where one process is passive in, say, its initial state. The remaining five processes of such configurations often (but not necessarily) “cover” the configurations based only on five processes.

Figure 4.1. Repeated patterns.
The main idea of our method is to exploit this small model property and perform parameterized verification by only exploring a small number of fixed instances to prove the system safe – for all sizes of configurations. In practice, it is often the case that we only need to compute the finite sets $R_1, R_2$ and $R_3$ to determine whether the whole set $\mathcal{R}$ contains a bad configuration.

The method automatically detects a cut-off point beyond which the verification procedure need not continue. Intuitively, it means that the information already collected during the exploration of the state-space until the cut-off point allows us to conclude safely that no bad configuration will occur in the larger instances. The cut-off detection is performed on-the-fly during the verification procedure itself (and illustrated in Figure 4.2).

![Figure 4.2. Small Model Property: The method detects a cut-off point beyond which the verification procedure need not continue.](image)

The configurations from the first instances of the system are abstracted but retain enough information to “reconstruct” the sets of reachable configurations of larger sizes, as we create larger configurations by combining small patterns. In fact, the collected patterns allow to characterize an over-approximation of the reachable configurations. It is possible that a recombination of patterns creates a configuration that the original system would not have computed. The key is to abstract the small configurations while retaining enough information in order to not over-approximate the set of reachable configurations too much.
Nevertheless, inspecting only small (and finite) instances of the system allows for an efficient method. As usual, if the over-approximation does not contain any bad configuration, we can safely conclude that the system is safe.

The chapter is structured as follows. We first introduce the abstraction at the heart of the method, focusing solely on \emph{atomically} checked global conditions. We then show the method, its soundness, and discuss its completeness. Finally, we present how the method can be extended to cope with non-atomic global conditions and how to handle Szymanski’s mutual exclusion protocol (i.e. the challenging problem of Section 2.5).

4.1 Abstract Domain

The \emph{view abstraction} considers the configurations of the system from the perspective of a few fixed number of processes. The abstraction is parameterized by a constant $k$, and any configuration is intuitively “broken down into pieces” of size (at most) $k$, called \emph{views}. A view retains the information about $k$ processes from a configuration and abstracts away the other processes. There is a certain freedom in what information to retain and what to abstract away. For the $k$ remaining processes, the information could be partial or intact, while the information about the abstracted processes could be fully or partially discarded. This choice defines the level of abstraction that transforms configurations into views.

We first define a simple abstraction: for every $k \in \mathbb{N}$, the $k$ chosen processes are retained intact, while the other abstracted processes are ignored. A view is then a subword of a configuration, using the subword relation $\sqsubseteq$ from Section 3.1. We use $\mathcal{V}$ to denote the set of all views (and $\mathcal{V}_k$ for the set of views of size up to $k$). Since views are here configurations of smaller sizes, we have hence that $\mathcal{V}_k = \{v \in \mathcal{V} \mid |v| \leq k\} \subseteq \{c \in \mathcal{C} \mid |c| \leq k\}$. The abstraction function $\alpha_k : \mathcal{C} \mapsto 2^{\mathcal{V}_k}$ maps a configuration $c$ into the set of all its views (subwords) of size up to $k$:

$$\alpha_k(c) = \{v \in \mathcal{V}_k \mid v \sqsubseteq c\}$$

We lift $\alpha_k$ to sets of configurations as usual. Observe that views resemble configurations of smaller sizes but this does not always need to be the case: they can be complex abstract entities (c.f. Section 4.3).

The concretization function $\gamma_k : 2^{\mathcal{V}_k} \mapsto 2^{\mathcal{C}}$ takes as input a set of views $V \subseteq \mathcal{V}_k$, and returns the set of configurations that can be reconstructed from the views in $V$. In other words,$^1$

$$\gamma_k(V) = \{c \in \mathcal{C} \mid \alpha_k(c) \subseteq V\}$$

$^1$In the field of abstract interpretation, $(\alpha_k, \gamma_k)$ forms a Galois connection (see Paper III and VII).
Figure 4.3. Consider the configurations on the left, theirs views in the middle and their concretization on the right. The concretization contains the original set of configurations, but also the extra configuration $\{1\ 2\ 4\}$, i.e. this abstraction is an over-approximation.

In general, the set of views collectively represent a set of reachable configurations. There is an important subtlety here to catch: No information about configurations is necessarily distorted or ignored in this abstraction, the configurations are merely scattered across views. If the information about a given process is ignored in one view, it will necessarily appear in another view. It is an over-approximation because the views can reconstruct the configurations they emerge from, but they could also be recombined to form new configurations that were not part of the original system, for any size, as illustrated in the example from Figure 4.3.

It is important to observe the precision of the set $\gamma_k(V)$. As we just mentioned, the views work collectively to represent the set of configurations. If a view, say $\{1\ 2\}$, is recombined into the configuration $\{1\ 2\ 3\}$ in $\gamma_2(V)$, it must be the case that the views $\{1\ 3\}$ and $\{2\ 3\}$ were also present in $V$. The correlation between the $k$ processes appearing in a view increases the precision of the abstraction. In other words, we get more precise recombinations with larger $k$, i.e. formally, for a set of configurations $X \subseteq \mathcal{C}$,

$$\gamma_1(\alpha_1(X)) \supseteq \gamma_2(\alpha_2(X)) \supseteq \gamma_3(\alpha_3(X)) \supseteq \cdots \supseteq X$$

For a set of configurations $X \subseteq \mathcal{C}$, we define the post-image of $X$ as the set $\text{post}(X) = \{ \delta(c, i) \mid c \in X, i \leq |c|, \delta \in \Delta \} = \{c' \mid c \in X, c \rightarrow c' \}$. The abstract post-image of a set of views $V \subseteq \mathcal{V}_k$ is defined, as usual, as the composition $\text{Apost}_k(V) = \alpha_k(\text{post}(\gamma_k(V)))$.

4.2 Method

Now that we have defined the abstraction function and how to jump between sets of configurations and sets of views, we are ready to describe the procedure that manipulates views. The procedure is a forward analysis, composed of two nested loops. The inner-loop explores if the initial configurations can reach the
set \( \mathcal{B} \) of bad configurations, in the abstract domain using views of size up to \( k \). The outer-loop determines a cut-off point \( K \) such that the views \( V \subseteq \mathcal{V}_K \) of size \( K \), computed by the inner-loop, satisfy the following properties:

(i) \( V \) is an invariant for all instances, i.e. \( \mathcal{R} \subseteq \gamma_K(V) \) and \( \text{Apost}_K(V) \subseteq V \)

(ii) \( V \) is sufficient to prove safety, i.e. \( \gamma_K(V) \cap \mathcal{B} = \emptyset \).

Point (i) expresses that we have reached a set of views that collectively over-approximates the set \( \mathcal{R} \) of all reachable configurations, and that we cannot get new views by applying the abstract post-image. So the set \( V \) is “stable” and point (ii) states that none of the configurations represented by \( V \) are bad, so the system is safe.

The outer-loop on line 1 searches for a suitable \( k \). The procedure starts by computing (on line 2) the reachable configurations of size \( k \), concretely (i.e. without abstraction), denoted \( \mathcal{R}_k \). Formally, \( \mathcal{R}_k = \text{post}^*(I_k) \) where \( I_k \) is the set of initial configurations of size \( k \) and \( \text{post}^* \) is the repeated application of the (concrete) post-image. If a bad configuration is discovered during that step, the system is surely not safe, and we can even pinpoint a counter-example. Otherwise, the procedure continues to the inner-loop, a fixpoint computation (line 3) of a set of views that at least covers the views of size \( k \) generated from the initial configurations. The inner-loop consists of computing the recombinations from the so-far collected set of views, and detecting if the abstract post-image generates new views. In such a case, the procedure loops and starts again from the new set of views. If not, the fixpoint is reached and no new views will be detected.

Assume that the inner-loop on line 3 reaches a fixpoint. The set \( V \), by construction at the end of the inner-loop, contains the views of \( \mathcal{I} \) and is stable under the abstract post-image, i.e. for some \( k \), we have both \( \alpha_k(\mathcal{I}) \subseteq V \) and \( \text{Apost}_k(V) \subseteq V \). It is not hard to derive that this set covers in fact all the views of size up to \( k \).

\[ V := \mu X . \alpha_k(\mathcal{I}) \cup \text{Apost}_k(X) \]

The Algorithm 3 as a diagram.

---

**Algorithm 3: Verification Scheme**

1. for \( k := 1 \) to \( \infty \) do
2. if \( \mathcal{R}_k \cap \mathcal{B} \neq \emptyset \) then return Unsafe
3. \( V := \mu X . \alpha_k(\mathcal{I}) \cup \text{Apost}_k(X) \)
4. if \( \gamma_k(V) \cap \mathcal{B} = \emptyset \) then return Safe

---

\[ \text{Figure 4.4. Algorithm 3 as a diagram.} \]

---

\[ ^2 \text{We consider here transitions that do not change the size of a configuration. Paper I presents some extensions to deal with non size-preserving transitions.} \]
views of $\mathcal{R}$, that is $\alpha_k(\mathcal{R}) \subseteq V$ and the set $V$ fulfills point (i). The proof is in paper VII.

The cut-off condition is tested on line 5. There are two outcomes:

(a) if the test fails, we don’t know whether the system is indeed unsafe or whether the abstraction introduced a spurious behaviour. The procedure increases $k$, hence the precision of the abstraction, and reiterates the outer-loop (line 1),

(b) the test succeeds so the computed set $V$ fulfills point (ii) (and point (i) by the fixpoint computation) so the system is then safe.

**Implementation**

Algorithm 3 is only a scheme and computing $Apost_k(V)$ is a central component of the verification procedure. It cannot be computed in a straightforward manner because the set $\gamma_k(V)$ is in general infinite and therefore hardly representable. However, we can easily see that applying the abstraction function on the post-image of the configurations, which are the (potentially infinite) re-constructions $\gamma_k(V)$ from some set of views $V$, will mostly return the same views that were already in $V$, and generate a few new ones. Indeed, for each configuration, only a “small” part changes, the other parts will be abstracted into the same views. This is why it is interesting to enable all transitions on views directly, as if they were performed on configurations, effectively removing the need to re-construct the full configurations.

Going in that direction, we must notice that the global condition of a transition can mention some process states that have been abstracted away, making the transition disabled. In order to enable those transitions, we try to extend the views such that they would encompass the missing witness processes, necessary to perform the transitions. In fact, we show that it is sufficient to consider only the configurations in $\gamma_k(V)$ with size up to $k + 1$ (that is, extensions with only one extra witness). There are finitely many such configurations, and hence their post-image can be computed. Formally, for $\ell \geq 0$, we define

$$\bigcup_{k}^{\ell} (V) := \{ c \in \mathcal{C} \mid \alpha_k(c) \subseteq X, \ |c| \leq \ell \}$$

and we have proven that the set of views $\alpha_k(post(\gamma_k(V))) \cup V$ can be instead computed using the following (finite) set:

$$\alpha_k(post(\gamma_k(V))) \cup V = \alpha_k(post(\bigcup_{k}^{k+1} (V))) \cup V$$

Using the previous equation, we can alleviate the problem of reconstructing the (potentially infinite) set of configurations $\gamma_k(V)$ in the abstract post computations. Instead, it is enough to extend the views $V$ of size $k$ in a consistent way, yielding a finite set, and apply the transitions on those extended views.
The union on both side of the equation handles the case where a transition is potentially disabled. The full proof can be found in paper I.

Moreover, it is the case that the transitions have small preconditions, that is, the views don’t need to be extended by much in order for transitions to be enabled. Depending on the transition, we can hence get away with a configuration that is only a slightly extended version of some view (here, extended with one extra witness process, see Figure 4.5).

Figure 4.5. Consider the configuration on the top and a view \( v \) (marked with a black border). The transition \( \text{if } \exists j > i : \{ w \} \text{ then } s \rightarrow t \) is disabled on the view as-is, but it is enabled in case the view is extended with the state \( w \) and gives then the view \( v' \).

To implement an effective procedure, the following steps are required (marked with a red wave in diagram 4.4):

1. **Computing the abstraction** \( \alpha_k(\mathcal{I}) \) **of initial configurations.** This is not a difficult step, since \( \mathcal{I} \) is usually a (very simple) regular set, and \( \alpha_k(\mathcal{I}) \) is easily computed using a straightforward automata construction. For instance, in the case of Szymanski’s protocol from Section 2.5, all processes are initially in state \( 0 \), hence \( \alpha_k(\mathcal{I}) \) contains only the words of size \( l \leq k \) consisting of only the letter \( 0 \).

