The Cubicle Fuzzy Loop : A Testing Framework for Cubicle

Alexandrina Korneva

▶ To cite this version:

Alexandrina Korneva. The Cubicle Fuzzy Loop : A Testing Framework for Cubicle. Logic in Computer Science [cs.LO]. Université Paris-Saclay, 2023. English. NNT : 2023UPASG095 . tel-04505645

HAL Id: tel-04505645
https://theses.hal.science/tel-04505645

Submitted on 15 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

université
PARIS-SACLAY

The Cubicle Fuzzy Loop: A Testing Framework for Cubicle

La boucle de fuzzing CFL: un cadre de test pour Cubicle

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique
Graduate School : Informatique et sciences du numérique. Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **Laboratoire Méthodes Formelles (Université Paris-Saclay, CNRS, ENS Paris-Saclay)**, sous la direction de **Sylvain CONCHON**, Professeur des universités, le co-encadrement de **Fatiha ZAIDI**, Professeure des universités

Thèse soutenue à Paris-Saclay, le 8 décembre 2023, par

Alexandrina KORNEVA

Composition du jury

Membres du jury avec voix délibérative

Membre	Rôle
Dominique QUADRI Professeure des universités, Université Paris-Saclay	Présidente
Régine LALEAU Professeure des universités, Université Paris-Est Créteil	Rapporteure & Examinatrice
Stephan MERZ Directeur de recherche, INRIA Nancy	Rapporteur & Examinateur
Sylvain HALLÉ Professeur des universités, Université du Québec à Chicoutimi	Examinateur
Pascal POIZAT Professeur des universités, Université Paris Nanterre	Examinateur

THESE DE DOCTORAT

NNT : 2023UPASG095

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

université PARIS-SACLAY

Titre: La boucle de fuzzing CFL: un cadre de test pour Cubicle

Mots clés: Systèmes paramétrés, Test à données aléatoires, Vérification de modèles

Résumé: L'objectif de cette thèse est d'intégrer une technique de test dans le *model checker* Cubicle. Pour cela, nous avons étendu Cubicle avec une boucle de Fuzzing (appelée *Cubicle Fuzzy Loop* – CFL). Cette nouvelle fonctionnalité remplit deux fonctions principales.

Tout d'abord, elle sert d'oracle pour l'algorithme de génération d'invariants de Cubicle. Ce dernier, basé sur une exploration en avant de l'ensemble des états atteignables, était fortement limité par ses heuristiques lorsqu'elles sont appliquées à des modèles fortement concurrents. CFL apporte une nouvelle manière plus efficace d'explorer ces modèles, en particulier il permet de visiter beaucoup plus d'états pertinents. Son deuxième objectif est de détecter rapidement et efficacement les problèmes et les vulnérabilités dans les modèles de toutes tailles, ainsi que de capturer les *deadlocks*.

L'intégration de CFL nous a également permis d'augmenter l'expressivité du langage d'entrée de Cubicle, avec l'inclusion de nouvelles primitives pour manipuler des threads (verrous, sémaphores, etc.).

Enfin, nous avons construit un cadre de test autour de Cubicle et de CFL avec un interpréteur interactif, utile pour le débogage, le prototypage et l'exécution pas à pas des modèles. Ce nouveau système a été appliqué avec succès sur une étude de cas d'un algorithme de consensus distribué pour blockchains.

Title: The Cubicle Fuzzy Loop: A Testing Framework for Cubicle

Keywords: Parameterized systems, Fuzzing, Model Checking

Abstract: The goal of this thesis is to integrate a testing technique into the Cubicle model checker. To do this, we extended Cubicle with a Fuzzing loop (called the Cubicle Fuzzy Loop - CFL). This new feature serves two primary purposes.

Firstly, it acts as an oracle for Cubicle's invariant generation algorithm. The existing algorithm, which is based on a forward exploration of reachable states, was significantly limited by its heuristics when applied to highly concurrent models. CFL introduces a more efficient way to explore these models, visiting a larger number of relevant states.

Its second objective is to quickly and efficiently detect issues and vulnerabilities in models of all sizes, as well as detect deadlocks.

The integration of CFL has also enabled us to enhance the expressiveness of Cubicle's input language, including new primitives for manipulating threads (locks, semaphores, etc.).

Lastly, we built a testing framework around Cubicle and CFL with an interactive interpreter, which is useful for debugging, prototyping, and step-by-step execution of models. This new system has been successfully applied in a case study of a distributed consensus algorithm for blockchains

Contents

1 Introduction **11**

2 Background **17**
2.1 Model Checking Modulo Theories . . . 17
2.2 Cubicle: An Efficient Implementation of MCMT . . . 24
2.2.1 Cubicle's Input Language . . . 24
2.2.2 Cubicle's Optimizations and Benchmarks . . . 29
2.3 Fuzzing . . . 34
2.3.1 A Motivating Example . . . 34
2.3.2 Fuzzing: A Quick Background . . . 35
2.3.3 Fuzzing and Cubicle . . . 36

3 Invariant Generation in Cubicle **39**
3.1 BRAB . . . 39
3.1.1 BRAB Algorithm . . . 40
3.1.2 BRAB Benchmarks . . . 44
3.2 Potential Limitations . . . 44
3.2.1 A Motivating Example . . . 45
3.2.2 Bad Approximations . . . 48
3.2.3 Buggy and unsafe models . . . 50

4 The Cubicle Fuzzy Loop (CFL) **53**
4.1 Fuzzing Cubicle . . . 53
4.2 General CFL Structure . . . 55
4.3 Fuzzing the Scheduler . . . 61
4.4 BRAB & CFL: Experimental Results . . . 64
4.5 Testing Models . . . 67
4.6 Parameterized Fuzzing . . . 69
4.7 Discussion: Heuristics, Decisions, Fuzzing . . . 72
4.8 Discussion: Stability . . . 73

5 CFL-Based Threads **77**
5.1 Primitives 79
5.2 Cubicle vs. Primitives 80
5.3 Language Extensions & Semantics 82
5.3.1 Distinguishing Threads and Processes 82
5.3.2 Initial States and Unsafe States 83
5.3.3 Locks and Reentrant Locks 84
5.3.4 Conditions 87
5.3.5 Semaphores 88
5.3.6 Restrictions 89
5.4 Examples 89
5.4.1 Producer-Consumer 89
5.4.2 Dining Philosophers 91
5.4.3 Other Examples 97

6 Interactivity, Execution, and Debugging **99**
6.1 Interpreter Commands 99
6.2 Usage Example 102

7 Test Case: Tenderbake **107**
7.1 Blockchains & Consensus 107
7.2 Tenderbake 113
7.3 Modeling & Testing a Simple Tenderbake 114

8 Related Work **125**

9 Conclusion & Perspectives **127**

List of Figures

2.1 Modified Mutual Exclusion Algorithm 18
2.3 Cubicle code type syntax 25
2.2 Cubicle code of a mutual exclusion algorithm 26
2.4 Cubicle code global declaration syntax 27
2.5 Cubicle code transition syntax 27
2.6 Cubicle code transition syntax 28
2.7 Cubicle Case Syntax 28
2.8 Simple Generated Cubicle Proof 30
2.9 Detailed Generated Cubicle Proof 31
2.10 Cubicle Benchmarks 33
2.11 Simple C Example 35

3.1 BRAB on a Simple Example 42
3.2 BRAB Benchmarks 45
3.3 Concurrent Systems Pattern 46
3.4 Pattern as a Concrete Example 47
3.5 BRAB with DFS and BFS on Modified Examples 48

4.1 Comparing CFL with Different Forward Strategies. 65
4.2 Number of Generated Proof Nodes for Each Strategy 66
4.3 Comparison with CMurphi 66
4.4 Unsafe: Backward and CMurphi 67
4.5 Unsafe: Backward vs. CFL 68
4.6 Deadlock Detection 69
4.7 Same Behaviors with Multiple Processes 70
4.8 Cubicle Code to Test Fuzzer Parameterization 71

5.1 Encoding a Semaphore in Cubicle 81
5.2 Cubicle Transition Actor Syntax 82
5.3 Cubicle Proc Subtype Syntax 83
5.4 Cubicle Lock Declarations 84
5.5 Cubicle Acquiring and Releasing Locks 84
5.6 Cubicle Acquiring and Releasing Locks 85

5.7 Cubicle Equivalent of Python Wait 88
5.8 Cubicle Semaphores 89
5.9 Producer-Consumer in Cubicle 92
5.10 Dining Philosophers in Python (no semaphore) 94
5.11 Dining Philosophers in Cubicle (no semaphores) 95
5.12 Dining Philosophers in Cubicle (with semaphores) 96

6.1 Interpreter Welcome Screen 100
6.2 Interpreter Producer-Consumer Initial Status Screen 102
6.3 Interpreter Helper Functions 103
6.4 Interpreter Results for Buggy Producer-Consumer 103
6.5 Interpreter Trace 104
6.6 Alternative Trace 104
6.7 Interpreter Replay 105
6.8 Reaching an Unsafe State 105
6.9 Alternative Trace 106

7.1 General Structure 108
7.2 P2P Network 109
7.3 A normal execution of PBFT with one faulty node 113
7.4 Baker automaton 115
7.5 Basic Version of Tenderbake 117
7.6 New Version of Tenderbake 119
7.7 Declarations for Third Version of Tenderbake 121
7.8 Transition for Third Version of Tenderbake 122

Remerciements / Acknowledgments

First of all, I would like to thank Régine Laleau and Stephan Merz for agreeing to review my thesis and for their feedback on my work. Thank you to Dominique Quadri for presiding over my thesis jury and thank you to Sylvain Hallé and Pascal Poizat for agreeing to be a part of my jury. I am glad that I was able to present my work in front of you.

I would also very much like to thank my thesis advisors - Sylvain Conchon and Fatiha Zaïdi. You guys were always great. Thank you, Fatiha, for always being easy to talk to about anything. And Sylvain - I don't think that I could write anything here that would truly express how grateful I am. You were a great advisor, and you helped me through a lot. You still do. I'm really glad that I get to keep working with you. Thank you :)

My PhD years wouldn't have been as great as they were without the ex-VALS team, especially Jean-Christophe, Kim, Fred, Safouan, Sylvie, Chantal, Thibaut, Guillaume. Thank you for being so welcoming and so patient with the endless decorating every holiday. A separate thank you to Andrei - I'm gonna miss the ducks and the random post-its.

And speaking of post-its, I want to say a huge thank you to my friends and fellow PhD students (not students anymore). Covering things in post-its, gluing things around the lab, wrapping furniture in gift wrap, moving furniture between floors, going to Costco... the list goes on and on.

And thank you to my family and friends. I'm going to have to translate this for some of you, but I just wanna say that your love and support were, and are, essential.

Overall, my PhD was a fantastic experience that changed my life for the better, and I will truly miss it.

Résumé

Les systèmes informatiques sont devenus non seulement plus présents dans la vie quotidienne, mais aussi plus grands et plus complexes. Les ordinateurs sont également de plus en plus présents dans les systèmes critiques, tels que les avions, les centrales nucléaires, la médecine, etc.