2. **Computing the abstract post-image.** We circumvent the potential infinite set \( \gamma_k(\mathcal{V}) \) and only consider the abstract post-image of the configurations from the finite set \( \phi_k^{k+1}(\mathcal{V}) \). This is the main reason that makes the method so efficient, together with the small precondition property from the transitions.

3. **Evaluating the test** \( \gamma_k(\mathcal{V}) \cap \mathcal{B} = \emptyset \). Since \( \mathcal{B} \) is usually the upward-closure of a finite set \( \mathcal{B}_{\text{min}} \), the test can be carried out by testing whether there is \( b \in \mathcal{B}_{\text{min}} \) such that \( \alpha_k(b) \subseteq \mathcal{V} \). For instance, in the case of Szymanski’s protocol from Section 2.5 (for \( k = 2 \)), all the bad configurations follow the same patterns: \( \ldots 9 \ldots 9 \ldots \), \( \ldots 9 \ldots 10 \ldots \), \( \ldots 10 \ldots 9 \ldots \) or \( \ldots 10 \ldots 10 \ldots \). This means that they must contain the views \( 9 \ 9 \), \( 10 \ 9 \), \( 9 \ 10 \) or \( 10 \ 10 \). We check whether any latter view is included in the set of views from the fixpoint.

4. **Exact reachability analysis.** Line 2 requires the computation of \( \mathcal{R}_k \). Since \( \mathcal{R}_k \) is finite, it can be computed with any (fast) procedure for exact exploration of finite-state systems.
The implementation presented on the right fulfills the above requirements. Notice a simple acceleration in that procedure, based on the observation that we can seed the fixpoint computation (on line 3) with a larger set than $\alpha_k(I)$, namely $\alpha_k(R_k)$. The effect of this acceleration is that most experiments happen to be (almost already) at fixpoint, using $\alpha_k(R_k)$ as initial input. This demonstrates the efficiency of the method and that most behaviours are captured by small instances of the system. The test performed on line 2 is carried out using the minimal elements $B_{\text{min}}$ of the set $B$ of bad configurations, such that we still compute $R_k$ but only check if $\alpha_k(R_k)$ contains a bad “view” from $B_{\text{min}}$.

### Termination and Completeness

We have shown so far that the method is in fact sound, i.e. when it returns that the system is safe, it is indeed the case. We show in Paper I and Paper VII that the method is also complete for a large class of systems, namely those that are monotonic with respect to a well-quasi ordering (see Chapter 3). Completeness states that if the system is safe, the method will show that it is. In our case, if the system is WQO and safe, there exists $k$ such that the test on line 4 succeeds. If the system is not safe, there is surely a $k$ such that $R_k$ exhibits the error. Consequently, for WQO systems, the method is guaranteed to terminate.

### 4.3 Contexts

The simple abstraction presented in Section 4.1 allows us to compute invariants that are downward-closed with respect to the subword relation $\sqsubseteq$. Several methods [22, 53, 21, 11, 14, 50, 9, 35, 52] have been devised to take advantage from such invariants, but there are several classes of systems that are beyond their applicability. The reason is that such systems do not allow good downward-closed invariants, and hence over-approximating the set of reachable configurations by downward-closed sets will give false counter-examples.

Let us briefly explain where the problem comes from. A deeper explanation can be found in Paper III. Consider for example the view $ab$. It emerges from some configurations that contain the former view as a subword. It might be the case that all those configurations do also contain $d$ on the right of $b$ (i.e. all the configurations contain in fact the subword $abd$). In such a case, the transition if $\forall j > i : \neg\{d\}$ then $a \rightarrow c$ will be enabled on the view $ab$, but it would not have been enabled on any of the configurations that contain that view as a subword. The problem is now that the new view $cb$ is potentially harmful in the sense that it can lead to a counter-example. Notice moreover
that no extension to views of larger size will always disable the transition on those views (i.e. increasing the precision will not eliminate the problem).

In this section, we introduce a new type of view and adapt the method from Section 4.2 to target a class of invariants which are needed in many practical cases and cannot be expressed as downward-closed sets.

4.3.1 Context-Sensitive Views

A context-sensitive view (henceforth only called view) is a pair \((b_1 \ldots b_k, R_0 \ldots R_k)\), often written as \(R_0 b_1 R_1 \ldots b_k R_k\), where \(b_1 \ldots b_k\) is a configuration and \(R_0 \ldots R_k\) is a context, such that \(R_i \subseteq Q\) for all \(i \in \llbracket 0, k \rrbracket\). We call the configuration \(b_1 \ldots b_k\) the base of the view where \(k\) is its size and we call the set \(R_i\) the \(i^{th}\) context. We use \(\mathcal{V}_k\) to denote the set of views of size up to \(k\). For \(k, n \in \mathbb{N}, k \leq n\), let \(H^k_n\) be the set of strictly increasing injections \(h: \llbracket 0, k + 1 \rrbracket \to \llbracket 0, n + 1 \rrbracket\), i.e. \(1 \leq i < j \leq k \Rightarrow 1 \leq h(i) < h(j) \leq n\). Moreover, we require that \(h(0) = 0\) and \(h(k + 1) = n + 1\).

We define the projection of a configuration. For \(h \in H^k_n\) and a configuration \(c = q_1 \ldots q_n\), we use \(\Pi_h(c)\) to denote the view \(v = R_0 b_1 R_1 \ldots b_k R_k\), obtained in the following way (depicted on the right):

(i) \(b_i = q_{h(i)}\) for \(i \in \llbracket 1, k \rrbracket\),
(ii) \(R_i = \{q_j \mid h(i) < j < h(i + 1)\}\) for \(i \in \llbracket 0, k \rrbracket\).

Intuitively, respecting the order, \(k\) elements of \(c\) are retained as the base of \(v\), while all other elements are collected into contexts as sets in the appropriate positions.

We also define projections of views (depicted on the right). For a view \(v = R_0 b_1 R_1 \ldots b_n R_n\) and \(h \in H^k_n\), we overload the notation from the projection of configurations and use \(\Pi_h(v)\) to denote the view \(v' = R'_0 b'_1 R'_1 \ldots b'_k R'_k\), such that:

(i) \(b'_i = b_{h(i)}\) for \(i \in \llbracket 1, k \rrbracket\) and
(ii) \(R'_i = \{b_j \mid h(i) < j < h(i + 1)\} \cup (\bigcup_{h(i) \leq j < h(i + 1)} R_j)\) for all \(i \in \llbracket 0, k \rrbracket\).

We define an entailment relation on views of the same size. Let \(u = R_0 b_1 R_1, \ldots, b_n R_n\) and \(v = R'_0 b'_1 R'_1, \ldots, b'_n R'_n\) be views of the same size \(n\). We say that \(v\) entails \(u\) or that \(u\) is weaker than \(v\), denoted \(u \preceq v\), if \(b_1 \cdots b_n = b'_1 \cdots b'_n\) and \(R_i \subseteq R'_i\) for all \(i \in \llbracket 0, n \rrbracket\). Views of different sizes are not comparable. For two sets \(V\) and \(W\) of views, we write \(V \preceq W\) if every \(w \in W\) entails some \(v \in V\). Formally, \(V \preceq W \iff \forall w \in W, \exists v \in V, v \preceq w\). We use \(|V|\) to denote the set of views in \(V\) that are weakest, i.e. minimal w.r.t. \(\preceq\). We use \(V \sqcup W\) to denote the set \([V \cup W]\).
We are now ready to define the abstraction and concretization functions using context-sensitive views as symbolic encoding. Let \( k \in \mathbb{N} \). The abstraction function \( \alpha_k \) maps \( x \), a view or a configuration, into the set of its projections of size \( k \) or smaller:

\[
\alpha_k(x) = \{ \Pi_h(x) \mid h \in H_{|x|}^{\ell}, \ell \leq \min(k, |x|) \}
\]

For a set \( X \) of views or of configurations, we define \( \alpha_k(X) \) as the set of its weakest projections \( \bigcup_{x \in X} \alpha_k(x) \). The concretization function \( \gamma_k \) maps a set of views \( V \subseteq \mathcal{V}_k \) into the set of configurations

\[
\gamma_k(V) = \{ c \in C \mid V \preceq \alpha_k(c) \}
\]

### 4.3.2 Verification Procedure and Approximation

The verification procedure from Algorithm 4 must be adapted to cope with contexts. The procedure will still take advantage of extensions, but the latter are no longer configurations of smaller size. They are now equipped with a context, and the concrete post from Section 4.1 cannot directly be used on them. Moreover, since \( \gamma_k(V) \) is in general infinite, we need to compute the abstract post-image symbolically.

Although it is possible to compute the abstract transformer precisely (see Paper III), we introduce an over-approximation for efficiency reasons and show that the resulting procedure is sound and complete.

#### Symbolic Post Operator

To define the symbolic post operator, we first define a transition relation on views. For a view \( v = (\text{base}, \text{ctx}), i \leq |\text{base}| \), and a transition \( \delta \in \Delta \), we define the symbolic immediate successor of \( v \) under a \( \delta \)-move of the \( i \)th process from \( \text{base} \), denoted \( \delta^\#(v, i) \). Informally, the moving process checks the other processes from the base. In addition, if \( \delta \) is a universal transition, the moving process checks as well the processes in the contexts. If the transition is enabled, the moving process from \( \text{base} \) changes its state according to the \( \delta \)-transition, otherwise it is blocked. The contexts do not change. In fact, we can here observe the role played by a context: it retains enough information in a view to disable (or block) universal transitions, which would have been otherwise enabled without the presence of contexts. This reduces the risk of running a too coarse over-approximation.

Formally, for a view \( v = R_0b_1R_1\ldots b_nR_n \) and \( i \leq n \), we have that \( \delta^\#(v, i) = R_0b_1'R_1\ldots b_n'R_n \) iff \( b_i = \text{src}, b_i' = \text{dst}, b_j = b_j' \) for all \( j : j \neq i \) and either

(i) \( \delta \) is a local rule \( \text{src} \rightarrow \text{dst} \), or

(ii) \( \delta \) is a global rule of the form \( \text{if } Q \sim j : i \text{ then } \text{src} \rightarrow \text{dst} \), and one of the following two conditions is satisfied:
(a) $Q = \forall$ and it holds both that $b_j \in S$ for all $j \in [1,n]$ such that $j \sim i$ and that $R_j \subseteq S$ for all $j \in [0,n]$ such that $j \sim i$, or

(b) $Q = \exists$ and there exists $j \in [1,n]$ such that $j \sim i$ and $b_j \in S$.

Note that we do not need to check the contexts in the latter case. Indeed, this is supported by the fact that the views work collectively. If there is a view where a process appears in a context, then there is always another view where it appears in the base, while the others are in a context. Finally, for a set of views $V$, we define $\text{spost}(V) = \{ \delta \#(v, i) \mid v \in V, i \leq |v|, \delta \in \Delta \}$.

We now explain how we define the symbolic post operating on views. It is based on the observation that a process needs at most one other process as a witness in order to perform its transition (cf. an existential transition as illustrated in Figure 4.5). A moving process can appear either (i) in the base of a view, or (ii) in a context. Extending adequately the view with one extra process is enough to determine whether the moving process, in case (i), can perform its transition. However, in case (ii), since $\text{spost}$ only updates processes of the base, a first extension “materializes” a moving process into the base and a second extension considers its witness. Therefore, it is sufficient to extend the views with two extra processes to determine if a transition is enabled, whether the moving process belongs to the base or a context of a view. Formally, for a set $V$ of views of size $k$ and for $\ell > k$, we define the extensions of $V$ of size $\ell$ as the set of views $\mathcal{g}_k^\ell(V) = \alpha_l(\gamma_k(V))$. Finally, we define the symbolic post as $\alpha_k \circ \text{spost} \circ \mathcal{g}_k^{k+2}(V)$. Similarly to the equation on page 64, we have shown in Paper III that the symbolic post is the best abstract transformer, i.e. for any $k$ and set of views $V$ of size up to $k$, it holds that

$$V \cup \alpha_k \circ \text{post} \circ \gamma_k(V) = V \cup \alpha_k \circ \text{spost} \circ \mathcal{g}_k^{k+2}(V)$$

**Approximation**

The computation of the above exact symbolic post comes at some cost, mostly due to the computation of $\mathcal{g}_k^\ell(V)$. We therefore introduce an over-approximation and compute the set

$$\mathcal{f}_k^\ell(V) = \{ v \in V \mid \alpha_k(v) \supseteq V, |v| \leq \ell \}$$

i.e. the set of views of size $\ell$ that can be generated from $V$, without inspecting its concretization first. Views in $\mathcal{f}_k^\ell(V)$ have (at least) the same bases as the views in $\mathcal{g}_k^\ell(V)$, but they might have smaller contexts (and are therefore weaker). As a consequence, they might enable more (universal) transitions than their counterparts in $\mathcal{g}_k^\ell(V)$. Consider for example the case where $k = 2$, $\ell = 3$ and the set of views $V = \{ ab, bc, ac[e], ce[f], ae, be, af, bf, cf, ef \}$ — we write contexts in brackets, and ignore the empty contexts for brevity. The set $\mathcal{g}_2^3(V)$ contains the view $abc[e]$ but $\mathcal{f}_2^3(V)$ contains the view $abc[e, f]$ because the
smallest configuration in $\gamma_2(V)$ that has $abc$ as a subword is $abcef$ (this is due to the view $ce[f]$ which enforces the presence of $f$). Another example is $V = \{ab, bc, ac[e], a[e]c, [a]ce\}$. Here, $\mathcal{G}_2^3(V)$ contains $abc[e]$, however, there is no view with the base $abc$ in $\mathcal{G}_2^3(V)$ since there is no configuration with the subword $abc$ in $\gamma_2(V)$.