Malheureusement, les systèmes informatiques contiennent des bugs qui peuvent conduire à des conséquences désastreuses, en particulier lorsqu'ils affectent des systèmes critiques. Et quand on regarde la taille et la complexité des systèmes d'aujourd'hui, trouver n'importe quel type de bug ou garantir l'absence de bug semble être impossible.

Heureusement, la plupart des grands systèmes sont construits sur une fondation constituée de petits éléments, et garantir l'absence de bugs dans ces éléments est déjà crucial, car une fondation défectueuse conduit à un système défectueux. Parmi les exemples d'éléments de fondation, on trouve les protocoles de cohérence de cache, les barrières de synchronisation et les algorithmes de consensus utilisés dans les blockchains. Tous les exemples que nous avons donnés d'éléments de fondation font référence à des systèmes concurrents.

Un aspect important des systèmes concurrents ou distribués est qu'ils sont généralement conçus pour un nombre arbitraire de composants. Par exemple, les algorithmes de consensus dans les blockchains sont censés fonctionner avec n'importe quel nombre de participants. De même, les protocoles de cohérence de cache devraient être opérationnels quel que soit le nombre de processeurs, tout comme les barrières de synchronisation doivent gérer n'importe quel nombre de threads. Ces types de systèmes, qui contiennent un nombre arbitraire et potentiellement inconnu de composants répliqués, sont appelés des systèmes paramétrés. Dans le contexte de cette thèse, nous nous concentrons sur la vérification de systèmes paramétrés, qu'ils soient concurrents ou distribués.

Lorsqu'on parle de vérifier des systèmes, il y a deux types de propriétés dont nous pouvons parler : *safety* et *liveness*. Dans le contexte de cette thèse, nous nous concentrons sur les propriétés de *safety*. Plus précisément, des propriétés de *safety* aussi simples que "un état donné est-il atteignable ou non ?".

Les systèmes qui ont un espace d'état fini peuvent être vérifiés automatiquement. Cependant, ce n'est pas le cas pour les systèmes paramétrés, pour lesquels le problème général est indécidable. Il peut cependant être décidable pour certains types restreints de systèmes, tels que les *array-based systems*, pour lesquels la *safety* est décidable sous certaines conditions. Les états dans les *array-based systems* sont représentés comme des tableaux indexés par un nombre arbitraire de composants. Une technique pour résoudre le problème

d'atteignabilité de ces systèmes est MCMT (Model Checking Modulo Theories), inventée par Ghilardi et Ranise. Dans MCMT, les états et les transitions sont représentés comme des formules dans un fragment particulier de la logique du premier ordre, et la technique pour résoudre le problème d'atteignabilité prend la forme d'une analyse de l'atteignabilité arrière, qui fait appel à un solveur SMT pour raisonner sur les formules.

Une implémentation efficace de MCMT qui utilise des invariants est Cubicle, un outil de vérification open-source pour les systèmes paramétrés, conçu par Conchon *et al.* à l'Université Paris-Saclay en partenariat avec Intel. Tout comme dans MCMT, Cubicle exécute une boucle d'atteignabilité arrière et exploite un solveur SMT. Cubicle renforce la boucle en exécutant une exploration en avant finie et limitée du système. Les états visités lors de cette exploration sont utilisés pour inférer des invariants. Cette technique est extrêmement efficace et a permis à Cubicle de prouver complètement automatiquement un protocole de cohérence de cache de taille industrielle appelé FLASH.

Bien qu'efficace, il est important de noter que cette méthode repose fortement sur la visite d'un nombre suffisant d'états représentatifs du système. Dans sa version actuelle, Cubicle utilise des stratégies d'exploration BFS et DFS optimisées. Malheureusement, pour certains types de systèmes, ces stratégies d'exploration en avant ne parviennent pas à visiter suffisamment d'états représentatifs, notamment ceux qui contiennent du non-déterminisme, du pipeline, des barrières de synchronisation complexes. Il y a deux conséquences à cela, toutes deux partageant la même cause racine de ne pas avoir assez d'informations. La première conséquence est que Cubicle est incapable de générer des invariants, ce qui signifie qu'il n'y a rien à utiliser pour élaguer l'espace d'état et accélérer la preuve, risquant ainsi de ne pas terminer. La deuxième conséquence est que Cubicle génère des invariants incorrects. S'il n'a pas visité suffisamment d'états, il commence à supposer que certains états ne sont pas atteignables et infère donc de mauvais invariants. Heureusement, lorsque cela se produit, Cubicle dispose de mécanismes lui permettant de détecter qu'il a fait une erreur et de revenir en arrière. Il est important de souligner que la complétude de Cubicle n'est jamais menacée par l'inférence incorrecte d'invariants, cependant, le retour en arrière le ralentit considérablement.

Une manière de résoudre ce problème est d'introduire un algorithme en avant capable de (i) explorer rapidement les états, et (ii) explorer suffisamment d'états critiques. Nous avons également besoin que les états visités soient diversifiés car, si l'algorithme en avant explore rapidement de nombreux états mais qu'ils sont principalement de même type, cela n'apporte aucune nouvelle information et donc aucun nouvel invariant.

Pour aborder ce problème, dans cette thèse, nous nous tournons vers

le monde du test pour inspiration. Le test ne cherche pas à prouver qu'une chose est jamais atteignable. Au contraire, le test essaie de montrer qu'une chose est atteignable.

Parmi les nombreuses techniques de test existantes, une technique souvent utilisée est le fuzzing. Le fuzzing est très utilisé dans le monde industriel parce qu'il n'est pas intrusif et nécessite seulement de pouvoir exécuter un système, de modifier (ou générer) ses entrées, et d'observer le comportement résultant. Le fuzzing essaie de provoquer autant de comportements différents que possible, ce qui nous convient bien, puisque notre objectif est de visiter autant d'états différents que possible. Nous implémentons une extension de Cubicle basée sur le fuzzing appelée Cubicle Fuzzy Loop (CFL).

La conséquence quelque peu inattendue de se tourner vers le fuzzing est que nous obtenons maintenant le meilleur des deux mondes. Notre algorithme en avant basé sur le fuzzing s'est avéré si efficace pour trouver des états subtils que nous avons décidé de le transformer en un outil autonome pour Cubicle, appelé CFL (Cubicle Fuzzy Loop). Évidemment, CFL n'est pas paramétré, mais il est suffisamment puissant pour couvrir suffisamment les espaces d'état et découvrir des bugs. Il est également beaucoup plus facile à étendre que le système de preuve de Cubicle, puisqu'il n'est pas restreint par le fait de devoir s'adapter aux *array-based systems*.

Chapter 1

Introduction

Throughout the years, computer systems have become more prevalent in everyday life. But not only have they become more common, they have also become larger and more complex. Microsoft, for example, went from 4000 lines of code with MS-DOS [1] in the 1980's to 45 million lines of code for Windows XP [2] in the 2000's. Smartphones today are more powerful than the guidance systems used by NASA in the 1960's [3]. By certain estimations, back in 2015, all of Google's internet services combined made up two billion lines of code [4]. Computers are also increasingly present in *critical systems* [5]. These are systems where any failure results in significant damage, be it economical, environmental, or the loss of human life. Critical systems include aircraft, nuclear power plants, the medical field, etc.

The widespread use and dependence on computer systems comes with the potential presence of *bugs*. Some bugs lead to disastrous consequences, especially when they hit critical systems. For example, this was the case with Therac-25 [6], a radiation therapy machine. Race conditions in the code led to the machine giving patients doses of radiation hundreds of times bigger than it was supposed to. Everyone today is aware that computer systems contain bugs, and they can be dangerous. But when you look at the size of modern computer systems, tackling the problem of bugs seems incredibly daunting- how can you find one tiny subtle bug in millions of lines of code? How can you guarantee the safety and security of systems that keep growing and getting more and more complex?

Fortunately, most giant systems are built on a foundation of smaller parts, and guaranteeing the absence of bugs in these parts is already important, because a faulty foundation leads to a faulty system. Some examples include:

- cache coherence protocols in multiprocessor systems, which allow processors to keep their local caches coherent with the memory
- synchronization barriers, which allow threads to reach a common point before continuing

- consensus algorithms in blockchains, which allow all members to decide on which blocks are added to the blockchain.

So far, the term *system* itself has been doing a lot of heavy lifting, because it encompasses a wide range of definitions. It can be used to described sequential computer programs, concurrent systems- where multiple components execute tasks simultaneously, and distributed systems- where multiple devices not only work together, but can also fail (partial, permanent, or Byzantine failures). The notion of *system* itself can also be used to represent smaller parts of larger systems such as algorithms and protocols, i.e the foundations we were talking about.

In this thesis, we use the word "system" interchangeably to talk about algorithms and protocols that constitute what we consider to be the foundations of huge computer systems. Furthermore, examples we gave for foundation elements (consensus, cache coherence, synchronization barriers) all talk about concurrent systems.

An important aspect of concurrent or distributed systems is that they are usually conceived for an arbitrary number of components. For example, blockchain consensus algorithms are supposed to work for any number of participants. Cache coherence protocols should work for any number of processors, as should synchronization barriers for threads. Such types of systems containing an arbitrary, potentially unknown, number of *replicated* components lead to what is known as *parameterized* systems. *In the context of this thesis, we focus on verifying concurrent and distributed parameterized systems.*

When talking about verifying systems, there are two types of properties we can talk about: safety and liveness. Safety properties state that something bad will *never* happen, while liveness properties state that something good will *eventually* happen. The terms "safety" and "liveness" and their definitions originate from Leslie Lamport [7]. *In the context of this thesis, we focus on safety properties. More precisely safety properties as simple as "is a given state reachable or not?".*

While the algorithms and protocols we are interested in are all the foundation elements of larger systems, they can range from easier-to-handle academic examples to huge, complex industrial versions. The main complexity is due to the inherent nondeterminism, which comes from processor speeds, message passing times, etc. The consequence of this complexity is that the state space of these systems is enormous, so to make sure that a certain state is never reachable, you need to visit a large number of the states, which is impossible to do manually. Adding on to that complexity are failures, such as message loss, which can happen at any time. Plus the arbitrary number of components. The only solution to proving on such large systems is to automate the verification process and make use of *computer-aided* techniques.

Computer-aided verification has been around for decades [8, 9]. Systems that have a finite state space can be automatically checked [10]. However, this is not the case for parameterized systems. As soon as a tool is expressive enough to simulate a Turing machine (like with something as simple as a Minsky machine with two counters) the *general* state reachability problem becomes undecidable [11]. While the general problem is undecidable, it can be decidable for certain restricted types of systems. *In this thesis we focus on a specific class of systems called array-based transition systems* [12] *for which safety is decidable under certain conditions*. *States in array-based systems are represented as arrays indexed by an arbitrary number of components.* A technique to solve the reachability problem of array-based systems is MCMT (Model Checking Modulo Theories) [13], invented by Ghilardi and Ranise. In MCMT states and transitions are both represented as formulas in a particular fragment of first-order logic and the technique to solve the reachability problem takes the form of a backward reachability analysis, which makes use of SMT solver to reason about the formulas.

Even though MCMT falls in a decidable fragment under certain conditions, there are no guarantees that is is capable of verifying a system efficiently. The main difficulty remains the state explosion problem. In MCMT, this problem results in logic formulas that can keep growing larger, resulting in SMT solvers struggling to treat them. To overcome this issue, one of the most powerful techniques is to prune the search space by finding logic formulas that characterize the largest set of state that can or cannot be reached. These formulas are called *invariants*.

An efficient implementation of MCMT that makes use of invariants is Cubicle [14, 15], an open-source verification tool for parameterized systems designed by Conchon *et al.* at Université Paris-Saclay in partnership with Intel. Just as in MCMT, Cubicle runs a backward reachability loop and exploits an SMT solver. The SMT solver in Cubicle is called Alt-Ergo Zero, a lighter, more optimized version of Alt-Ergo [16]. Cubicle reinforces the loop by running a finite, limited forward exploration of the system. The states visited during this exploration are used to infer invariants, and even though the exploration is limited, it provides Cubicle with enough information. This technique is extremely efficient and allowed Cubicle to *completely automatically* prove an industrial-sized cache coherence protocol called FLASH [17].

While being effective, it is important to note that this method strongly relies on visiting enough representative system states. In its current version, Cubicle makes use of optimized BFS and DFS exploration strategies. Unfortunately, for certain types of systems, these forward exploration strategies struggle to visit enough representative states. Both BFS and DFS are quickly overwhelmed as soon as systems begin exhibiting high levels of features like branching, nondeterminism, pipelining, complex conditions. There are two

consequences to this, both sharing the same root cause of not having enough information. The first consequence is Cubicle is unable to generate any invariants, meaning there is nothing to use to prune the state space and accelerate the proof. This means that Cubicle risks not terminating. The second consequence is that Cubicle generates *incorrect* invariants. Recall that Cubicle uses a set of *visited* states to infer invariants. If it has not visited enough states, it starts assuming states are not reachable and thus infers wrong invariants. Fortunately, when this happens, Cubicle has built-in mechanisms allowing it to detect that it has made a mistake. These mechanisms force it to *backtrack*. It is important to highlight that Cubicle's completeness is **never** threatened by incorrectly inferring invariants, however backtracking slows it down considerably.

A way of resolving this problem is introducing a forward algorithm that is capable of (i) rapidly exploring states, and (ii) exploring enough critical states. We also need visited states to be diverse because, if the forward algorithm explores a lot of states rapidly, but they are mostly the same types of states, this brings in no new information and therefore no new invariants.

To tackle this problem, in this thesis we turn to the world of testing for inspiration on how to modify the forward exploration. Testing does not look to prove that something is *never* reachable. On the contrary, testing tries to show that something *is* reachable. It looks for paths to get somewhere, because when testing a system, you need to find those intricate places where bugs could be hiding. The same applies to our forward exploration - we do not want our forward exploration to exhaustively prove that a state is unreachable. We do not even want it to exhaustively visit all possible states. We only want it to find a diverse enough set of reachable states so that Cubicle in turn can infer invariants.

Among the numerous existing testing techniques, an often-turned to technique is *fuzzing* [18, 19, 20]. Fuzzing is popular in the industrial world because it is not intrusive and only requires being able to execute a system, modify (or generate) its inputs, and observe the resulting behavior. Fuzzing tries to provoke as many different behaviors as it can, which works out well for us, since our goal is to visit as many different states as possible. We implement an extension of Cubicle based on fuzzing called the Cubicle Fuzzy Loop (CFL).

The somewhat unintended consequence of turning to fuzzing is that we now get the best of both worlds. Our fuzzing-based forward algorithm turned out to be so efficient at finding subtle states, that we decided to turn it into a standalone tool for Cubicle, called CFL (Cubicle Fuzzy Loop). Obviously CFL is not parameterized, but it is powerful enough to sufficiently cover state spaces and discover bugs. It is also much easier to extend than Cubicle's proof system, since it is not restricted by having to fit into array-based systems.

Contributions and Outline

Our contributions are as follows:

- A fuzzing approach for Cubicle called CFL (Cubicle Fuzzy Loop), for which we present and discuss different heuristics. This approach is implemented in a new extension of Cubicle.
- CFL-based extensions to Cubicle's language, which have allowed us to model and run algorithms that we were unable to before, as well as test models in a standalone fashion. CFL also introduces deadlock detection to Cubicle.
- An integration of CFL as an Oracle into Cubicle's existing invariant generation algorithm
- A new, executable version of Cubicle, along with a built-in interactive layer for step-by-step execution and debugging.

The rest of this document is organized as follows: in **Chapter 2**, we give an overview the existing techniques that the work in this thesis relies on, namely Model Checking Modulo Theories, the model checker Cubicle, and fuzzing. In **Chapter 3** we describe how Cubicle synthesizes invariants and uses them to prove complex models. We discuss current limitations. **Chapter 4** introduces CFL. It shows how we adapted fuzzing to Cubicle, and describes the various implemented heuristics. Chapter 4 also presents various benchmarks and discussions relating to CFL and its performance. **Chapter 5** presents our CFL-based thread extension for Cubicle and applies it to various classic concurrent programs. Chapter 6 shows the interactive layer that has been added to Cubicle. **Chapter 7** applies CFL to Tenderbake [21], a blockchain consensus protocol. Finally **Chapter 9** concludes and discusses various perspectives.

Chapter 2

Background

Contents

2.1 Model Checking Modulo Theories 17

2.2 Cubicle: An Efficient Implementation of MCMT 24

2.2.1 Cubicle's Input Language 24

2.2.2 Cubicle's Optimizations and Benchmarks . . 29

2.3 Fuzzing 34

2.3.1 A Motivating Example 34

2.3.2 Fuzzing: A Quick Background 35

2.3.3 Fuzzing and Cubicle 36

Parameterized model checking [22, 23, 24, 25] addresses the problem of verifying that properties hold for a system with an arbitrary number of (replicated, indistinguishable) components. Among the techniques for treating parameterized systems, and the one Cubicle is based on, is MCMT - Model Checking Modulo Theories. To counter the undecidability result [11], MCMT restricts the types of systems you can verify to *array-based* systems.

2.1 Model Checking Modulo Theories

The Model Checking Modulo Theories (MCMT) framework [13, 26] proposed by Ghilardi and Ranise is a declarative framework for parameterized systems in which (sets of) states, transitions and properties are expressed in a particular fragment of first order logic with enumerative data types.

Systems in MCMT are called *array-based transition systems*, because their states can be seen as a set of unbounded arrays (denoted by capital letters $\mathsf{X}, \mathsf{Y}, \ldots$) whose indexes range over elements of a parameterized domain, called proc, of process identifiers (denoted by $i, j, \ldots$).

If we have an array X and a process variable i, we write X$[i]$ for an array access of X at index i. Systems can also contain simple variables. Note that, although from a theoretical point of view a variable is seen as an array with the same value in all its cells, we use the terms *array* and *variable* for easier legibility.

Throughout this section, we will be explaining every concept through the example given in Figure 2.1, which gives an array-based system implementing a simple mutual exclusion algorithm.

Mutual Exclusion

User-declared types:

$\texttt{type}\ \ state = \texttt{Idle}\ \ |\ \ \texttt{Crit}\ \ |\ \ \texttt{Want}$

Globals:

$\texttt{Turn}\ :\ \ proc$

$\texttt{State}\ :\ \ (proc, state)\ \ array$

Init:

$\forall k.\ \ \texttt{State}[k] = \texttt{Idle}$

Unsafe:

$\exists\, k \neq \ell.\ \ \texttt{State}[k] = \texttt{Crit} \wedge \texttt{State}[\ell] = \texttt{Crit}$

req:

$\exists k.\ \ \texttt{State}[k] = \texttt{Idle} \wedge \texttt{Turn} = k\ \wedge$
$\qquad \texttt{State}' = \texttt{State}[k \leftarrow \texttt{Want}] \wedge \texttt{Turn}' = \texttt{Turn}$

enter:

$\exists k.\ \ \texttt{State}[k] = \texttt{Want} \wedge \texttt{Turn} = k\ \wedge$
$\qquad \texttt{State}' = \texttt{State}[k \leftarrow \texttt{Crit}] \wedge \texttt{Turn}' = \texttt{Turn}$

exit:

$\exists\, k \neq \ell.\ \ \texttt{State}[k] = \texttt{Crit}\ \wedge$
$\qquad \texttt{State}' = \texttt{State}[k \leftarrow \texttt{Idle}] \wedge \texttt{Turn}' = \ell$

Figure 2.1: Modified Mutual Exclusion Algorithm

User-defined Types In our example, we define an enumerative type $state$ with three constructors: `Idle`, `Crit`, `Want`. These symbolize the three possible

states of a process in our mutual exclusion:

- `Idle`: the process is not doing anything in particular
- `Want`: the process has requested access to the critical section
- `Crit`: the process has been granted access to the critical section

An array-based system is defined by three things: a set of global arrays and variables, an initial state, a set of transitions expressing how you get from one state to another.

Global Declarations In our example, we declare one variable and one array.

$$\begin{aligned} \texttt{Turn} &: \ proc \\ \texttt{State} &: \ (proc, state) \ array \end{aligned}$$

The variable `Turn` has the $proc$ type and will indicate whose turn it is to access the critical section in our mutual exclusion. The array `State` maps processes to a value of type $state$, i.e. indicates what state each process is in.

Init An array-based system has to contain an initial state. The initial state, noted I, is a universally quantified formula that defines the initial values for arrays and variables in the system. If an array or variable does not appear, then that means that its value can be anything in its domain. In our example, the initial formula is given as

$$\forall k. \quad \texttt{State}[k] = \texttt{Idle}$$

In the initial state, the value of `State` for each process is `Idle`, and the value of `Turn` is not fixed, since it does not appear. Its initial value can be anything from the $proc$ type domain, i.e. any process identifier.

Unsafe Safety properties to be verified on array-based systems are expressed in their negated form as formulas that represent unsafe states, i.e states you want to avoid. In our example, the unsafe formula is:

$$\exists\, k \neq \ell. \quad \texttt{State}[k] = \texttt{Crit} \wedge \texttt{State}[\ell] = \texttt{Crit}$$

This states that the system is unsafe if it reaches a state where any distinct two processes are in the critical section simultaneously. For that, we use the notation $k \neq \ell$. It is also possible to declare multiple unsafe states.

Transitions A transition t is represented by an existentially quantified formula. The formula expresses the values of variables and arrays before and after the transition. The values before a transition are noted as V and the values after are noted V'. For example, the transition **Exit** in our example:

$$\exists k \neq \ell. \quad \texttt{State}[k] = \texttt{Crit} \ \wedge \ \texttt{State}' = \texttt{State}[k \leftarrow \texttt{Idle}] \wedge \texttt{Turn}' = \ell$$

states that there are two distinct processes, k and ℓ such that process k is in the critical section. *If* there is such a process, then it leaves the critical section, going back to being idle ($\texttt{State}[k \leftarrow \texttt{Idle}]$) and $\texttt{Turn}$ has changed to ℓ. The $\texttt{State}' = \texttt{State}[k \leftarrow \texttt{Idle}]$ notation denotes an array equal to $\texttt{State}$, except for cell k, which is now equal to $\texttt{Idle}$. The other two transitions in our example, **Req** and **Enter**, are written in the same fashion.