We show in Paper III, that it is an over-approximation, i.e. for any $\ell \geq k$ and $V \subseteq \mathcal{V}$, it holds that

$$\mathcal{G}_k^\ell (V) \preceq \mathcal{G}_k^\ell (V)$$

### Sound and Complete algorithm

We combine the fixpoint computation of the symbolic post with a systematic state-space exploration in order to find a bad configuration.

The algorithm (described succintly in Algorithm 5) proceeds by iteration over configurations and views of size up to $k$, starting from $k = 1$ and increasing $k$ after each iteration. Every iteration consists in two computations in parallel:

(a) Using the exact post-image, we compute the set $R_k$ of configurations reachable from the initial configurations, involving only configurations of size $k$ (line 2). Note that there are only finitely many such configurations and that we consider length-preserving transitions, so this step terminates and

(b) the fixpoint computation of the symbolic post over views of size up to $k$. A reachable bad configuration of some size must be reachable through a sequence of transitions involving configurations of some maximal size, so it will be eventually discovered. In the fixpoint computation of the symbolic post (line 3), it is sound to replace $\mathcal{G}_k^{k+2}$ with the over-approximation $\mathcal{G}_k^{k+2}$.

Finally, the termination criteria on line 2 and 4 require the use of the function $\text{bad}$, which returns whether a set of configurations contains a bad configuration or whether a set of views characterizes a bad configuration, depending on the type of its input parameter. If its input parameter is a set $X$ of configurations, the function $\text{bad}$ is implemented by checking whether any configuration from $B_{\min}$ can be a subword of some configuration in $X$. If its input parameter is a set $V$ of views, the function $\text{bad}$ is implemented by checking whether an element of $B_{\min}$ can appear within the base of a view from $V$. We do not inspect any context, because the views work collectively and there is always another view in the set which contains this context element in its base.

The resulting verification algorithm is sound and terminates for some $k$ if and only if there is a reachable bad configuration or if there is a good invariant (of a specific form which expresses more than downward-closed properties —
see Paper III). It uses the property of small models, that is, most behaviors are captured with small instances of the systems, either in the form of configurations and views.

**Acceleration**

In a similar manner to the procedure from 4.2, the fixpoint computation on line 3 can be accelerated by leveraging the entailment relation. It is based on the observation that $R_k$ contains configurations of size up to $k$, which can be used as initial input for the fixpoint computation (rather than $I$). All views of size $k$ in $\alpha_k(R_k)$ have empty contexts (i.e. they are weakest). They avoid the computations of the symbolic post on any stronger views. A similar argument can be used to see that it is not necessary to apply $spost$ on the views in $\alpha_k^{k+2}(X)$ that are stronger than the views in $\alpha_k(R_k)$. We therefore seed the fixpoint computation with a larger set than $\alpha_k(I)$, namely $\alpha_k(R_k \cup R_{k+1} \cup R_{k+2})$, and cache the set of views $\alpha_k^{k+2}(R_k)$. 

**4.4 Dropping the Atomicity Assumption**

We present an extension of our method to handle parameterized systems where global conditions are *not* checked atomically. It is necessary for the model to keep track of intermediate configurations when a non-atomic global condition is evaluated at the same time as another transition, potentially also guarded by a global condition. We use the model presented in Section 2.3: both existentially and universally guarded transitions are replaced with a variant of a for-loop rule of the form: if $\textbf{foreach } j \sim i : S \textbf{ then src } \rightarrow \textbf{ dst else src } \rightarrow e$ where $e \in Q$ is the *escape* state. We recall that, for a configuration with linear topology, in that model, a process inspects the states of the other processes in-order. Therefore, it is sufficient to only keep track of the last position that each process has inspected (using the total map $\triangledown : [1,n] \rightarrow [0,n]$). It follows that, for every process $i$, lower positions than $\triangledown(i)$ (in the available range $\sim$) are then also inspected, while higher positions are not yet inspected.$^{3}$

In order to instantiate an abstract domain, we need to handle the fact that a context is a set and does not reflect which processes have been inspected by another given process. We extend the (context-sensitive) views with some extra information. A view is now of the form $(R_0 q_1 R_1 \ldots q_n R_n, \triangledown, \rho)$, where $(q_1 \ldots q_n, \triangledown)$ is a configuration called the *base*, and $(R_0, \ldots, R_n, \rho)$ is a context, such that $R_0, \ldots, R_n \subseteq Q$ and $\rho : [1,n] \rightarrow 2^Q$ is a total map which assigns a subset of $Q$ to every $i \in [1,n]$. Intuitively, the role of $\rho(i)$ is to keep

$^{3}$In the case processes do not loop in-order, $\triangledown$ is replaced with a binary relation $R \subseteq [1,n] \times [1,n]$ on positions, initially empty. When process $i$ inspects process $j$, the pair $(i,j)$ is added to the relation. We say that $i$ ticks $j$. This can be implemented with a matrix of size $n \times n$ of boolean values and allows us to cover the case where processes inspect each other in a random order.
track of the processes that process $i$ has not yet inspected in case they get mixed up in a context with other already inspected processes. This will be the case, as depicted in Figure 4.6, for one context only, say $R_ℓ$ (in fact, $R_ℓ$ is the context where $✓(i)$ is projected to). It is obvious that contexts of higher (resp. lower) indices than $ℓ$ contain processes that are not (resp. are) inspected by process $i$.

In order to use Algorithm 5 with the new abstract domain (i.e. abstraction and concretization), we need in fact only to adjust the notion of projections, the entailment relation and the symbolic post on views. The procedure will then run as in the previous section.

The projection of a configuration into a view is defined in a similar manner as in Section 4.2. For $h \in H_n^k, k \leq n$, and a configuration $c = (q_1 \cdots q_n, ✓)$, $\Pi_h(c) = (\Pi_h(q_1 \cdots q_n), ✓', \rho')$ where ✓' and ρ' are defined as follows. For all $i \in \llbracket 1, k \rrbracket$, there exists $ℓ$ such that $h(ℓ) \leq ✓(i) < h(ℓ + 1)$. Then, $✓'(i) = ℓ$ and $ρ'(i) = \{q_j | ✓(i) < j < h(ℓ + 1)\}$. The projection of views is defined analogously. Note that this definition also implies that the concretization of a set of views is precise enough and reconstructs configurations with in-order ticks.

The entailment relation between the views $v = (R_0 q_1 R_1 \ldots q_n R_n, ✓, ρ)$ and $v' = (R'_0 q'_1 R'_1 \ldots q'_n R'_n, ✓', ρ')$ (of the same size) is defined such that $v \preceq v'$ iff

(i) both have the same base, i.e. $(q_1 \cdots q_n, ✓) = (q'_1 \cdots q'_n, ✓')$,

(ii) $R_i \subseteq R'_i$ for all $i \in \llbracket 0, n \rrbracket$, and

(iii) $ρ(i) \subseteq ρ'(i)$ for all $i \in \llbracket 1, n \rrbracket$.

This intuitively reflects that the more unticked states within a context the likelier it is for a transition to be blocked, and the larger contexts are the likelier they retain non-ticked states.

Since we do not maintain any order in the contexts, we cannot make a particular view reflect an intermediate state of an in-order for-loop rule. However, recall that views work collectively, so there will be another view distinguishing that intermediate state. Therefore, the symbolic post $spost$ is adapted from the previous post operators, with the particularity that it “ticks” each context at once as a block and handles the extra $ρ$ information adequately. Intuitively, process $i$ inspects its “next” context by “ticking” the elements of $ρ(i)$ unless it has to escape. If it then cannot escape, it moves on to the following position and

![Figure 4.6. Projection with nonatomicity. The blue states have been inspected by process $i$, the green states have not. The abstraction needs to distinguish for that context which states have been inspected from those that have not.](image-url)
marks the following context as unticked (the new value of $\rho(i)$). Notice that the inspected context might contain process states that would make the transition escape (but not $\rho(i)$). It means that those processes potentially changed state after they got ticked. Finally, process $i$ terminates if there is no more context or base element to inspect.

Formally, we fix a view $v = (R_0b_1R_1\ldots b_nR_n, \checkmark, \rho)$ of size $n$, a position $i \in [1,n]$ and a global transition $\delta = \text{if } \text{foreach } j \sim i : S \text{ then } \text{src} \rightarrow \text{dst} \text{ else } \text{src} \rightarrow e$ from $\Delta$ (since the case of a local transition is trivial). As before, we distinguish three types of symbolic $\delta^\#$-move on $v$ by the process in the base at position $i$ (See Figure 4.7): (a) $\delta^\#_i(v,i)$ for a loop iteration, (b) $\delta^\#_e(v,i)$ for escaping and (c) $\delta^\#_t(v,i)$ for termination. Each type of move is defined only if $b_i = \text{src}$.

**Figure 4.7.** Non-atomic “ticking” for the global transition if foreach $j \neq i : \neg\{1,2\}$ then $3 \rightarrow 7$ else $3 \rightarrow 4$. Note that contexts are ticked at once.

Recall that $\checkmark(i)$ represents the position that process $i$ has last inspected, which is either 0 at the start or always points to an element of the base.

(a) Carefully looking at the indices in the view, process $i$ now inspects the processes of the first context that it has not inspected, i.e. the context $R_{\checkmark}(i)$. $\delta^\#_i(c,i)$ is defined if both the following two properties are satisfied: (i) $\rho(i) \subseteq S$ and (ii) $\checkmark(i) + 1 \sim i, \checkmark(i) + 1 \leq n$ and $b_{\checkmark(i)+1} \in S$. It is then obtained from $v$ by only updating $\checkmark(i)$ to $\checkmark(i) + 1$ and resetting $\rho(i)$ to $R_{\checkmark(i)+1}$.

(b) $\delta^\#_e(c,i)$ is defined if one of the following two properties is satisfied: (i) $\rho(i) \not\subseteq S$ or (ii) $\checkmark(i) + 1 \sim i, \checkmark(i) + 1 \leq n$ and $b_{\checkmark(i)+1} \not\in S$. It is obtained from $v$ by changing the state of the process $i$ to $e$ and resetting $\checkmark(i)$ to 0 and $\rho(i)$ to $\emptyset$. Intuitively, process $i$ has found a reason to escape.

(c) $\delta^\#_t(c,i)$ is defined if $\checkmark(i) + 1 \not\sim i$ or $\checkmark(i) + 1 > n$. It is obtained from $v$ by changing the state of the process $i$ to dst and resetting $\checkmark(i)$ to 0 and $\rho(i)$ to $\emptyset$. Intuitively, process $i$ has reached the end of the iteration and terminates its transition (i.e. moves to its target state).

This concludes how we adapted Algorithm 5 to cope with non-atomically checked global conditions in the presence of contexts. For further details, refer to Paper I, III and VII.
What we learned in Chapter 4

A solution to the reachability problem from Section 2.4.
Abstraction and Concretization functions define how to travel between sets of views and sets of configurations. An important notion to retain is that views work collectively to characterize configurations.

Verification Procedure is composed of two nested loops, one of which is a simple fixpoint. The other loop searches for a cut-off point.

Soundness. The method computes an invariant that covers the reachable configurations of any size, using views of small sizes.

Completeness. The method is complete for WQO and for almost downward-closed invariants.

Approximation is introduced in order to leverage the entailment on views and makes it easier to compute.

Acceleration is achieved by seeding the fixpoint computation with more views.