Safety and Backward Reachability To check that the unsafe state are unreachable, MCMT uses a backward reachability loop. Starting from the unsafe state, it iteratively computes the pre-image closure (to be understood as unreachable states). MCMT make use of an SMT back-end for termination and safety tests. A pre-image pre of a state s is a state from which s can be reached by taking one transition. The backward reachability algorithm, Bwd, is given in Algorithm 1. The function Bwd takes as input a triplet $\mathcal{S}$ consisting of the initial state I, the set of variables and arrays $\mathcal{X}$, and the set of all transitions τ. The function also takes the unsafe state(s), U.

Algorithm 1: Backward Reachability Algorithm

```
1  function Bwd(S, U) : begin
2      V := ∅;
3      push(Q, U);
4      while not_empty(Q) do
5          φ := pop(Q);
6          if φ ∧ I satisfiable then
7              return unsafe
8          else if φ ⊭ V then
9              V := V ∪ {φ};
10             push(Q, Pre_τ(φ));
11     return safe
```

Bwd$(\mathcal{S}, U)$ computes the pre-image closure of U by maintaining two collections of states:

- Q contains the (unsafe) states to visit (it is initialized with U)
- V is filled with the visited states (initially empty)

Each iteration of the loop performs the following operations:

1. (*pop*) retrieve and remove a formula φ from Q
2. (*safety test*) check the satisfiability of $\varphi \wedge I$, *i.e.* determine if the states described by φ intersect with the initial states I. If so, the system is declared *unsafe*
3. (*fixpoint test*) check if $\varphi \models V$ is valid, *i.e.* determine if the states described by φ have already been visited. If so, discard φ and go back to 1
4. (*pre-image*) compute the pre-image $\text{Pre}_\tau(\varphi)$ of φ and add the new state(s) to Q. The notation $\text{Pre}_\tau(\varphi)$ stands for the disjunction of all $\text{Pre}_t(\varphi)$, i.e. the pre-image calculations of each transition t is τ.

If Q is empty at step 1, then all of the state space has been explored and the system is declared *safe*. Note that the (non-trivial) fixpoint and safety tests are discharged to an embedded SMT solver.

We illustrate how this works on our mutual exclusion example from Figure 2.1, following Algorithm 1 to see if our system is safe.
We recall the unsafe state U, which is passed to the algorithm along with the triplet containing the initial state, the variables and arrays, and the transitions:

$$\exists\, i \neq j.\, \texttt{State}[i] \texttt{ = Crit } \wedge \texttt{ State}[j] \texttt{ = Crit}$$

After lines 2 and 3, V is empty and Q contains our unsafe state U:

$$V = \emptyset$$
$$Q = \{U\}$$

Since Q is not empty (line 4), it will be popped and φ will be equal to U.

The states described by φ do not describe the states described by the initial state, I, so the check on line 6 ($\varphi \wedge I$ satisfiable) does not pass. We can therefore move on to line 8. We add φ to V on line 9, and we calculate and push all of the pre-images of φ to Q on line 10. The pre-images are the states that can lead to φ in one transition.

Our formula φ is

$$\exists\, i \neq j.\, \texttt{State}[i] \texttt{ = Crit } \wedge \texttt{ State}[j] \texttt{ = Crit}$$

and we want to figure out which transition(s) could have set either of the values to Crit. If we look at our system in Figure 2.2, the only transition that changes a state to Crit is enter:

$$\exists k. \quad \texttt{State}[k] = \texttt{Want} \land \texttt{Turn} = k \land \\ \texttt{State}' = \texttt{State}[k \leftarrow \texttt{Crit}] \land \texttt{Turn}' = \texttt{Turn}$$

This means that there was a process k that satisfied the requirements for the transition and activated it. We use the notation $\texttt{t}(i)$ to signify that the transition t was applied to process i. We therefore want to calculate $\text{Pre}_{\texttt{enter}}(\varphi)$ which means calculating the pre-image of φ by the transition enter in the cases where k equals i, j, or a new process m. We note these pre-images as $\text{Pre}_{\texttt{enter}(i)}(\varphi)$, $\text{Pre}_{\texttt{enter}(j)}(\varphi)$, and $\text{Pre}_{\texttt{enter}(m)}(\varphi)$.

Case 1: $k = i$, where the pre-image becomes:

$$\text{Pre}_{\texttt{enter}(i)}(\varphi) = \\ \exists i \neq j.\, \texttt{State}[i] = \texttt{Want} \land \texttt{State}[j] = \texttt{Crit} \land \texttt{Turn} = i$$

Case 2: $k = j$, where the pre-image becomes:

$$\text{Pre}_{\texttt{enter}(j)}(\varphi) = \\ \exists i \neq j.\, \texttt{State}[i] = \texttt{Crit} \land \texttt{State}[j] = \texttt{Want} \land \texttt{Turn} = j$$

Case 3: $k \neq i \land k \neq j$, in which case the pre-image is calculated for some process m:

$$\text{Pre}_{\texttt{enter}(m)}(\varphi) = \\ \exists i \neq j \neq m.\, \texttt{State}[i] = \texttt{Crit} \land \texttt{State}[j] = \texttt{Crit} \land \texttt{State}[m] = \texttt{Crit} \land \texttt{Turn} = m$$

We refer to these pre-images as $p1$, $p2$, and $p3$, respectively. All of them are pushed to Q, which now contains $p1$, $p2$, $p3$ while V contains U:

$$V = U \\ Q = \{p1, p2, p3\}$$

The algorithm will pop $p1$, check that it is not subsumed by V, and then calculate its pre-image(s). The forumula $p1$ is

$$\exists i \neq j.\, \texttt{State}[i] = \texttt{Want} \land \texttt{State}[j] = \texttt{Crit} \land \texttt{Turn} = i$$

so we can look for a transition that changes a state to Want, a state to Crit, or modifies Turn.

Transition req modifies the state to Want. In this case, we only care about $\texttt{req}(i)$ and $\texttt{req}(m)$, since the process j is in Crit. If we calculate the pre-image for $\texttt{req}(m)$, we get an incoherent result, where the variable Turn has two values. So we ignore it. We keep the calculated pre-image for $\texttt{req}(i)$:

($p4$) $\text{Pre}_{\texttt{req}(i)}(p1)$ =
$\texttt{Turn = } i \wedge \texttt{State[}i\texttt{] = Idle} \wedge \texttt{State[}j\texttt{] = Crit}$

The transition `enter` modifies a state to `Crit`. We can calculate `enter` with j or m. However, in both cases, we end up with contradictory values for `Turn`, so we can throw away the results.
The variable `Turn` is modified by `exit`. In our case we can calculate $\texttt{exit(}j\texttt{,}i\texttt{)}$ and $\texttt{exit(}m\texttt{, }i\texttt{)}$. The pre-image calculation for $\texttt{exit(}j\texttt{, }i\texttt{)}$ will also be thrown away, because the formula becomes inconsistent. Our state contains $\texttt{State[}j\texttt{] = Crit}$, whereas had the transition been applied, the value would be $\texttt{State[}j\texttt{] = Idle}$. The pre-image for $\texttt{exit(}m\texttt{, }i\texttt{)}$ is, however, valid.

($p5$) $\text{Pre}_{\texttt{exit}(m,i)}(p1)$ =
$\texttt{Turn = } i \wedge \texttt{State[}i\texttt{] = Want} \wedge \texttt{State[}j\texttt{] = Crit}$
$\wedge \texttt{State[}m\texttt{] = Crit}$

These are added to Q:

$$V = U, p1$$
$$Q = \{p2, p3, p4, p5\}$$

The formula $p2$ is popped:

$$\exists i \neq j.\, \texttt{State}[i] = \texttt{Crit} \wedge \texttt{State}[j] = \texttt{Want} \wedge \texttt{Turn} = j$$

This formula is subsumed by $p1$, which is now in V, so Cubicle will not calculate its pre-image. We now have

$$V = U, p1$$
$$Q = \{p3, p4, p5\}$$

Next, we pop $p3$:

$$\exists i \neq j \neq m.\, \texttt{State}[i] = \texttt{Crit} \wedge \texttt{State}[j] = \texttt{Crit} \wedge \texttt{State}[m] = \texttt{Crit} \wedge \texttt{Turn} = m$$

This formula is subsumed by U, so no pre-images are calculated. We now have:

$$V = U, p1$$
$$Q = \{p4, p5\}$$

We pop $p4$:

$$\texttt{Turn = } i \wedge \texttt{State[}i\texttt{] = Idle} \wedge \texttt{State[}j\texttt{] = Crit}$$

This leads to one new pre-image:

($p6$) $\text{Pre}_{\texttt{exit}(m,i)}(p4)$ =
`Turn` = i ∧ `State[`i`]` = `Idle` ∧ `State[`j`]` = `Crit` ∧ `State[`m`]` = `Crit`

We now have:

$$V = U, p1, p4$$
$$Q = \{p5, p6\}$$

Next we pop $p5$:

`Turn` = i ∧ `State[`i`]` = `Want` ∧ `State[`j`]` = `Crit` ∧ `State[`m`]` = `Crit`

which is subsumed by U, so we do not calculate its pre-image. Similarly, $p6$ is also subsumed bu U. We now have:

$$V = U, p1, p4$$
$$Q = \emptyset$$

Since all values have been popped from Q, the algorithm has terminated and can return *safe*.

2.2 Cubicle: An Efficient Implementation of MCMT

The problem with the Bwd algorithm in Algorithm 1 described earlier is that if you simply implement it without any second thought, it does not handle anything but the most trivial examples well. For example, if we look at line 6 ($\varphi \wedge I$ satisfiable) this check is done by an SMT solver. However, if the solver is called on every single formula, even the most trivially incorrect one, it can quickly get overwhelmed.

Cubicle implements the MCMT framework described earlier, but it optimizes the base-version of the algorithm to make it efficient. In this section we describe these optimizations and show how they affect the system.

2.2.1 Cubicle's Input Language

We first present Cubicle's input language. The examples used further in the thesis are written in Cubicle syntax, so here we give an explanation on how to read them.

Cubicle's language is built to model (i) a set of type, global variable, and global array declarations, (ii) a formula describing the initial state, (iii) one (or multiple) formula(s) describing potential unsafe states, and (iv) a set of

guarded transitions. It was inspired by the language of the Murφ [27] model checker. Arrays and variables in Cubicle may contain **integers**,**real numbers**, **booleans**, **constructors** from an **enumerative user-defined datatype**, or **process identifiers**. Figure 2.2 shows the Cubicle code equivalent of our example from Figure 2.1 with a few slight modifications native to Cubicle.

Type and Global Declarations All type declarations in Cubicle follow the same form, as shown in Figure 2.3. A user can declare as many types as they want, provided that the type and constructor names are all unique.

```
type name = Constructor1 | Constructor2 | ...
```

Figure 2.3: Cubicle code type syntax

Variable and array declarations follow the same general structure, given in Figure 2.4.