Requirements. The procedure requires to be able to compute the initial views, test for the characterization of bad configurations (using the upward-closedness of the set of bad configurations and checking for the presence of some “bad” views).

Efficiency. The method has proven to be very efficient as shown in the results (see Paper I and III). It exhibits the small model properties, i.e. most patterns occur in small instances.
5. Shape Analysis

In this chapter, we consider a difficult challenge in software verification, namely to automate its application to algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared heap. Such algorithms are notoriously difficult to get correct and verify, since they often employ fine-grained synchronization and avoid locking wherever possible. This chapter presents an efficient technique to verify that a concurrent implementation of a common data type abstraction, such as a queue or a stack, conforms to a simple abstract specification of its (sequential) functionality. There are several combined challenges to be addressed.

The abstract specification is infinite-state, because the implemented data structure may contain an unbounded number of data values from an infinite domain.

The program is infinite-state in several dimensions:

- it consists of an unbounded number of concurrent threads,
- it uses unbounded dynamically allocated memory, and
- the domain of data values is unbounded.

The program does not rely on automatic garbage collection. It manages memory explicitly.

We present, in the first section of this chapter, the type of programs we consider. In the following sections, we introduce in a stepwise manner how we cope with each of the above challenges. In the last section, we combine the different techniques and present the verification method.

5.1 Program Model

We consider systems consisting of an arbitrary number of concurrently executing threads. Each thread may at any time invoke one of a finite set of methods. Each method declares local variables (including the input parameters of the method) and a method body. In this chapter, we assume that variables are either pointer variables (to heap cells), or data variables (assuming values from an unbounded or infinite domain $\mathbb{D}$). The body is built in the standard way from atomic commands using standard control flow constructs (sequential composition, branching, and loop constructs). Method execution is terminated by
executing a return command, which may return a value. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread which is invoking the corresponding method. We assume that the global variables and the heap are initialized by an initialization method, which is executed once at the beginning of program execution.

Programs manipulate heap cells of type node, each consisting of a val field and a next field, which carry respectively a data value and the address to another heap cell. Atomic commands include assignments between data variables, pointer variables, or fields of cells pointed to by a pointer variable. The command new node() allocates a new structure of type node on the heap, and returns a reference to it. The compare-and-swap command CAS(a, b, c) compares the memory locations a and b. If equal, it also atomically assigns the value c to a and returns TRUE. Otherwise, it leaves a unchanged and returns FALSE. Note that a, b and c can be pointers or variables using the referencing and dereferencing C constructs & and *.

```
struct node {data val, node* next}
node* Top;

0 void push(data d){
1    node* n := new node();
2    n->val := d;
3    n->next := NULL;
4    while(TRUE){
5        node* t := Top;
6        n->next := t;
7        if(CAS(Top, t, n)) break;
8    }
9 }

10 data pop(){
11    while(TRUE){
12        node* t := Top;
13        if(t == NULL) return empty;
14        node* x := t->next;
15        data result := t->val;
16        if(CAS(Top, t, x) ) return result;
17    }
18 }
19 }
```

**Figure 5.1.** Treiber’s non-blocking stack [60].

As an example, Figure 5.1 shows a version of the lock-free concurrent stack by Treiber [60], in a C-like language. The program represents a stack as a null-terminated linked-list from the node pointed to by Top, depicted in Figure 5.2. The global variable Top points to the top of the stack. Initially, the stack is empty and Top is null. Each thread either calls a method push(d) to insert a cell containing the data value d at the top of the stack, or calls a method pop() which returns empty if the stack is empty, and otherwise advances Top, returning the data value stored in the cell at the top of stack. For example, the command CAS(Top, t, n) of the push method on line 7 (while slightly abusing the notation) checks whether the pointers Top and t point to the same location. If so, it also atomically assigns n to Top, so that the node pointed by n becomes the top of the stack. Otherwise, it fails and the method loops (often called a retry-loop scheme).
Challenge \( \copyright \) implies the absence of garbage collection, and the program must handle the cell deallocations using the command `free`. Furthermore, it must handle the ABA problem \([38, 64]\), where a thread takes action as if the data structure was still in state A, even though it had already changed from state A to B. The thread erroneously misses the changes and takes action upon an outdated version of the data structure, potentially compromising the internal structure and losing data. This situation happens in programming language like C, when a thread mistakenly confuses an outdated pointer for a valid one. Avoiding the ABA problem requires additional mechanisms where each pointer is equipped with an additional `age` field, which is incremented whenever the pointer is assigned a new value.\(^1\) Paper II presents such an example: the lock-free concurrent queue by Michael & Scott (without garbage collection —see the description in Section 6.16, page 109).

The verification method presented in this chapter can handle the case of explicit memory management. However, to simplify the presentation, we assume the presence of garbage collection, which ensures that a new cell is fresh, i.e. not accessible by any other thread but the one creating it. This is the case for the example in Figure 5.1.

\(^1\)It also requires special assignment and CAS instructions, capable of handling several word units atomically. Those are not available in all hardware but developers can get away using atomic double-word assignments.
5.2 Specification for a Concurrent Data Structure

In a concurrent program, the methods of the different executing threads can overlap in time. It is no longer the case, as for sequential programs, that every method finishes before another starts, such that all memory accesses can be totally ordered. The latter assumption makes the analysis simpler. Instead, each thread now executes in a particular context, which encompasses the values of the global variables and the heap cells that the current thread can reach (even by following a succession of pointers). Other threads can manipulate this context at any time, by for example changing the value of the global (and therefore shared) variables. If one method call precedes another, then the earlier call must have taken effect before the later call. However, if method calls overlap, then the order in which they take effect is ambiguous. The instructions inside a particular method are totally ordered with respect to each others, but instructions from separate methods might not. They form a partial order, and it is difficult to reason about program execution using partial orders. It is hence tempting to inspect the outcome of each method in isolation.

![Linearizability diagram](image)

*Figure 5.3. Linearizability, where the commit points are marked with ●.*

Linearizability, depicted in Figure 5.3 provides the illusion that each method invocation takes effect instantaneously at some point (called the *linearization point*) between method invocation and return [36]. A linearization point is often a step where the effect of the method becomes visible to other threads.

A program is linearizable if, for any concurrent execution, there is an equivalent execution, in which the methods were sequentially ordered according to the linearization points (depicted below the timeline in Figure 5.3). The latter execution is a linearization of the concurrent execution (and there might exist
several ones). We could then use existing methods to sequentially analyze this linearization. In fact, most of the methods we know of, (such as [36, 61]) first show that the concurrent program is linearizable and then show that the linearized versions conform to a sequential behaviour. According to its definition, a linearization point appears inside the method. However, it does not cover the case where it is located in the code of another method. Neither does it if its location is changing, depending on the execution path.

The method presented in this chapter does not proceed so. It shows directly that a concurrent program conforms to the behaviour of its sequential counterpart. As a side-effect, it also implies that the program is linearizable.

In Chapter 3 and 4, it was sufficient to observe the outcome of each program instruction separately, in order to account for all interleavings. More importantly, at any point of an interleaving, it was enough to look at the configuration itself to determine whether it was bad or not. For heap-manipulating programs, we can no longer inspect the outcome of one particular instruction, and determine if the program computed a bad shape, mainly due to the possible changes in the context of the executing thread. We must remember how the program got to a particular state of the data structure, and we therefore take into account sequences of instructions, called traces, rather than inspecting each instruction individually and separately.

In order to derive the totally ordered execution from a concurrent execution, each method is instrumented to generate a so-called abstract event whenever a commit point is passed. Depending on the execution path, a commit point is identified at some (atomic) instruction where the whole method appears to take effect. When an execution passes a commit point, the abstract event is communicated to an external observer (see Section 5.4), which records the sequences of abstract events from the code execution. The overlapping methods are in effect re-ordered according to those commit points, as if they took place in isolation (see the timeline in Figure 5.3). For the source code of Treiber’s non-blocking stack in the presence of garbage collection (in Figure 5.1), the push method emits an abstract event when it passes a successful CAS operation on line 7. The event corresponds to the fact that some data is pushed onto the stack. For the pop method, the location of its commit point depends whether the stack is empty or not. The test on line 13, when successful, emits that the stack is empty. If not, the commit point is located on a successful CAS operation on line 17. For this event, some data is popped from the stack.

We have now reduced the problem to observing traces of abstract events, gathered from concurrent executions. The behaviours of the system, as a whole, are characterized by the set of all traces of events, which is however potentially infinite since the data domain $D$ might be infinite or unbounded (see challenges $\mathbb{A}$ and $\mathbb{B}$). We first introduce a data-independence argument in order to make the trace specification finite. We then introduce the notion of observer, which essentially separates good traces of events from bad ones. Several observers shall be used to specify the safety property.
5.3 Data Independence

We can notice that data structures such as stacks and queues are merely containers and do not look at data: their execution does not branch in some particular part of the code depending on the data values. All data values are treated equally. For instance, if \( \mathbb{D} = \mathbb{N} \) and if the system can input the values 1,2,3 and output 3,2,1 (i.e. a stack), we can easily see that there is an equivalent behaviour where the system inputs the data values 6,5,4 and outputs 4,5,6. The system can execute in the same way regardless of the data values that it manipulates. Intuitively, data values do not matter, so we might as well rename them. In that previous simple example, we could rename 6 into 1, 5 into 2, and 4 into 3. We would then have two equivalent traces.

Even though we can rename the traces adequately, the set of all traces includes traces where input events can be repeated using the same data value, and is therefore still potentially infinite. However, we can observe that it is possible for any trace to “count” the repeated input values and enumerate them while renaming them, effectively creating a trace where the data values of input events are all distinct. In such a case, we call the latter a differentiated trace. For example, if the system takes 1,2,3,1,2 as input, we could rename the second 1 into 4 and the second 2 into 5. This creates an equivalent trace where all the data values are distinct.

![Differentiated Stack](image-url)

**Figure 5.4.** Stack do not look at data

82
We can now introduce the desired definition: we say that a set of traces (or the program it characterizes) is data independent if and only if (i) it is closed under renaming and (ii) it can be generated by a set of differentiated traces. Detecting data-independence is easily done by syntactically checking that data values are not compared and that uninitialized variables are not used in the source code.

If a data independent system accepts a bad trace, then it will also accept a bad differentiated trace (and vice-versa). In consequence, for a data independent program, to determine whether the safety property is satisfied, it is sufficient to only consider differentiated traces, i.e. executions in which any data value is inserted at most once. Furthermore, by adapting an argument from Wolper [65], we can abstract away the data values, by picking any two of them and flattening all the others values to a third one. In Figure 5.4, the top part represents a system which inputs a sequence of distinct data values and outputs them in reverse order, i.e. a stack. In the bottom part, on the other hand, we abstract the data away and choose blue and red to be the “important” values while another “neutral” value white is not. If the bottom system is safe, so is the top one, and so is the system in general (c.f. Paper II for further details).

5.4 Observers

We show now how to cope with challenge A. The starting point is the automata-theoretic approach to model checking [62], in which programs are specified by automata that accept precisely those executions that violate the intended specification, and verified by showing that these automata never accept when they are composed with the program. The properties to be satisfied by the data structure are:

No Lossiness
It cannot input a value and then return the empty value when it tries to output some value.

No Duplication
It cannot input a value, and output it twice without inputing it in-between.

No Creation
It cannot output a data value if it has not received it first.

Orderedness
It must respect
- the LIFO order, for a stack, or
- the FIFO order, for a queue.

We therefore introduce observers, a special kind of automata, which accept only when the above properties are violated and which use a finite set of variables instantiated with a finite set of abstract data values, using the data
Orderedness

Lossiness

Duplication

Creation

Figure 5.5. Stack observers

induction argument from the previous section. As a result, we can now succinctly specify such data structures, using finite observers with typically less than 3 variables. Those observers capture the fact that input/output involving the white value are ignored while input/output of the red/blue value are recorded. We present, in Figure 5.5, an example of four automata that capture the above properties in the case of a stack.

5.5 Three Degrees of Unboundedness and No Garbage Collection

We can now specify if a property gets violated using the finite observers on an abstract stack using data independence. Thus, we can handle Challenge A and B. However, the memory layout of heap cells (called shapes) that the program can create and manipulate is not bounded in size. To tackle Challenge B, it is necessary to handle the size of those shapes.