The types, i.e. `type_of_Var/Array`, can be any of Cubicle's builtin types (i.e. proc, int, real, bool), or a user-defined type.
Technically speaking, an array declaration is of the form:

```
array Array [parameter_type] : type_of_Array
```

Since Cubicle is only parameterized by processes (proc), `parameter_type` is always `proc`. One can, however, declare 2D arrays, i.e. matrices, by writing:

```
array Array [proc,proc] : type_of_Array
```

Init, Unsafe The syntax for initial formulas and unsafe formulas is fairly straightforward when compared to our example in Figure 2.1. As we can see in Figure 2.2, the initial state is written as

```
init (i) {...}
```

and is a mandatory part of any Cubicle model. Contrary to Figure 2.1, when declaring unsafe states in Cubicle that touch multiple processes (as in our example where we define an unsafe state in regards to two processes), you do not need to specify that the processes are different. When declaring an unsafe state, every process in the parentheses is unique. So in our case in Figure 2.2, when we write

```
unsafe (i j) {...}
```

processes `i` and `j` are different.

mutual_ex.cub

```
(*user type declaration*)
type state = Idle | Want | Crit

(*variable declaration*)
var Turn : proc
(*array declaration*)
array State[proc] : state

(*initial state*)
init (i) { State[i] = Idle }

(*unsafe state*)
unsafe (k l) { State[k] = Crit &&
               State[l] = Crit }

(*transitions*)
transition req (k)
requires { State[k] = Idle && Turn = k}
{ State[i] := Want }

transition enter (k)
requires { State[k] = Want && Turn = k}
{ State[i] := Crit; }

transition exit (k l)
requires { State[k] = Crit }
{
  Turn := l ;
  State[k] := Idle;
}
```

Figure 2.2: Cubicle code of a mutual exclusion algorithm

```
var Variable  : type_of_Var

array Array[proc] : type_of_Var
```

Figure 2.4: Cubicle code global declaration syntax

Transitions Transitions in Cubicle follow the syntax given in Figure 2.5, where `guard` is the conditions that need to be met for the transition to be triggered and `actions` are the updates to the global variables and arrays. The

```
transition transition_name (parameters)
requires { guard }
{ actions }
```

Figure 2.5: Cubicle code transition syntax

parameters in a transition are all processes, since Cubicle is only parameterized by the process type. As with unsafe states, all the process listed in parameters are unique.

Guards Guards in Cubicle transitions are either comparisons ($=$, $\neq$, $<$, $\leq$) between two terms or *universal guards*, i.e. $\forall$ appearing in the guard. The theoretical framework Cubicle is based on does not include universal guards. However Cubicle is less restrictive than MCMT, which makes it more powerful in what it can do. Our mutual exclusion example does not include any universal guards, but we could easily rewrite a transition to include one. For example if we take transition `enter`, we could add the condition that all other processes be in a state different from `Want`. This would be overkill in real life, because we are protected by the `Turn` variable, but it is not *technically* incorrect. This transition is given in Figure 2.6.

```
transition enter (i)
requires { State[i] = Want && Turn = i &&
           forall_other j. State[j] <> Want }
{ State[i] := Crit; }
```

Figure 2.6: Cubicle code transition syntax

Actions Actions in Cubicle are updates to variables and arrays and are of the form:

```
Variable_name := Value;
Array_name[process identifier]:= Value
```

Cubicle also accepts pattern matching in the form given in Figure 2.7.

```
{ Array[k] := case | condition1 : value1;
                   | condition2 : value2;
                   | _ : value_by_default }

{ Variable := case | condition1 : value1;
                   | _ : value2 }
```

Figure 2.7: Cubicle Case Syntax

When applying pattern matching to arrays, the process identifier has to be different from the process identifiers in the parameters of the transition. This is because the pattern is basically a forall statement on the indexes of the array. If you want to change the value for a process `i` that was a parameter, then you would just write

```
Array[k] := case : | k = i : ...
```

In fact, the syntax we see in our example in Figure 2.9 in transition `enter` for example:

```
State[i] := Crit
```

is just syntactic sugar for:

```
State[k] := case : | k = i : Crit
                   | _ : State[k]
```

which is read as "for all process indexes `k` of array `State`, if `k` is `i`, then `State[i]` becomes equal to `Crit`, else it remains unchanged."

In the Cubicle code version of our example, transition `exit` has only one parameter and `Turn` is updated by the action

```
Turn := .
```

which differs from Figure 2.1, where the transition has two parameters and `Turn` is updated to the second one. The effect we wanted to convey was that `Turn` changes to a random process. In Cubicle's syntax, the operator `.` means "a random value of this variable's type".

Atomicity An important aspect of Cubicle's transitions is that the actions are all atomic. The semicolon in the transitions is not a sequential operator. So for example, if we have:

```
Variable_A := True;
Variable_B := Variable_A
```

The value of `Variable_B` will be the value of `Variable_A` *before* the transition, and not the new value it was given.

2.2.2 Cubicle's Optimizations and Benchmarks

Cubicle introduces numerous optimizations to make the theoretical framework viable in real life. We introduce some of them in this section. For a more thorough and detailed explanation, we refer the reader to [28].

When it comes to the SMT solver, Cubicle tries to optimize its calls as much as possible to avoid overwhelming the solver. Solver calls are incremental- when checking if a node is subsumed by another node, the context grows and is checked at each step to find incoherences as fast as possible. The formulas are instantiated in a way to minimize them and only give the solver the necessary parts that it will need. Subtyping is used to simplify how many cases the solver will need to verify when something has an enumerative data type. Cubicle also tries to first subsume formulas *syntactically* before sending them to the solver.

The way nodes are popped from Q is never specified in Algorithm 1. Cubicle implements various exploration strategies and heuristics (BFS, DFS, etc) when deciding how to explore the state space efficiently.

Visited states (V) can also be pruned of nodes that are subsumed by newly discovered nodes.

While not an optimization, Cubicle also implements an option called `-dot` that allows us to visualize the generated safety proof in various levels of detail.

For example, for our mutual exclusion example, Cubicle can generate several versions of the proof. The version in Figure 2.8 shows only the unsafe state and the two visited states(i.e. pre-images), as well as the transitions used to calculate the pre-images. A more detailed graph generated by Cubicle is given

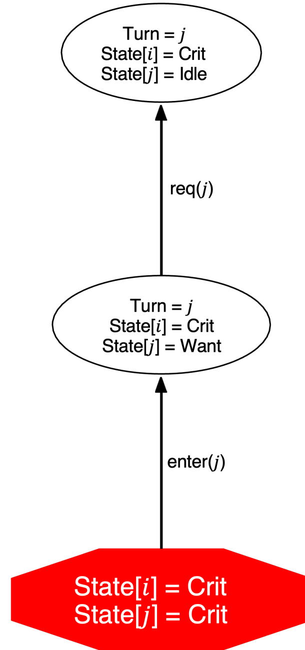

Figure 2.8: Simple Generated Cubicle Proof

in Figure 2.9. Nodes that were removed on line 8 in Algorithm 1 are in a light gray and the dotted arrows leading from them point to the nodes that they were subsumed by.

Benchmarks

We present several examples from Cubicle's library originally used to benchmark Cubicle. The full Cubicle code for each example can be found in Cubicle's Github repository [29]. Throughout this thesis, we will constantly refer to and use these benchmarks, building off of them to test CFL.
The examples can be separated into two categories: mutual exclusion and cache coherence.

Mutual Exclusion

Mutual exclusion is a problem in concurrent programming where processes have critical and noncritical sections in their code. The goal is to synchronize the processes so that no two end up in the critical section at the same time. It was first described and solved by Edsger W. Dijkstra [30, 31].

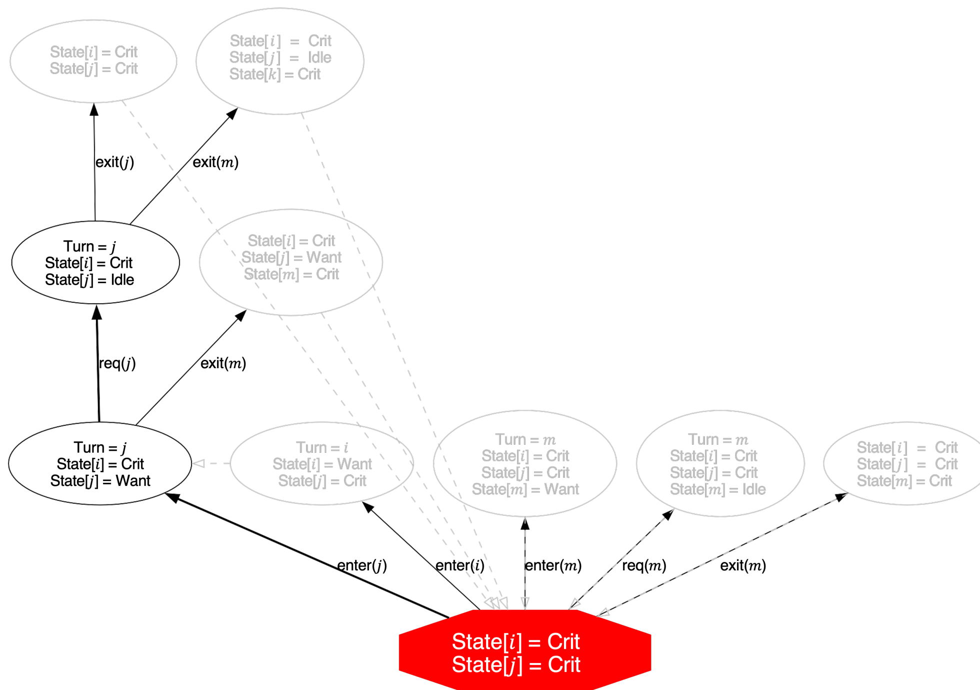

Figure 2.9: Detailed Generated Cubicle Proof

Dekker's Algorithm Historically, Dekker's algorithm is considered to be the first solution to the mutual exclusion problem for *two* processes. Dijkstra attributes the solution to the mathematician Th. J. Dekker [30], although notes that adapting Dekker's solution to more processes is initially "unclear". Dijkstra later presented and proved Dekker's solution for two processes correct and gave a generalization for an arbitrary number of processes [31, 32]. The version used for our benchmarks is a simplified version of Alain J. Martin's generalization of Dekker for n processes [33], which is based on Dijkstra's solution.

Lamport's Bakery The Bakery algorithm is a solution to the mutual exclusion problem proposed by Leslie Lamport [34]. The algorithm uses the analogy of a bakery, where customers (i.e. processes) take a number upon entering the bakery. Buying bread from the baker is equivalent to entering the critical section. By treating customers individually by number, the baker (i.e. the critical section) guarantees that only one customer is served at a time. The version we use for our benchmarks in Cubicle is a simplified version of Lamport's Bakery based on [35].

Szymanski's Algorithm Szymanski's algorithm is a mutal exclusion algorithm presented by Boleslaw K. Szymanski [36]. Szymanski's algorithm views

a system as three parts: the *prologue*- the part before the critical section, the critical section, and the *epilogue*- the part after the critical section. The analogy in this algorithm is that of a waiting room. The prologue is seen as two sections: the entrance and the waiting room. There is a door between the entrance and the waiting room, and a door between the waiting room and the critical section. Initially, the first door is opened and all processes that want to enter the critical section go through to the waiting room. When the last process enters, the door to the waiting room is closed and the door to the critical section is opened. At this point processes enter the critical section one by one in order of process identifier. The door to the waiting room is reopened only when the last process to initially enter the waiting room leaves the epilogue. We use two versions of the algorithm for our benchmarks- an atomic and nonatomic version.