Each thread has a finite set of variables and can access the heap cells that are pointed to by one of its variables or a global variable. The heap cell that are not pointed by any variable can be potentially accessed through a succession of next pointers. Otherwise, they are considered garbage. The

|\footnote{In fact, observers are more powerful since observer variables can assume values from an unbounded domain. This allows observers to specify properties that should hold for an infinite number of data values. It is however not necessary here, thanks to data independence.}
idea is to only keep track of important cells. Those are the ones pointed by some variable and the ones containing some important data, i.e. red and blue in the case of Treiber’s stack. Intuitively, the cells not pointed by any variable play a secondary role. There are merely “relays” between the cells that the threads can currently access. We do not need to keep how many there are, we only need to keep the memory layout they form.

We thus adapt a variant of the transitive closure logic by Bingham and Rakamarić [18] for reasoning about heaps with single selectors, to our framework. This formalism, called here shape analysis, tracks reachability properties between pairs of pointer variables. Moreover, we have developed a novel optimization, based on the observation that it suffices to track the possible relations between each pair of pointer variables separately.

We use shape analysis to cope with challenge (B$_2$) but it is still not enough to bound the shapes, since we have an unbounded number of threads. Each thread could have one variable pointing to a different cell and that creates an unbounded shape. To cope with challenge (B$_3$), we try to adapt the successful thread-modular approach [17], which verifies a concurrent program by generating an invariant that correlates the global state with the local state of an arbitrary thread. In other words, it keeps track of the shape that one thread can see, abstracting away (and ignoring) all the other threads. The resulting shape is necessarily of bounded size using the above transitive closure. We can draw a parallel with the view abstraction method from Chapter 4, where the configurations are abstracted into views of size 1 ($k = 1$). The thread-modular approach includes a step where it takes the information about one thread and couples it with the information from another thread, in order to take into account the interference of all the other threads on the first thread. This step is akin to the concretization from the view abstraction, where we extend the views which additional information from other views, in a consistent manner.

However, this leaves us with a problem when $k = 1$: the interfering step would create shapes where threads possibly share some particular cell, even though the concrete system would never produce those shapes. This is a too coarse approximation that leads to erroneous behaviors. In fact, in the case of garbage collection, even though the thread-modular approach only keeps track of one thread, it is possible to get around the problem by including additional information on the shape cells, to reflect whether a cell is shared or not. Thanks to the guarantee that garbage collection provides, when a cell is created, the creating thread is the only one accessing that cell. The cell is fresh and we mark it as such. This flag is lost as soon as the cell becomes accessible by another thread or a global variable. This information allows the interference step to be more precise, and separate the situations where it would otherwise merge together two fresh cells from different threads, making the resulting cell erroneously shared. The thread-modular approach, along with the freshness information, is often enough to carry on the verification.
Nevertheless, it is not sufficient to cope with challenge © and the ABA problem. The generated invariant must be able to express that at most one thread accesses some cell on the global heap. Since this cannot be expressed in general by the thread-modular approach, we need to extend it to generate invariants that correlate the global state with the local states of an arbitrary pair of threads. By correlating two threads, we increase the precision of the abstraction, alleviating the above problem. This is similar to the view abstraction when \( k = 2 \) and the concretization function extends views in a consistent manner.

In Figure 5.6, we show how a concrete shape with three threads can be abstracted into a bounded abstract shape. The figure depicts only one of the shape abstraction that we obtain. There is a combinatorial factor in the choice of which two threads to pick out of, here, the three threads. Indeed, the above abstraction techniques bring the needed precision for verification, at the price of significant state-space explosion, which mainly arises from the fact that, reusing the terminology from Chapter 4, the dynamically allocated heaps are scattered across several views which describe the correlation between pairs of threads. We have therefore developed a novel optimization, where we merge abstracted shapes and manipulate the compound instead of every individual abstracted shapes. Intuitively, merging does not bring a penalizing imprecision due to the structure of the retry-loops. When the thread’s local variables are initially placed, but the global shape changed through the actions of other threads, the current thread will detect it and restart its initialization. Hence, recording precisely where those local variables are is often superfluous information that can be discarded. We touch more on that optimization in the next section.
5.6 Verification procedure

To verify that no trace of the program is accepted by an observer, we form, as in the automata-theoretic approach [62], the cross-product of the program and the observer, synchronizing on abstract events, and check that this cross-product cannot reach a configuration where the observer is in an accepting state. Synchronizing on abstract events means that we instrument the program to communicate with the observer every time it passes a commit point successfully. The observer will then advance state depending on the event that has been communicated.

Using data independence, shape analysis and thread bounding, we characterize all the reachable configurations of the program, from the point of view of two distinct executing threads, with a symbolic encoding. The encoding consists in a combination of several layers of conjunctions and disjunctions, to record pairwise the relationships of the local configurations of the two threads with each other, the relationships of the local variables of a thread with global variables, the observer configuration, and finally the assertions about the heap.

Figure 5.7. Encoding during the analysis and a corresponding shape

For example, in Figure 5.7, we represent the encoding in two parts. The first part is some bookkeeping information, denoted $\sigma$, consisting of the pc of each thread and the observer state. The second part is a matrix representing the relationships between the variables pairwise. A local variable $v$ from thread $n$ is denoted $v_n$. In the running stack example, the variables in concern are the global variable Top, the special term # (i.e. the null constant), the local variables of the two current threads $t_1, x_1, t_2, n_2$, and the cells that contain important data, dictated by the observer, (here red and blue). The precise mathematical definitions can be found in Paper II.
Here, we only give the intuition behind the relationships:

• \( t_a = t_b \): the variables \( t_a \) and \( t_b \) point to the same cell,
• \( t_a \mapsto t_b \): the next field of the cell \( t_a \) points to, and \( t_b \) point to the same cell,
• \( t_a \rightarrowrightarrow t_b \): \( t_b \) points to a cell that can be reached by following a chain of two or more next fields from the cell that \( t_a \) points to,
• \( t_a \nRightarrow t_b \): none of \( t_a = t_b \), \( t_a \mapsto t_b \), \( t_b \mapsto t_a \), \( t_a \rightarrowrightarrow t_b \), or \( t_b \rightarrowrightarrow t_a \) is true.

For any term \( t \), we let \( t = \# \) denotes that \( t \) is null. In Figure 5.7, the relationship between \( t_2 \) and \( n_2 \), denoted \( \pi[t_2, n_2] \), is that they are unrelated.

This encoding is precise enough to represent an abstract shape. However, to combat an obvious state-space explosion problem, we merge these matrix representations into yet another matrix where each cell is a disjunction of relations from the set \{\( =, \mapsto, \leftarrow, \rightarrowrightarrow, \leftrightarrow, \nRightarrow \}\}. We depict for example in Figure 5.8, two abstract shapes, where the threads are (of course) at the same pc, which are merged into one matrix representation. Notice that the matrix representation now also characterizes two other shapes that were not considered in the merge. In other words, this new matrix representation is an over-approximation of the set of concrete shapes it characterizes.

\[ \begin{array}{c}
\begin{array}{c}
\text{Thread 1 (pc: 16)} \quad \text{Thread 2 (pc: 8)} \\
\text{Top} \quad \text{Top} \\
\# \quad \# \\
\text{t} \quad \text{t} \\
x \quad x \\
\text{n} \quad \text{n}
\end{array}
\end{array} \]

\[ \begin{array}{c}
\begin{array}{c}
\text{Figure 5.8. Merging shapes: Each cell is a disjunction over \{\( =, \mapsto, \leftarrow, \rightarrowrightarrow, \leftrightarrow, \nRightarrow \}\}.}
\end{array}
\end{array} \]

We can now describe how those matrix representations will be manipulated and cover the different cases which arise in a concurrent setting. We need to consider two cases: On one hand, given a matrix, one of the two represented and currently executing threads performs a step. On the other hand, we must consider the case where another third and distinct thread interferes and changes the matrix, even when the two represented threads do not perform any step.

In the first scenario, where one of the two represented threads performs a step, we can compute the resulting matrix usually in a straightforward manner. We (i) remove all disjuncts that must be falsified by the step (ii) add all disjuncts that may become true by the step, (iii) saturate the result. The details can be found in Paper II. For example, when the action is a variable assignment from a given thread, the row and column relating that variable are “canceled” and the new information is derived from what it is assigned to. A saturation procedure
determines whether there are new disjuncts to be added or if some old ones are to be voided.

In the case where we need to account for possible interference on the matrix by a third thread, we proceed as follows. We (i) extend the matrix with the interfering thread, (ii) apply the first scenario to compute new matrices and (iii) project away the interfering thread from the resulting matrices. The steps are described in Figure 5.9. On the top left corner, we show the abstract shape to be considered, (but we hide and replace the content of the matrices with dots). In step (i), if a third interfering thread $i_3$ (here in light purple) is to exist in the presence of the two other passive threads $i_1$ and $i_2$ (here in yellow and light pink), then it must be the case that both the shape correlating $i_1$ and $i_3$, and the shape correlating $i_2$ and $i_3$, must exist among the shapes that have been created by the program! If so, we extend the first shape accordingly with the third thread, which might strengthen the information in the larger matrix. Naturally, the $\sigma$ information is extend accordingly too. We then apply a post operation (step (ii)) as in the first scenario, which creates new matrices where the content related to thread $i_1$ and $i_2$ might have changed (shown with stars). Finally, in step (iii), we project away the third thread. The resulting matrices correlate thread $i_1$ and $i_2$ but are the result of a third interfering thread. Notice the similarity with the view extensions from Chapter 4.

Figure 5.9. Interference
The main idea of the method is then to collect *all* possible abstract shapes without sending the observer onto an accepting state. It is implemented using a fixpoint procedure, starting from the shapes that characterize the set of initial configurations of the program. Upon termination, we obtain an invariant of the program which characterizes the configurations of the program from the point of view of two distinct executing threads $i_1$ and $i_2$. The main advantage of the method is that it is a direct approach for verifying that a concurrent program is a linearizable implementation of, for example here, a stack. It consists in checking a few small properties of the algorithm, and is thus suitable for automated verification. Previous approaches typically verified linearizability separately from conformance to a simple abstraction, most often using simulation-based arguments, which are harder to automate than simple property-checking. Moreover, the method can automatically verify concurrent programs that use explicit memory management. This was previously beyond the reach of automatic methods.
What we learned in Chapter 5

**Data Independence** is an argument used when verifying programs that treat data equally. It allows us to rename the data and typically handle the verification task using 3 key values.

**Observers** are automata that allow us to capture concisely the bad behaviours of the data-structures they are paired with.

**Heap Abstraction** is necessary to bound the size of the different shapes that the program can create, without introducing too much imprecision. It uses a shape analysis based on the transitive closure of the reachability relation. We showed that it is enough to record pairwise the reachabilities of program variables.

**Joined Heaps** help combat the state-space explosion problem (from page 27).

**Verification** is performed through a simple forward analysis, with a fixpoint computation. The task consists in checking that the cross-product of the program and an observer never sends the latter onto an accepting state.

**Garbage Collection.** In the absence of garbage collection, programs may suffer from the ABA problem. The method can handle that case, which is one of the contribution of this thesis.

**Linearizability** offers the illusion that methods happen at once. The method does not prove linearizability per say, but it shows directly the conformance of a concurrent program to its sequential specification.

**Combined Challenges.** The method addresses several challenges.

- The specification is infinite-state, because the data domain is unbounded.
- The program is infinite-state in several dimensions since it consists of an unbounded numbers of current threads, it uses unbounded dynamically allocated memory and the data domain is also unbounded.
- It handles the case of explicit memory management.

To the best of our knowledge, no other method can handle those challenges at the same time.
6. Case Studies

6.1 Burns’ mutual exclusion

Burns’ algorithm [44] implements a mutual exclusion protocol and can be modeled as a parameterized system where the processes are arranged in a linear topology. Each process can communicate with and distinguish its neighbors on its right or its left.

The local state of a process ranges over state \( \{1, \ldots, 6\} \) where \( 6 \) represents the critical section. Transitions are guarded with conditions on the states of the neighbors, on the right, the left or both and is enabled if the guard is not violated.

For Process \( i \)

\[
\begin{align*}
1 & \quad \text{flag}[i] := 0 \\
2 & \quad \text{if } \exists j < i : \text{flag}[j] = 1 \text{ then } \text{goto} 1 \\
3 & \quad \text{flag}[i] := 1 \\
4 & \quad \text{if } \exists j < i : \text{flag}[j] = 1 \text{ then } \text{goto} 1 \\
5 & \quad \text{await } \forall j > i : \text{flag}[j] \neq 1 \\
6 & \quad \text{flag}[i] := 0; \quad \text{goto} 1
\end{align*}
\]

Initially, all processes are in state \( 1 \). A bad configuration is detected if two processes or more are in the critical section, i.e. if the array contains at least two processes in state \( 6 \). The transitions are depicted in the following state diagram. The process \( i \) is the current process, \( j \) is another process and \( j \) its state.