Cache Coherence

In multiprocess architectures with shared memory, each component has its own cache. This is done to reduce the latency associated with constantly accessing the main memory. This use of caches leads to a problem known as *cache coherence*. Component caches need to be consistent and compatible with each other - if a component modifies something in its local cache, everyone needs to be made aware. Many techniques and protocols exist to deal with cache coherence [37, 38, 39, 40]. To run our benchmarks, we focus on two specific examples.

German German is a cache coherence protocol presented by Steve German [41]. In this protocol, there are N processes which may all request either *Shared* or *Exclusive* access to a shared cache line. We use several versions of the algorithm for our benchmarks. Some are simplified versions of the algorithm (called Germanish), while others are adaptations of other implementations of German ([42, 43]).

FLASH FLASH [44] is a multiprocess architecture designed to work for thousands of processor cores, where each core maintains its own local cache. We use a version of FLASH called Flash_nodata, which simplifies the protocol by removing all data-related aspects.

We take the examples that we described and model them in Cubicle. We give a list of how many elements each model contains:

- Dekker: three transitions, two arrays, one variable
- Germanish: a simplified German with six transitions, three variables, two arrays, and two enumerative data types

- Germanish2: a slightly more complex version than Germanish. This one has eight transitions, three variables, three arrays, two enumerative data types
- Germanish4: an even more complex version, with 10 transitions, four arrays, three variables, and two enumerative data types
- German: the actual German algorithm, with 13 transitions, six arrays, 3 variables, two enumerative data types
- German_Baukus: 13 trransitions, six arrays, 3 variables, two enumerative data tyoes
- German_CTC: 12 transitions, 10 arrays, six variables, 3 data types
- German_pfs: 15 transitions, seven arrays, three variables, four data types
- Szymanksi_at: nine transitions, four arrays, one data type
- Szymanksi_na: 16 transitions, fours arrays, one matrix, one data type
- Bakery_lamport: five transitions, three arrays, one variable, one data type
- Flash_no_data: 69 transitions, nine arrays, 22 variables, eight data types.

The results of running Cubicle are given in Figure 2.10. Timeout was set to 60

Model	**Cubicle**
Dekker	0.01s
Germanish	0.02s
Germanish2	0.04s
Germanish4	0.56s
German	3.36s
German_Baukus	3.19s
German_CTC	19.48s
German_pfs	2min
Szymanski_at	3.6min
Szymanski_na	T.O.
Bakery_lamport	0.03s
Flash_no_data	T.O.

Figure 2.10: Cubicle Benchmarks

minutes. We see that even with all of the optimizations in Cubicle, it cannot handle every example, and the more complicated the examples get, the more time it takes (as seen with the German variants). The issue is that our biggest example here, Flash_no_data, is not even close to industry-sized. We will see in Chapter 3 how Cubicle tackles the more complex systems.

2.3 Fuzzing

Fuzzing [18, 19, 20] is a technique designed to (quickly) explore a program and its execution paths. This is done by feeding said program large quantities of random inputs and seeing what effect that has on the output (i.e. detecting program crashes, memory leaks, etc).

In this section we will see what kind of problems we face with the systems that we are interested in, and how we are inspired by fuzzing.

2.3.1 A Motivating Example

In this section, we present a concurrent program in C that is very representative of the types of systems and problems we want to tackle in this thesis. In this program, several threads will run in parallel. One of the threads will observe the others, counting how many times they synchronize, i.e. reach the same program point. The goal is to run this program for a certain amount of time and see how often the threads reach a synchronization point. The code is given in Figure 2.11.

We run this program for four threads. Three of these threads will execute the function `action` (line 3) and one thread will execute the function `observe` (line 16). We declare a three-element array `PC` a program counter to show where each action-thread is in the program. The function `action(i)` on line 3 takes a thread identifier `i` (the thread identifier which executes the function) and runs a sequence of conditional instructions. We use the expression `rand() % 2 == 0` to simulate nondeterminism. In each case, a `action`-thread modifies its `PC`. At the same time, the `observe`-thread instantiates a counter (`cpt`, line 17) and observes what is going on with the `action`-threads, waiting for a specific condition to become true (line 19). This condition states that all `action`-threads have their `PC` equal to 2. When the condition is true, the `observe`-thread increments `cpt` and prints it.

We want to observe how often the `observe`-thread sees a synchronization point. We let this program run for five seconds. At the end of the five seconds, the value of `cpt` is **five**. Considering that the `observe`-thread tried the condition approximately 1379840 times in those five seconds, we can see that the condition is barely ever valid. This shows that a purely random execution, where we have no control of the scheduler, will rarely lead us to certain areas of the program. This could be very problematic if a large chunk of the code is hidden behind these synchronization points. So if we want to test this program, we risk never visiting specific areas which might contain crucial information. A technique that tries to counter this problem is *fuzzing*, which tries to visit every part of the code more evenly, including hard to reach areas.

```
1          volatile int PC[3];
2
3          void * action (int i) {
4            while (True) {
5              PC[i] = 0;
6              if (rand() % 2 == 0) {
7                PC[i] = 1;
8                if (rand() % 2 == 0)
9                PC[i] = 2;
10             }
11             else
12             PC[i] = 3;
13           };
14           return NULL;
15         }
16         void * observe(){
17           int cpt;
18           while (True) {
19             if (PC[0] == 2 && PC[1] == 2 && PC[2] == 2) {
20               cpt++;
21               printf("cpt = %d\n", cpt);
22             }
23           };
24           return NULL;
25         }
26
```

Figure 2.11: Simple C Example

2.3.2 Fuzzing: A Quick Background

When fuzzing was originally introduced [45, 46], it had no underlying focus on security, and simply meant giving random inputs to a program [47]. Today, fuzzing is primarily associated with the detection of security vulnerabilities [47], to the point where it has become very popular and is a key part of software development [47]. For example, Microsoft has included fuzzing in its *Microsoft Security Development Lifecycle* [48, 49]. Google uses fuzzing on all of its products [50, 51, 52] and is also responsible for launching OSS-Fuzz [53, 54, 55], a (free) framework for fuzzing open source projects, credited to have "helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects." [55]. Large companies are not the only ones employing fuzzing, with entities like the United States Department of Defense releasing reference guides to software development which include fuzzing [56], as well as applying fuzzing to their systems [57].

Generation vs. Mutation

We stated earlier that fuzzers originally took randomly generated input. However, the problem with purely random input is that due to its *random* nature, inputs can be syntactically invalid, and therefore rejected by the program under test.

Techniques exist in place of purely random inputs, these are mutation-based fuzzing and generation-based fuzzing [58, 20].

Mutation-based fuzzers take existing input and introduce small changes, hoping that the result will remain valid. Mutation-based fuzzers usually start from an initial set of inputs, known as seeds. Seeds can be generated by the fuzzer or, ideally, provided to the fuzzer by the user. These seeds are mutated to create new input, which is later again mutated, and so on.

Generation-based fuzzers use information about the format of the program input to generate valid inputs that follow a certain specification/structure.

Note that the literature surrounding fuzzers also talks about *grammar-based* fuzzers [59, 60] , where input generation follows a certain set grammar to produce valid data.

Black Box, Gray Box, and White Box

Fuzzers can be separated into three categories: (i) black box fuzzers, (ii) white box fuzzers, and (iii) gray box fuzzers. Black box fuzzers are the simplest of the three. They are completely unaware of the actual program code being tested, and are only able to observe the output behavior.

White box fuzzers have full knowledge of the program they are being run on, i.e. they have full access to the code. This allows them to reason about the code and explore system paths more systematically, but in return makes them slower than their black box counterpart.

Gray box fuzzers are everything that lies in between - the fuzzers are not completely blind to the code, but at the same time they do not reason about it to the extent of white box fuzzers. For example, gray box fuzzers could dynamically gather data about the program while they run, such as code coverage.

2.3.3 Fuzzing and Cubicle

The main takeaway from our quick background on fuzzing is that pure random does not work. Fuzzers that employ completely random input generation are today considered to be *naive* fuzzers. We saw the same effect with our C example from Figure 2.11- when we simply let the program scheduler do what it wanted, we did not get the results that we needed.

What the fuzzer does can be simplified to three steps:

1. Execute the program
2. Observe the program and its results
3. Modify your behavior according to what you observed.

We stated in Section 2.3.1 that the issue with our example lies with the scheduler. We have no control over which thread does what and when. Contrary to what we described about fuzzing, our example has no input to modify- the threads simply execute actions. So while we can still execute and observe what happens, we cannot *modify* any input. The only remaining solution is fuzzing the scheduler. Instead of telling the program what input to run on, we tell the program *how* to run.

As we will see in Chapter 4, this is the approach we use with CFL in Cubicle.

Chapter 3

Invariant Generation in Cubicle

Contents

3.1 BRAB **39**

3.1.1 BRAB Algorithm 40

3.1.2 BRAB Benchmarks 44

3.2 Potential Limitations **44**

3.2.1 A Motivating Example 45

3.2.2 Bad Approximations 48

3.2.3 Buggy and unsafe models 50

The backward reachability algorithm that Cubicle runs presented in Chapter 2 struggles with more complex models, such as our non-atomic Szymanski or the dataless FLASH from Figure 2.10. The reason is that those types of models have too many variables and too many possible states.

A solution to this problem is to give Cubicle more information about the system, i.e. give it system invariants. These could be given by the user, or better yet, they can be generated by Cubicle itself. This is the idea behind the BRAB (*Backward Reachability with Approximations and Backtracking*) algorithm [17]. The goal is to use a set of forward-reachable states and then exploit those states to generate invariants strong enough to help prove large models.

3.1 BRAB

As we saw earlier, the pre-image closure that is calculated is a set of unreachable states, meaning that every pre-image itself is an unreachable state. Our mutual exclusion example is fairly small, with only a couple of variables, so

the pre-images are few and simple. But if we have an algorithm with multiple variables/arrays and many more transitions, suddenly (1) the pre-images become bigger in size and (2) there are many more of them to calculate and check.

We also saw that previously-calculated pre-images are added to a set V of visited states, used for fixpoint checks. Unfortunately, when there are a lot of variables/arrays and they all start appearing in pre-images, making the states too detailed and precise and V less useful. A good idea would be to simplify the calculated pre-images, i.e. make them a little less detailed but a lot more useful for fixpoint checks. The less detail there is in a pre-image, the more states can be subsumed by it later on. The only problem is simplifying intelligently, because a badly simplified pre-image might lead to an intersection with the initial state. This means that you need some kind of Oracle to tell you what a good vs. a bad simplification is.

3.1.1 BRAB Algorithm

BRAB's solution to the Oracle problem is to use a **limited-depth forward exploration** of the system. It first computes a set $\mathcal{M}$ of reachable states using a *forward exploration* for a finite instance of the system (with a *fixed number* of processes). Then, Cubicle performs the backward reachability analysis of the *parameterized* system (as described previously by Algorithm 1). At each loop iteration, Cubicle simplifies the pre-image, i.e. it computes over-approximations and checks that they represent states that are not in $\mathcal{M}$. All of these approximations can be seen as *candidate invariants*, as they are states that can be classified as "never happening in the system". The approximations are model checked together with the original safety property. BRAB can be seen as an enhanced version of Algorithm 1, and its algorithm is given in Algorithm 2.

In the algorithm, the Bwd function takes as input a parameterized system $\mathcal{S}$ and a cube U, as before. However, now, it also takes two integers $d_{\max}$ and k. It starts by initializing a variable $\mathcal{M}$ with the set $\texttt{FWD}(d_{\max}, k)$ of reachable states constructed by a forward exploration of the reachability graph for k processes starting in a state defined by the formula $I(\#1) \land \cdots \land I(\#k)$ and limited to depth $d_{\max}$. When we run a forward exploration, the processes are instantiated, and we use the notation $\#k$, where $k \in \mathbb{N}$. FWD is not fixed and can be any user-chosen forward exploration strategy. The two choices in Cubicle are BFS and DFS.

Comparing BRAB with the original algorithm in Algorithm 1, the only striking difference is the addition of lines 11-13. This corresponds to what we mentioned earlier about approximating pre-images. The `Approx` function takes the node that was popped, and calculates its potential approximation ψ. If

Algorithm 2: Enhanced Backward Reachability Algorithm

```
1  function Bwd(S, U, d_max, k) : begin
2      M := FWD(d_max,k);
3      V := ∅;
4      push(Q, U);
5      while not_empty(Q) do
6          φ := pop(Q);
7          if φ ∧ I satisfiable then
8              return unsafe
9          else if φ ⊭ V then
10             V := V ∪ {φ};
11             ψ := Approx(φ);
12             if M ⊭ ψ then
13                 push(Q, Pre_τ(ψ))
14             else
15                 push(Q, Pre_τ(φ))
16     return safe
```

the calculated ψ represents states that are not in $\mathcal{M}$ (checks the validity of $\mathcal{M} \not\models \psi$), then Cubicle will replace φ with ψ and calculate $\text{Pre}_\tau(\psi)$ Otherwise, Cubicle will keep φ and calculate $\text{Pre}_\tau(\varphi)$.

We can look at how this works with our small mutual exclusion example. Granted, the model is small enough that it does not *require* the use of BRAB, but that does not mean that BRAB does not work on it. Figure 3.1 gives a graphical representation of how BRAB works. Some states and arrows have been omitted for the sake of readability. The candidate nodes in blue rectangles and have labels starting with the letter A (for approximation), whereas the classic pre-image nodes are in ovals and have labels starting with the letter S (for state).

We start with our unsafe state **S1**:

$$\texttt{State}[i] = \texttt{Crit} \land \texttt{State}[j] = \texttt{Crit}$$

which Cubicle will try to approximate. The way Cubicle approximates formulas is by removing literals. So if our formula has three literals for example, Cubicle will try the versions with two literals removed, and then one literal removed. So the only way to approximate the state is by `State[i] = Crit` (or j, but since the processes are existentially quantified, it means the same

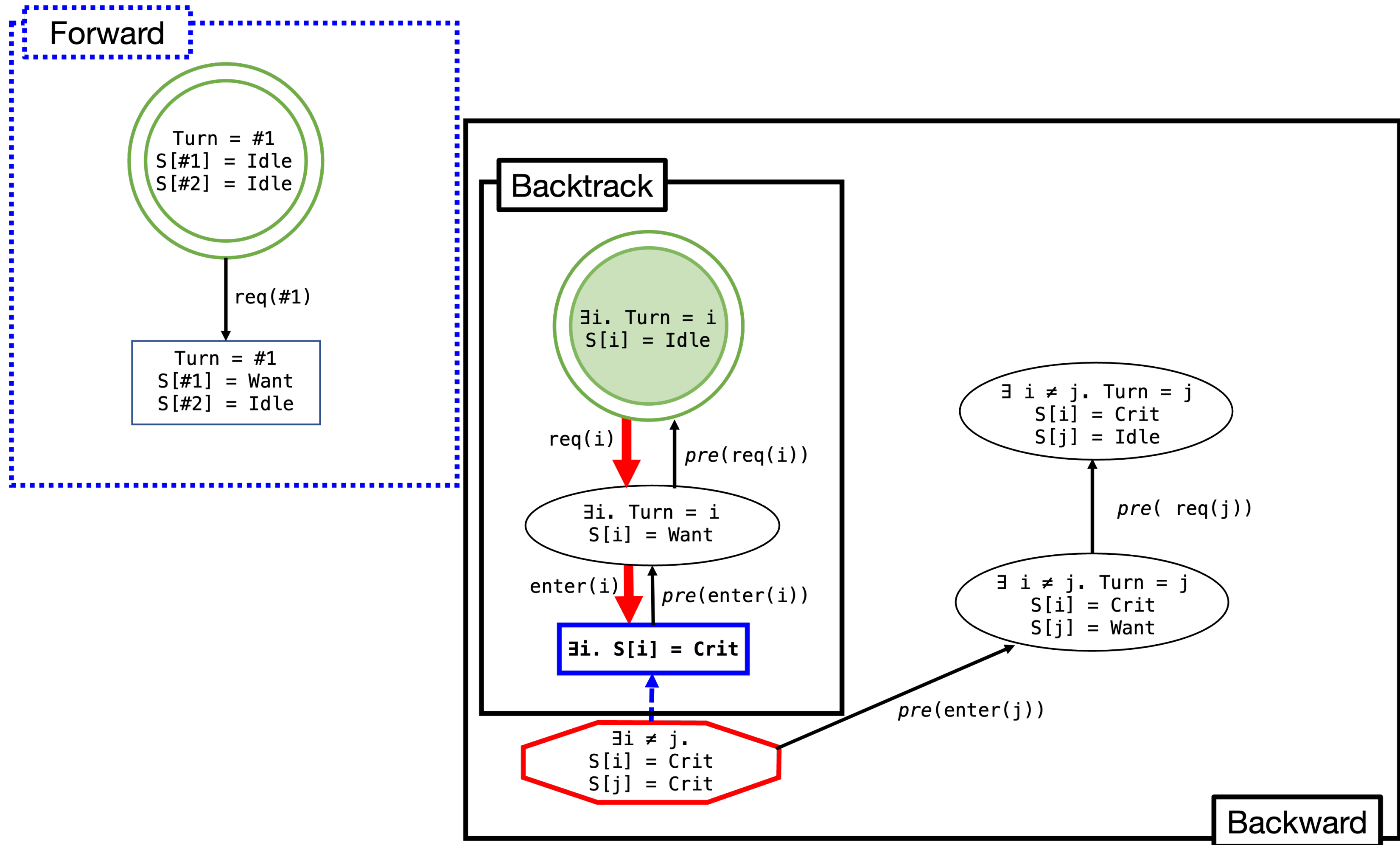

Figure 3.1: BRAB on a Simple Example

thing). However, during its forward exploration, Cubicle encountered a state that contained a `State[#1] = Crit`, meaning that saying "there is a process whose state is never in `Crit`" is a bad candidate. It is therefore rejected, and, since there are no other approximations, Cubicle computes the pre-image of **S1**. Similarly, it does the same thing with the next popped node, **S2**:

$$\texttt{Turn = } i \wedge \texttt{State[}j\texttt{] = Want} \wedge \texttt{State[}j\texttt{] = Crit}$$

In this case, there are six possible approximations:

1. $\texttt{Turn = } i$
2. $\texttt{State[}i\texttt{] = Want}$
3. $\texttt{State[}j\texttt{] = Crit}$
4. $\texttt{Turn = } i \wedge \texttt{State[}i\texttt{] = Want}$
5. $\texttt{Turn = } i \wedge \texttt{State[}j\texttt{] = Crit}$
6. $\texttt{State[}i\texttt{] = Want} \wedge \texttt{State[}j\texttt{] = Crit}$

Cubicle privileges the least amount of literals, so it will try the first three approximations first. Unfortunately, all of these can be found in the limited forward exploration. Next, Cubicle tries to privilege the approximations with the least amount of distinct processes. So it will try the fourth approximation from the list. It is also unfortunately part of the visited forward states. The last two approximations are never seen in the forward exploration, so Cubicle will choose one. In this case it chose the sixth approximation.

We note this approximation **A3** in our Figure 3.1. Cubicle will calculate its pre-image instead of **S2**'s. The pre-images calculated are:

S3 $\texttt{Turn = } i \wedge \texttt{State[}i\texttt{] = Want} \wedge \texttt{State[}j\texttt{] = Want}$

S4 $\texttt{Turn = } i \wedge \texttt{State[}i\texttt{] = Idle} \wedge \texttt{State[}j\texttt{] = Crit}$

The same approximation process will be applied to **S3** and **S4**. As mentioned, we have omit many of the pre-images and nodes in Figure 3.1. In the case of **S4**, there are no possible approximations- all possible combinations of literals can be seen in the forward exploration and all of the pre-images are subsumed by already-visited states. **S3** is approximated by **A4**:

$$\texttt{State[}i\texttt{] = Want} \wedge \texttt{State[}j\texttt{] = Want}$$

Cubicle then calculates **S4**'s pre-image and gets **S5**:

$$\texttt{State[}i\texttt{] = Want} \wedge \texttt{State[}j\texttt{] = Idle}$$

All of the pre-images for **S5** are subsumed by already visited nodes (not pictured in Figure 3.1). Cubicle therefore stops, declares the model safe, and returns the two generated invariants, which are the two valid approximations **A3** and **A4**:

A3 State[i] = Want $\wedge$ State[j] = Crit

A4 State[i] = Want $\wedge$ State[j] = Want

These invariants are to be read as:

A3 It is impossible to have two processes where one is in `Want` and the other is in `Crit`

A4 It is impossible to have two processes with their states equal to `Want`.

The danger with approximating is that sometimes these approximations can be too coarse (as we will discuss in further detail in Section 3.2), which leads to false positives. Cubicle protects itself from these cases by backtracking. If a calculated pre-image intersects with the initial state, Cubicle checks whether or not the path that led to this intersection contains an approximation. If it does, then Cubicle will backtrack, throwing away the bad approximation. This ensures completeness- a bad approximation will never affect the final safe/unsafe result.

3.1.2 BRAB Benchmarks

We go back to our examples from Chapter 2 and rerun them, this time using BRAB. We use the default forward exploration - BFS. Figure 3.2 brings back our results from Figure 2.10 and adds a column BRAB, which shows how long it took for Cubicle to prove safety while using BRAB. The Invariants Found column gives the number of invariants that BRAB was able to infer for each model. Using BRAB, Cubicle is able to prove both our non-atomic Szymanski and the dataless FLASH. We can see in Figure 3.2 that using BRAB drastically improves the time it takes for Cubicle to prove safety. For example German goes from three seconds to 30 milliseconds, in the case of our dataless FLASH, Cubicle goes from not being able to prove it at all (without timing out) to proving it in less than a second.

3.2 Potential Limitations

Before discussing what the main limitations Cubicle faces are, we first look at what kind of systems we want to tackle, and see how our problems stem from there.