6.2 Szymanski’s mutual exclusion

This example is presented in Section 2.5 (on page 40).
6.3 Dijsktra’s mutual exclusion

Dijsktra’s algorithm implements a mutual exclusion protocol and can be modeled as a parameterized system where the processes are arranged in a linear topology. Each process can communicate with its neighbors and check their status.

```c
/* For Process i */
flag[i] := 1
if p ≠ i then
    await flag[p] = 0 then
    p = i
if ∃j ≠ i : flag[j] then goto 1
flag[i] := 0; goto 1;
```

The algorithm is described in the above code listing. It makes use of a pointer, i.e. a variable ranging over process indices. We model this pointer by a local boolean variable p for each process state. p is true iff the pointer points to this current process. When the pointer changes, this information must be passed onto all other processes, which we model as a broadcast transition. Concretely, upon pointer assignment, the current process sets its local variable p to true and simultaneously sets p to false in all other processes.

We denote the state of process as St (resp. Sf) when the process is in state S and the pointer p is true (resp. false). The state S of a process ranges over \{1, ..., 6\} where 6 represents the critical section. Initially, one process is in state 1t and all other processes are in state 1f. A bad configuration is detected when 2 or more processes are in the critical section, i.e. when their state is either 6t or 6f.
6.4 Gribomont-Zenner’s mutual exclusion

This algorithm could be seen as a version of Szymanski’s algorithm 2.4, with transitions that are finer-grained in the sense that tests and assignments are split over different atomic transitions. In this model, the local state of a process ranges over state \{1, \ldots, 13\} where \textcolor{red}{1} is the initial state and \textcolor{red}{12} represents the critical section. Configurations not satisfying mutual exclusion are those where at least two processes are at state \textcolor{red}{12}.

6.5 Parosh’s mutual exclusion

This protocol ensures mutual exclusion between processes. Each process has five local states \{0, 1, 2, 3, 4\} and is initially in state \textcolor{red}{0}.

A process in the critical section is at state \textcolor{red}{4}. The set of bad configurations contains exactly configurations with at least two occurrences of state \textcolor{red}{4}. Processes move from state \textcolor{red}{0} to \textcolor{red}{1}, and then \textcolor{red}{2}. Once the first process is in state \textcolor{red}{2}, it “closes the door” on the processes which are still in \textcolor{red}{0}. They can no longer leave state \textcolor{red}{0} until the door is opened again (when no process is in state \textcolor{red}{2} or \textcolor{red}{3}). Moreover, a process is allowed to cross from state \textcolor{red}{3} to \textcolor{red}{4} only if there is at least one process still in state \textcolor{red}{2} (i.e., the door is still closed on the processes in state \textcolor{red}{0}). This prevents a process to first reach state \textcolor{red}{4} along with a process to its left starting to move from \textcolor{red}{0} all the way to state \textcolor{red}{4} (thus violating mutual exclusion). From the set of processes which have left state \textcolor{red}{0} (and which are now in state \textcolor{red}{1} or \textcolor{red}{2}), the leftmost process has the highest priority and it is encoded in the global condition: a process may move from \textcolor{red}{2} to \textcolor{red}{3} only if all processes on its left are in state \textcolor{red}{0}.

We have shown in Paper III, using the view abstraction method from Section 4, that this protocol is not safe in the case of non-atomic global transitions.
6.6 Bakery mutual exclusion

This case study describes a simplified version of the original Bakery algorithm [43]. In this version [51], processes have states that range over \{1, 2, 3\}, where 1 is the initial state. A process gets a ticket with a value strictly higher than the ticket value of any process in the queue (transition 1 → 2). A process accesses the critical section if it has a ticket with the lowest value among the existing tickets (transition 2 → 3). Finally, a process leaves the critical section, freeing its ticket (transition 3 → 1). Mutual exclusion violation corresponds to configurations where more than one process is in state 3.

6.7 MOSI Cache Coherence Protocol

The MOSI protocol is an extension of the basic MSI cache coherency protocol. It is a snoop-based protocol. It adds the state Owned O, which indicates that the current processor owns this block, and will service requests from other processors for the block. This also reduces the amount of write-back data upon cache eviction.

The protocol can be modeled as a parameterized system where the processes are arranged in a multiset. Each process can communicate with its neighbors by broadcasting a message on the connecting bus.

The state of a cache line can be Modified M, Owned O, Shared S, and Invalid I. The broadcast communication is depicted using the two following automata. Initially, all cache lines are in state Invalid I. The permitted states of any given pair of cache lines is given in the table beside.

The first automaton represents the action taken when the given process issues the broadcast message and/or when it manipulates the cache line. The cache can be written (CPUwrite), caused by a store miss, it can be read (CPUread) caused by a load miss and finally, it can be replaced (CPUrepl). The message sent on the bus vary depending on the state of the current cache line. The active process can send a read-to-share (BUSrsts) request, a read-to-write (BUSrtw) request, a write-back (BUSwb) request (in case of cache eviction) or an invalidate (BUSinv) request. We label each edge of the automaton with the action issued by the current process and with the message it sent on the bus. We use − when no message is sent on the bus.
6.8 German Cache Coherence Protocol

The directory-based cache protocol consists of a central server $H$ (for Home), and a set of client caches $C_1, \ldots, C_n$. Each client $C_i$ communicates with the server through a set of three message channels namely $ch^1_i, ch^2_i, ch^3_i$ in a star topology.

A client $C_i$ can request to share a given cache and sends a $ReqShared$ or $ReqExclusive$ through its $ch^1_i$. The Home node will reply in $ch^2_i$ with an $Invalidate$, a $GrantShared$, or a $GrantExclusive$ message. The client acknowledges the reception by sending a $InvAck$ message in $ch^3_i$.

A cache line can be in state $Invalid$, $Shared$ or $Exclusive$. If a cache line is in state $Invalid$ in the cache $C_i$, the client $i$ does not have access to that cache line. If a client has been granted the access, by $Home$, possibly along with
other clients, the cache line is in state *Shared*. *Home* can also grant the access exclusively to a client, if which case the cache line is in state *Exclusive*.

Initially, all channels are empty and all cache lines are in state *Invalid*. A bad configuration is detected when two (or more) processes have exclusive access to a given cache line, or if one accesses the cache line exclusively whilst the others still have a shared access.

We model each client in a parametrized system with multiset topology. The actions of *H* are represented in each process and its (bounded) local variables are modeled as shared variables. The model follows [53]. We assume the channels to be of length one and therefore can represent them by a local variable (for each client). In addition to channels, the central controller manipulates five data:

- **excl**: a flag to remember whether the exclusive access has been granted.
- **ctl**: a pointer to the client that sent the request being served.
- **sh_list**: a list of the processes having an access, either shared or exclusive, to the cache line.
- **inv_list**: a list of processes which have to be invalidated in order to serve the current request.
- **cmd**: a message read in some buffer.

A client *i* may perform one of the following actions:

- **c1**: If in state invalid and *ch*$_1^i$ is empty, the client *i* sends a request to *H* for a shared access via its *ch*$_1$.
- **c2**: If in state invalid or shared while *ch*$_1^i$ is empty, the client *i* sends a request to *H* for an exclusive access via its *ch*$_1$.
- **c3**: If the client *i* is granted a shared access via *ch*$_2^i$, it consumes the message from the channel and update the state of the cache line to *Shared*.
- **c4**: If the client *i* is granted an exclusive access via *ch*$_2^i$, it consumes the message from the channel and update the state of the cache line to *Exclusive*.
- **c5**: If the client *i* receives an invalidation message through *ch*$_2^i$ and *ch*$_3^i$ is empty, it changes the state of the cache line to *Invalid*, empties *ch*$_2^i$ and sends an invalidation acknowledgment to *H* via *ch*$_3^i$.

Depending on the content of the channels and the values of the shared variables, *H* may perform one of the following actions.

- **h1**: If *H* is idle (i.e. **cmd** is empty) and receives a request via some *ch*$_1$, it consumes the received request from *ch*$_1$ into **cmd**, selects the sender to be the current client and copies the content of the sharer list to the invalidation list.
In case some \( ch_2 \) is empty, the current command is a shared request and the exclusive access has not been granted, then \( H \) grants a shared access to the current client via \( ch_2 \) and adds the client to the shared list and returns idle.

If the current command is either a shared request (while the exclusive flag is set) or an exclusive request, some \( ch_2 \) is empty, then \( H \) sends an invalidation message every process through \( ch_2 \) and removes these processes from the invalidation list.

In case the current command is a request for either a shared or an exclusive access and \( H \) receives an invalidation acknowledgment from a client via \( ch_3 \), then \( H \) removes a client from the sharer list, resets the exclusive flag and empties \( ch_3 \).

Using this model, the safety properties we checked are: (i) no two clients are simultaneously granted an exclusive access, and (ii) no client in state shared coexists with a client in state exclusive. We described the model, the variables and transitions in plain text. The complete description and pseudocode can be found in [53].

Modeled as a Petri Net.

A simplified version of the protocol has been modeled as a Petri Net as in [33]. It uses 12 places: \( idle, serveS, serveE, grantS, \) and \( grantE \) model which request is being served, \( ex, notEx \) to model the exclusive flag, \( waitS, waitE \) to model the response from \( H \), and finally \( shared, excl, invalid \) to model the cache line states.

Initially one token is in \( ex \) (such that we model the exclusive flag is \( \text{true} \)) and each token that represents a process is in \( invalid \). A bad configuration is detected when the place \( excl \) contains 2 or more tokens.
6.9 Tree Protocols

In this section, we present the tree protocols that have been used in benchmarks in the papers of this thesis. For a tree transition, if the tree pattern \( p_1, l_1, r_1 \) is found in the tree, the rule is applied and the nodes change their local state to \( p_2, l_2, r_2 \) respectively. We represent the rule in a compact form on the right.

6.9.1 Token Protocol

The protocol operates on binary trees to transmit a token from the root to the leaves. A node can be labeled as having the token \( t \), or not having the token \( n \). Initially, the token is in the root. As an example:

The set of bad constraints \( B \) is represented by trees where at least two nodes contain the token.

6.9.2 Two-way Token Protocol

This protocol is a generalization of the token protocol by allowing the token to both move upwards and downwards. Initially, the token is in the root. The set of bad constraints \( B \) is represented by trees where at least two nodes contain the token, similarly to the simple token protocol. The rules for the propagation of the token are:
As an example:

6.9.3 The Tree Percolate Protocol

The protocol [41] operates on binary trees of processes and evaluates the disjunction of the values in the leaves up to the root. The states are \{0, 1, u\}. Initially, the leaves contain either 0 or 1 and all other nodes are labeled as undefined u. A still undefined inner node will be labeled as 1 if at least one of its children contains 1, and 0 otherwise. As an example:

The rules are depicted as follows.

The set of bad constraints B is represented by trees where 0 gets propagated upwards while there is 1 below.

6.9.4 The Leader Election Protocol

The protocol operates on binary trees to elect a leader among processes which reside in the leaves and which are candidates. A leaf process can be labeled as candidate c or non-candidate n. An inner node is initially labeled as undefined u and will be labeled as candidate if at least one of its children is candidate, and non-candidate otherwise.
In a first phase, the information that a node is candidate or not travels up to reach the root. In a second phase, the decision $el$, initially in the root, travels down from candidate parent to candidate child. (If several children are candidates, the parent chooses one undeterministically). Once the decision reaches a leaf, this leaf process is elected as leader.

The set of initial constraints $I$ is represented by trees where leaves are either candidates or non-candidates, inner nodes are labeled undefined and the root is labeled $el$. The set of bad constraints $B$ is represented by trees where at least two nodes in different branches are elected (on the right). The rules for the upwards propagation of candidate information are:

$$\begin{array}{c}
\text{u/c} \\
c \\
n
\end{array}$$

$$\begin{array}{c}
\text{u/c} \\
c \\
n
\end{array}$$

$$\begin{array}{c}
\text{u/c} \\
c \\
n
\end{array}$$

$$\begin{array}{c}
\text{u/n} \\
n \\
n
\end{array}$$

The rules for the downwards propagation of election decision are:

$$\begin{array}{c}
\text{el} \\
c/el
\end{array}$$

$$\begin{array}{c}
\text{el} \\
c/el
\end{array}$$

Below follows a small scenario:

6.9.5 The Tree Arbiter Protocol

The protocol supervises the access to a shared resource of a set of processes arranged in a tree topology. The processes competing for the resource reside in the leaves. A process in the protocol can be in state $idle$, $requesting$, $token$ or $below$. All the processes are initially in state $idle$. A node is in state $below$ whenever it has a descendant in state $token$. When a leaf is in state $requesting$, the request is propagated upwards until it encounters a node which is aware of the presence of the token (i.e. a node in state $token$ or $below$). A node that has the token (in state $token$) can choose to pass it upwards or pass it downwards to a requesting child (node in state $requesting$).
We model the tree-arbiter protocol with a parameterized tree system \( P = (Q, \Delta) \) where \( Q = \{ s_n \mid s \in \{ i, r, t, b \} \land n \in \{ \text{leaf}, \text{inner}, \text{root} \} \} \) and \( \Delta \) is as depicted below, in Figure 6.9.5. Observe that in the definition of \( Q \), we use the subscript \( n \) to model the nature (leaf, inner or root) of the nodes. In the definition of the rules, we will drop the script whenever we mean that it is arbitrary (it can take any value).

The rules to model this protocol are as follows: 2 rules to propagate the request upwards, 2 rules to propagate the token downwards, 2 rules to propagate the token upwards and one rule to initiate a request from a leaf.

![Figure 6.1. The rewrite rules for the tree-arbiter protocol. Notice that there are more rules in the model: For example, the first rule on the left is represented in the concrete model by 2 rules, each of which corresponds to a particular combination of the natures of the parent and child nodes: For the parent there are 2 possibilities (\( i/\text{inner} \) and \( i/\text{root} \)) while for the child, there are 2 (\( r/\text{inner} \) and \( r/\text{leaf} \)).](image)

The set of initial configurations \( I \) contains all trees where the leaf nodes are either idle or requesting, inner nodes are idle, and the root has the token. The set of bad constraints \( B \) is represented by trees where at least two processes obtained the token (i.e. two leaves in state \( t/\text{leaf} \), see the figure on the right).

6.9.6 The IEEE 1394 Tree Identification Protocol

The 1394 High Performance serial bus \([56]\) is used to transport digitized video and audio signals within a network of multimedia systems and devices.

The tree identification protocol is used in one of the phases implementing the IEEE 1394 protocol. More precisely, it is run after a bus reset in the network and leads to the election of a unique leader node.

In this section, we consider a version working on tree topologies. Furthermore, we assume that (i) each inner node is connected to 3 neighbors, (ii) the root is connected to 2 neighbors, and (ii) communication is atomic.

Initially, all nodes are in state \( u \). We identify two steps in the protocol depending on the number \( n \) of neighbors which are still in state \( u \). If \( n > 1 \), the node waits for (“be my parent”) requests from its neighbors. If \( n = 1 \), the node sends a request to the remaining neighbor in state \( u \). We can observe that the leaf nodes are the first to communicate with their neighbors.

Formally, we derive a parameterized tree system model \( P \) as follows. We define the set of states by \( Q = \{ s_n \mid s \in \{ u, c, l \} \land n \in \{ \text{leaf}, \text{inner}, \text{root} \} \} \) where the script \( n \) describe the nature of the node. In the definition of the state
(s), the letters $u$, $c$ and $ℓ$ stand respectively for undefined, child and leader. In a similar manner to the previous protocol, we drop the script whenever we mean that it can take any value (see caption of Figure 6.9.5).

The rewrite rules $Δ$ are described as follows.

- The leaves initiate the communications:

- The inner nodes become children or wait for requests:

- The leader is chosen:

The set of initial configurations $I$ is represented by trees where all nodes are in state undefined, and the set of bad constraints $B$ is represented by trees where at least 2 leaders are elected.

6.10 Agreement protocol on a Ring

Interacting peers are organized in a circular pipeline and are given a number. The protocol in place is to ensure that every participant in the ring knows which number in the maximum among the values of the ring members. We model this protocol as an instance of the framework on rings. Each participant in the ring can communicate with its adjacent neighbors. We assume the ring oriented and a member can send messages to its (immediate) “successor” and receive messages from its “predecessor”. The reception is usually blocking while the sending is not. However, we will model this communication as a rendez-vous.
Initially, all processes are in a *dormant* state. One of the processes will wake up first. This is the one which initializes the protocol (denoted in the pseudocode as $P_0$). Every process sends the max between its own value and the value it received from its predecessor (the biggest value it has seen so far). Note that the protocol doesn’t terminate when $P_0$ receives the biggest value in the ring. It must indeed communicate this value to the others. Each will receive it from its predecessor and only pass it along to its successor (after recording it). The protocol terminates when the predecessor of $P_0$ receives the biggest value and avoids the re-sending. A bad configuration is detected if one of the participants is in its final state (6 or 17) but has not seen the biggest value go by. We depict the protocol using the following pseudocode. The channel between $P_{i-1}$ and $P_i$ is called `values[n]`. A message in the channel will contain the number the sender sent.

```plaintext
channel values[n](int largest);

/* For Process $P_0$: initiates the exchanges */
1 int val; // Assume val has been initialized
2 int largest = val; // Initial state
3 send values[1](largest);
4 receive values[0](largest);
5 send values[1](largest); // Finally
6 // end

/* For Process $P_1,\ldots,n-1$ */
int val; // Assume val has been initialized
int largest; // Initial state
receive values[i](largest);
// then update its val by comparison
if (val > largest) { largest = val; }
// Send the result to the next process
send values[(i+1) % n](largest);
// and then wait to get the global result
receive values[i](largest);
if (i < n-1) send values[i+1](largest); // Finally
// end
```
6.11 Critical section guarded by a lock

We can model the critical section problem by read and write access to a resource shared by multiple processes. There is no particular topology so we will model it as an instance of a parametrized system with multisets. More precisely, as a Petri Net. As the petri net of a concurrent program, described at that level of granularity, can quickly grow in size, we choose a short example: The processes repeatedly grab the lock, increment a counter and release the lock. A bad configuration is detected when two or more processes are accessing the shared variable simultaneously while one is writing.

The shared variable counter is associated with two places, Read\textsubscript{counter} and Write\textsubscript{counter}. The tokens of a place in the petri net represent the count of process in a given state, or the available resources. A process places a token in Read\textsubscript{counter} (resp. Write\textsubscript{counter}) if it is currently accessing the variable counter for reading (resp. writing). We model read and write accesses to shared variables with two transitions, denoted by the dotted rectangle in the following figure. There is a place L associated with a lock. Intuitively, if L contains a token, the lock is free, otherwise it is busy. This ensures that only one process can hold the lock at a time. Note that L is a global variable and that we omit the input places used to balance out the number of tokens in the net.

Initially, the lock is free and the processes are in the initial state init. The petri contains then one token in L and the others in init. A bad situation is detected when the petri net contains two or more tokens in Write\textsubscript{counter} or when there is one (or more) token in Read\textsubscript{counter} and one (or more) token in Write\textsubscript{counter}.

6.12 Light Control

This algorithm implements a simple solution to the light system of an office room. An arbitrary number of people may get in or out of the room. Initially, the light is off and everyone is outside the room. The first person to enter the room turns the light on and the last person to exit the room turns it off.

A bad configuration is detected when the light is on, but there is noone in the room, or when the light is off while there are still people in the room.
We model this algorithm with a Petri Net with Inhibitor arcs. There are 2 places associated with the on and off status of the light. And there are 2 places associated with the fact that a person is inside or outside the room. An extra place exiting is used to check the person who wishes to exit is the last one in the room. Initially, there is a token in the off place, and all the other tokens in the outside place. For readability, we represent the transitions piecewise as follows.

6.13 Priority Allocator

A ferry transports one car at a time from one side of the river to the other side. At the departure, there are 2 queues of cars: One with high priority, one without priority. If there are cars in the high priority lane, the ferry must load them first. If not, it can load a car from the lane with no priority, not even with a first-come-first-served basis.

Initially, the cars must choose their respective lane, but are not allowed to board the ferry, which is not loaded. A bad configuration is detected when the ferry has loaded a car from the no-priority lane while there are still cars in the high-priority lane.

We model this situation with a Petri Net with Inhibitor arcs. There are 2 places associated with the high and low priority lanes. And there are 2 places associated with the fact that a car has been granted access on the ferry. There is an initialization phase that models how the car get attributed a priority.
6.14 Simple Barrier

A barrier is a tool to ensure synchronization between threads and allows a programmer to impose restriction for an asynchronous multi-threaded program. A process arriving at the barrier must stop at this point and cannot proceed until all other processes reach this barrier.

The following model represents a barrier with a *pivot*. The first thread at the barrier will take the role of a pivot and all other processes wait as long as there is a pivot. When all processes have arrived at the barrier, the pivot can then proceed, which in turn releases all the waiting processes.

A process can be in state *before* \( B \), *wait* \( W \), *pivot* \( P \) or *after* \( A \). Initially, all processes are in state \( B \) and a bad configuration is detected when there is a process in state \( A \), while there is still a process in state \( B \). The transition diagram and inhibitor net are as follows.

The following invariant can be derived: (i) a pivot \( P \) can coexist with a process in state \( B \) and/or \( W \), or is alone, i.e. the set of configurations \( \{ P \land B^* \land W^* \} \), (ii) a process after the barrier can only coexist with other process after the barrier or still waiting, i.e. the set of configurations \( \{ W^* \land A^+ \} \), and (iii) initially, all processes were before the barrier, i.e. \( \{ B^* \} \).

It is also possible to model it with a linear topology (as in III). In that case, we can consider a non-atomic version, which would not check the state of the other processes all at once.

A typically implementation uses an atomic counter as follows.

```plaintext
shared counter  // Initially 0, Ranges over \{0,...,n\}
shared go       // Atomic bit
local  local.go // A bit

local.go = go; // remembers the current value
< counter = counter + 1; > // atomic increment
if ( counter == n ) { // last at the barrier
    counter = 0; // reset
    go = 1 - go; // notify all
} else {
    while(local.go == go) {} // not the last
}
```

108
6.15 List with Counter Automata

Consider the following single-threaded program. It scans a list starting from the pointer $L$ and must end when it finds a marker $P$, placed prior to the start of the program. In this case, the parameter for this parametrized system if the length of the list. The program risks a segmentation fault is the marker $P$ is not found. Our analysis retain the presence of the marker information in a context (and would forget it without context).

\begin{verbatim}
  i := L
  while i ≠ P do 
      i := i.next 
  end
\end{verbatim}

6.16 Michael & Scott’s Lock-Free Queue

We show a version of the concurrent queue by Michael and Scott [49]. The program represents a queue as a linked list from the node pointed to by $Head$ to a node that is either pointed by $Tail$ or by $Tail$’s successor. The global variable $Head$ always points to a dummy cell whose successor, if any, stores the head of the queue. In the absence of garbage collection, the program must handle the ABA problem [38, 64] where a thread mistakenly assumes that a globally accessible pointer has not been changed since it previously accessed that pointer. Each pointer is therefore equipped with an additional $age$ field, which is incremented whenever the pointer is assigned a new value.

The queue can be accessed by an arbitrary number of threads, either by an enqueue method $enq(d)$, which inserts a cell containing the data value $d$ at the tail, or by a dequeue method $deq(d)$ which returns $empty$ if the queue is empty, and otherwise advances $Head$, deallocates the previous dummy cell and returns the data value stored in the new dummy cell. The algorithm uses the atomic compare-and-swap (CAS) operation. For example, the command $CAS(&Head, head, \langle next.ptr, head.age+1 \rangle)$ at line 29 of the $deq$ method checks whether the extended pointer $Head$ equals the extended pointer head (meaning that both fields $ptr$ and $age$ must agree). If not, it returns $FALSE$. Otherwise it returns $TRUE$ after assigning $\langle next.ptr, head.age+1 \rangle$ to $Head$.

The linearization points are at line 9, 22 and 29. For instance, line 9 of the $enq$ method called with data value $d$ is instrumented to generate the abstract event $enq(d)$ when the CAS command succeeds; no abstract event is generated when the CAS fails. Generation of abstract events can be conditional. For instance, line 22 of the $deq$ method is instrumented to generate $deq(empty)$ when the value assigned to $next$ satisfies $next.ptr = NULL$ (i.e. it will cause the method to return $empty$ at line 25).
6.17 Treiber’s Lock-Free Stack

You can find the example for Treiber with Garbage Collection in Section 5.1. We present here the code in the case there is no garbage collection. As in the previous section, we equip each pointer with an `age` field in order to avoid the ABA problem.

```c
void initialize() {
    node* n := new node();
    n→next.ptr := NULL;
    Head.ptr := n;
    Tail.ptr := n;
}

void enq(data d) {
    node* n := new node();
    n→val := d;
    n→next.ptr := NULL;
    while(TRUE){
        pointer_t tail := Tail;
        pointer_t next := tail.ptr→next;
        if(tail = Tail)
            if(next.ptr = NULL)
                if(CAS(&tail.ptr→next, next, ⟨n, next.age+1⟩))
                    break;
                else
                    CAS(&Tail, tail, ⟨next.ptr, tail.age+1⟩);
            }
        CAS(&Tail, tail, ⟨n, tail.age+1⟩);
    }

    data deq()
    { 
        pointer_t head;
        while(TRUE){
            pointer_t head := Head;
            pointer_t tail := Tail;
            pointer_t next := head.ptr→next;
            if(head = Head)
                if(head.ptr = tail.ptr)
                    if(next.ptr = NULL) return empty;
                    CAS(&Tail, tail, ⟨next.ptr, tail.age+1⟩);
                else
                    data result := next.ptr→val;
                    if(CAS(&Head, head, ⟨next.ptr.head.age+1⟩))
                        break;
                    free(head.ptr);
                    return result;
            }
        }

    push(data d){
        pointer_t n;
        n.ptr := new node();
        n.ptr→val := d;
        n.age := 0;
        n→next.ptr := NULL;
        while(TRUE){
            pointer t := Top;
            n→next := t;
            if(CAS(&Top, t, ⟨n, t.age+1⟩))
                break;
        }
    }

    pop(){
        pointer_t t;
        while(TRUE){
            pointer_t t := Top;
            if(t.ptr == NULL) return empty;
            pointer t := t→next;
            data result := t.ptr→val;
            if(CAS(&Top, t, ⟨n, t.age+1⟩))
                break;
            free(t.ptr);
            return result;
        }
    }
}```
My Contributions

I. I designed the method with Lukáš Holík. I am the sole implementer of the prototype and responsible for the experimentation. I participated in all parts of the writing, equally as my co-authors.

II. I designed the method with Lukáš Holík and Ahmed Rezine. I am the sole implementer of the prototype and responsible for the experimentation. I participated in all parts of the writing, equally as my co-authors.

III. I designed the method with Lukáš Holík. I am the sole implementer of the prototype and responsible for the experimentation. I participated in all parts of the writing, equally as my co-authors.

IV. I participated in the discussions and writing of the paper. I implemented the prototype and conducted the experimentation with Jonathan Cederberg.

V. I participated in the discussions and writing of the paper. I implemented the prototype and conducted the experimentation with Noomene Ben Henda.

VI. I figured how to apply the method to the setting of race detection. I am the sole implementer of the prototype and responsible for the experimentation. I am the main author of the paper.

VII. I designed the method with Lukáš Holík. I am the main author of the paper.
Conclusion and Future Work

We have presented, in this thesis, a special class of infinite-state systems, namely parameterized systems, and two methods to automatically verify their safety property. Parameterized systems arise naturally in the modeling of mutual exclusion algorithms, distributed protocols, or cache coherence protocols. They typically consist of several identical components, organized according to particular topologies, such as words, trees, rings, or multisets. The manner in which the components communicate with each other determine the different features of such systems.

Furthermore, this thesis presents the complex verification problem of concurrent programs manipulating dynamically allocated heaps, such as stacks and queues. Such programs induce an infinite-state space in several dimensions: they (i) consist of an unbounded number of concurrent threads, (ii) use unbounded dynamically allocated memory, and (iii) the domain of data values is unbounded. On top of that, they do not rely on automatic garbage collection, but manage memory explicitly. This increases even further the complexity of the problem.

The number of components is only a parameter of the verification problem and, in this thesis, we focus on proving the safety property for the system, as a whole, regardless of the value of this parameter. In order to prove safety properties, we first extract a model from the program, while omitting details that appear irrelevant for the given property. This defines the states and transitions of the program. Every behaviour of the program is represented as a succession of transitions, starting from some initial states. We transform the verification problem into the following reachability problem:

*Can the initial state reach the bad states by following the transitions of the system?*

Algorithms consist then in loops exploring the state-space. Since the set of states can be infinite, we group them together into symbolic representations, which allow algorithms to make bigger strides in their exploration. This, in turn, defines an abstract model of the program, and we employ approximation techniques to further abstract and reduce the problem into a finite-state model. In fact, we use an over-approximation, such that the abstract model encompasses all the behaviours of the original system (but it can also include others). If the bad states are not reached from the initial states in the abstract system, it is considered safe, and so is the original system. If they do, we might have found a real counter-example — in other words, a bug — or it is possible that
the approximation was too coarse and introduced a spurious behaviour to the abstract system. In that case, we need to refine the abstraction and start the method anew.

The first method performs a backward reachability analysis. The idea is to start from the bad configurations and run a model of the program backwards in order to determine which configurations could potentially reach the bad ones, i.e. configurations that are indirectly bad. Therefore, if the initial configurations do not belong to the latter set, the safety property is proven.

The second method starts from the initial configurations and focuses on small values of the parameter, incrementally. The method compute a threshold for which it is not necessary to continue: the method has indeed all the necessary information to conclude that there will be no bad configurations in instances of the system for larger values of the parameter than this threshold. To simplify, the method breaks down configurations in pieces of a given size and recombines them consistently in all possible way, especially into configurations of larger sizes. The idea is to collect all such pieces and ensure that none of the recombinations corresponds to a bad configuration. If it is the case, the threshold is found. If not, we start over with pieces of slightly larger sizes.

Both methods capture the reachability of bad configurations to imply safety. They are sound since they derive an over-approximation of the system. Besides soundness, we have discussed the completeness of both methods. The first method has been applied to a wide class of well-quasi ordered systems. The second method is precise enough to verify systems with fine-grained transitions (i.e. the non-atomic case). We have successfully extended the framework to the case of multi-threaded programs operating on dynamic heap structures.
Future work

Showing that the verification can be carried out through the analysis of only a small number of processes allowed for more efficient algorithms for these systems. We are currently working on extending the view abstraction method to infinite-state systems. This requires to modify the abstraction, the entailment, and how views are interpreted. Such models could then potentially handle data.

The shape analysis can be generalized to heap structures with multiple next-selectors. It is possible to extend the method in several directions:

- by taking into account data values attached to cells in the heap,
- by introducing new observers (to handle concurrent sets for example),
- by refining the reachability logic used to represent the shapes.

Regarding the last point, the method is designed such that it is possible to swap the current shape analysis for another one. It is already the case in [13], where an extension of the Forester tool is used to deal with data structures such as singly- and doubly-linked lists, binary search trees, as well as skip lists. Extending the reachability logic requires to refine the information regarding the relationships between variables, and potentially introduce markers such as data values, thread ownership of the pointed cell, or memory allocation status. We are currently working on representing such relationships as automata which would allow us to verify complex data-structures such as skip-list (with two or three levels).\(^1\)

Moreover, by updating the shape analysis at the heart of the method, it is possible to tackle programs where the commit point is either not within the code of the invoking method (i.e. located in another method), or changing place, depending on the instructions flow. This makes the analysis very complex, as it is possible for a particular method to synchronize with an observer, not at the moment it is performed, but later when another method commits. That is, for example, the case when the insert method is buffered. It really commits when the buffer is cleared. That is also the case when the method to check membership in a concurrent set takes into account when the insert and delete methods commit.

Another possible line of work is to extend the view abstraction to multi-threaded programs running on machines with different memory models. Such hardware systems employ store buffers and cache systems that could be modeled using views. This is an interesting challenge since it would help programmers to write their code under a given memory model that is simpler to reason around, and verify that the behaviour of the program is the same under another less-restricted memory model.

\(^1\)On a side note, it is probably possible to use the small model theorem on the levels of the skip-list, meaning that we would break the levels into small pieces and use view abstraction to reconstruct the levels in the shape analysis. That is an interesting way to reduce the state-space using the power of both methods.
References


[13] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, CongQuy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with


[38] IBM. System/370 principles of operation, 1983.
[50] Kedar S. Namjoshi. Symmetry and completeness in the analysis of


Index

ABA problem, 79
Abstract Model, 30
Abstract Post-image operator, 62
Abstraction, 28, 45, 47, 59, 61, 68, 71
Acknowledgements
    Thank You, Parosh, vii
Approximation, 29
    Over-approximation, 29
    Under-approximation, 29
Arbiter, 102
Automation, 26, 41
Bad configurations, 35, 39, 46
Bad states, 30
Broadcast Transitions, 36
Bug, 23
Cache Coherence Protocols, 35, 51
CEGAR, 29
Challenge, 41, 77, 79
Chapter summaries, 33, 43, 57, 75, 91
Commit Point, 81
Communication primitives, 36
Compare-and-Swap (CAS), 78, 79
Completeness, 66
Concretization, 61, 68, 69
Concurrency, 77
Configuration, 37, 67
Context, 80
Context-sensitive view, 71
Correctness, 26
Coverage, 24, 27
Critical section, 40
Critical systems, 23
Cut-off, 60, 63, 64
Data Independence, 83
Decidability, 47
Deductive methods, 26
Distributed Protocols, 35, 51
Downward-closed sets, 66
Duplication, 83
Entailment, 46, 67, 71, 72
Error-free Guarantee, 25
Existential transition, 37
Experiments, 51
Feasible executions, 24
Finite-state, 35
Finite-state systems, 27, 37
Firewire, 103
Fixpoint, 63, 90
Formal verification, 25
Fresh cell, 79, 85
Galois Connection, 61
Garbage-Collection, 85
Generators, 46
Global Conditions, 36
    non-atomic, 36, 38, 71
Global Transitions, 36
    destination, 37
    quantifier, 37
    range, 37
    source, 37
Heap Analysis, 31, 77, 85
Infinite-state, 35, 77
Infinite-state systems, 28, 35, 46
Initial configurations, 35, 39
Initial states, 30
Interference, 85, 89
Invariant, 63, 66, 70, 85, 86
Leader, 101
Leader Election, 101
Linearizability, 80
Liveness, 25
Local Transitions, 36
Lossiness, 83

121
Minimal elements, 46
Model, 26, 30
   Abstract Model, 28
Model Checking, 26
Monotonicity, 46
Mutual Exclusion, 35
Non Garbage-Collection, 77, 85
Non-Monotonicity, 46
Observers, 83
Operational Semantics, 27
Orderredness, 83
Ordering, 45, 49
Over-approximation, 30, 47, 60
Parameterized Systems, 30, 35, 59
   Features, 35
   Formal definition, 37
Parameterized Verification, 30, 41
Partial Order, 27
Percolate, 101
Post-image operator, 37, 62, 68, 88
Pre-image operator, 48, 50
Precision of View Abstraction, 62, 86
Predecessors, 48
Process correlation, 62
Process creation and deletion, 36
Program Model, 77
Projection, 67, 72
Push-button technology, 26
Reachability, 30, 35, 39, 47, 59, 85
   Backward Analysis, 48
   Forward Analysis, 59
Rendez-vous Transitions, 36
Retry-Loop, 78, 86
Safety, 23, 25, 30, 35, 39, 63
Shape Analysis, 85
Shared variable update, 36
Size-preserving transitions, 59
Small Model Property, 59, 71
Small Preconditions, 65
Soundness, 29, 30, 66, 70
Specification, 25, 80
Stack, 78, 84
   Treiber’s, 78
State-space
   acceleration, 27
   exploration, 27
   explosion, 25, 36
States, 26
Subword, 45
Successors, 37
Summary
   Résumé , xv
   Sammanfattning , xi
Symbolic Representation, 86
Symbolic Representations, 27, 49, 68, 87
Szymanski, 40
   Mutual exclusion protocol, 40
Theorem Prover, 26
Theoretical limitations, 32
   Decidability, 32
   Intractability, 32
Thesis outline, xxi
Thread Corrolation, 86
Thread-Modular, 85
Token Protocol, 100
Topology, 35
   linear, 35, 37
   multiset, 35
   ring, 35
   tree, 35
Traces, 81, 82
Transitions, 26
Transitive Closure, 85
Tree-Arbiter Protocol, 102
Treiber’s stack, 78
Two-way Token, 100
Unbounded, 77, 84
Universal transition, 37
Upward-closure, 45
Verification Algorithm, 50, 64
Acceleration, 65, 71
Requirements, 65
Termination, 50, 66, 70

Verification Methods
  CEGAR, 29
  Deductive methods, 26
  Model Checking, 26
  Simulation, 24
  Slicing, 24
  Static Analysis, 24
  Testing, 24
  Theorem Prover, 26

Verification Scheme, 49, 62
View, 61
  Context-sensitive, 67
  Extensions, 64, 69

Well-Quasi Ordering (WQO), 49, 66
Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1302

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-264171