Model	Cubicle	BRAB	Invariants Found
Dekker	0.01s	0.02s	0
Germanish	0.02s	0.02s	3
Germanish2	0.04s	0.02s	3
Germanish4	0.56s	0.03s	11
German	3.36s	0.10s	33
German_Baukus	3.19s	0.11s	33
German_CTC	19.48s	0.16s	33
German_pfs	2min	0.32s	43
Szymanski_at	3.6min	0.03s	14
Szymanski_na	T.O.	0.06s	15
Bakery_lamport	0.03s	0.05s	4
Flash_no_data	T.O.	0.08s	32

Figure 3.2: BRAB Benchmarks

3.2.1 A Motivating Example

If we consider real-life concurrent systems and how they are built, there are three prevailing features: (i) pipeline parallelism, (ii) synchronization barriers, and (iii) branching.

Pipeline parallelism breaks up a task into a sequence of sub-tasks, where each one can be treated concurrently by the system. This is done to improve performance by leveraging parallel processing. It complicates system models, because it not only adds depth, since each sub-task becomes an independent transition, it also introduces more interleavings to check.

Synchronization barriers are necessary to coordinate the multiple processes in a concurrent system. For example processes may be required to be in a certain configuration before gaining access to specific parts of the system. These conditions can be very precise, which can lead to them appearing rarely.

Branching is inherent to concurrent systems- processes can behave independently or run tasks in parallel, and the order in which they do this can differ from execution to execution. Not only does this lead to more branchings in

the system, it also adds *nondeterminism*.

We condense these features into a specific pattern that we want to apply to Cubicle. The pattern is shown in Fig. 3.3. In this figure we can see an

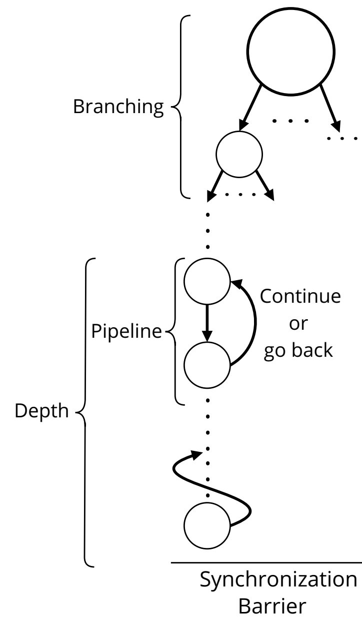

Figure 3.3: Concurrent Systems Pattern

initial node (at the top) with multiple arrows leading from it. This is to simulate **branching** and **nondeterminism**, since a process at that stage would be able to choose any of the arrows. After branching, we insert the **pipeline** - multiple transitions to represent a task. This adds depth to our models. Note that at any point, when a process gets finished with a sub-task, it can decide to either continue forward to the next task, or go back. Also note that at any point in time, any processes that have not yet moved from the initial state can execute the branching transitions. All of this culminates with a **synchronization barrier** that demands processes behave a certain way to be activated. It is important to note that while we constructed our pattern in this order, in real life the elements can appear wherever and however often they want. For example, we could have multiple synchronization barriers, or branchings can appear in multiple places throughout the system. This pattern can also repeat itself, leading to **hierarchical** systems.

The problem is that this specific pattern and its repetition, so prevalent in concurrent systems, is exactly at the root of Cubicle's limitations. Using our pattern, we build one example template for Cubicle, shown in Figure 3.4. Obviously, this is just an example, and many different templates can be built off of our described pattern and features.

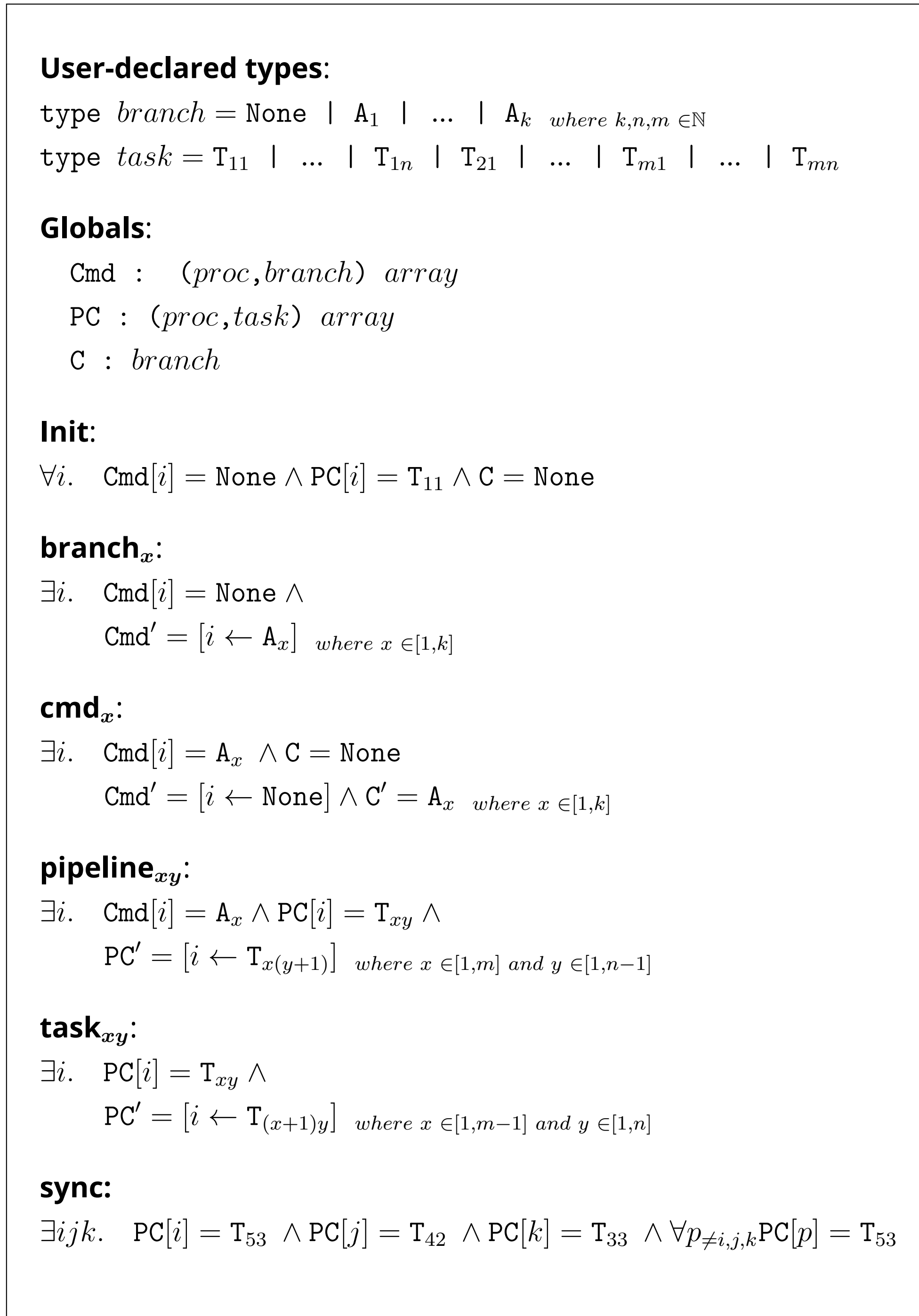

Figure 3.4: Pattern as a Concrete Example

The `branch` transitions in Figure 3.4 are to give a process initial choices. The following `cmd` transitions add even more branching by resetting the `Cmd`

Model	Forward Time	BFS			DFS		
		States	Safe	Total Time	States	Safe	Total Time
Dekker	10s	466K	T.O.	-	605K	Yes	12.72s
Germanish	10s	424K	T.O.	-	593K	Yes	12.91s
Germanish2	10s	315K	T.O.	-	515K	Yes	12.26s s
Germanish4	10s	287K	T.O.	-	547K	Yes	14.54s
German	10s	312K	T.O.	-	547K	Yes	16.25s
German_Baukus	10s	359K	T.O.	-	591L	Yes	14.82s
German_CTC	50s	1 429K	T.O.	-	2 010K	Yes	62.81s
German_pfs	10s	416K	T.O.	-	431K	Yes	17.37s
Szymanski_at	10s	372K	T.O.	-	534K	T.O.	-
Szymanski_na	10s	270K	T.O.	-	483K	T.O.	-
Bakery_lamport	40s	1 565K	T.O.	-	2038K	T.O.	-
Flash_no_data	40s	862K	T.O.	-	1 048K	T.O.	-

Figure 3.5: BRAB with DFS and BFS on Modified Examples

array. The transitions `pipeline` and `task` simulate breaking up one task into multiple sub-tasks. Note that these transitions can be repeated many times to complicate the system. We give an example synchronization barrier transition `sync`. This transition's guard can easily range from simple to more complex, and, as stated earlier, there can be multiple synchronization points, optionally with more `branch` and `task` transitions between them.

We now go back to our set of benchmarks. We modify each one to add the above pattern. Each model now contains multiple branch transitions, pipelining, and a synchronization barrier. The part of the model which represents the protocols/algorithms from our benchmarks is now hidden behind a synchronization barrier. We now rerun Cubicle with BRAB, trying both BFS and DFS. The results are given in Table 3.5. We let each forward exploration run, noting the time it ran (Forward Time) and how many states were visited (States). The columns Safe and Total Time indicate whether a model is deemed safe by Cubicle, and if yes, how much time did it take to prove safety. The Total Time includes the Forward Time. We can observe that BFS is incapable of visiting enough states to allow Cubicle to prove safety. DFS fares a bit better, but is still limited.

3.2.2 Bad Approximations

We mentioned earlier that the main issue facing BRAB is bad approximations, which force Cubicle to backtrack. Technically, there is nothing inherently *bad*

about backtracking in the sense that it does not affect the final result - safe systems will be declared safe, unsafe systems will be declared unsafe. Backtracking however slows Cubicle down, and renders BRAB inefficient. Backtracks are caused by Cubicle not visiting enough states during its forward exploration and being unable to generate crucial invariants.

Lets go back to our mutual exclusion example, and this time artificially stop the forward exploration after a depth of one level. Notice that in Figure 3.1, the very first candidate (**A1**) was rejected because of something seen in the forward exploration at a depth of two. As a consequence, now that we have forced Cubicle to stop its forward exploration earlier, this approximation, that we know is bad, will not be rejected. Cubicle will accept `State[#1] = Crit`, and calculate the pre-image from there. Calculating the pre-image via transition `enter[#3]` will lead to state `State[#1] = Want` and `Turn = #1`. In turn, when Cubicle calculates the pre-image for that, it will get `State[#1] = Idle` and `Turn = #1`, which intersects with the initial state. Cubicle will therefore be forced to backtrack and throw away our bad approximation. Once that approximation is gone, Cubicle will proceed as described before and end up with the same `Safe` result.

While Cubicle does reply `Safe` in both instances, there are two big differences between the runs. The first difference is the time- Cubicle goes from 0.01s to 0.02s. The second, more costly difference, is that no invariants are generated. In this small example, this is not very dangerous- the system is small enough to not need invariants to be proved safe. But if we take this problem, and we apply it to the model we described in Section 3.2.1, the consequences are much more noticeable. Not generating invariants in a model that size could lead to Cubicle not being able to prove safety. The issue with the concurrency pattern we described is that both DFS and BFS, Cubicle's current forward exploration techniques, struggle to get to the synchronization barrier often enough.

Lets look at a concrete example that implements the pattern we described in Figures 3.3 and 3.4. We create a model that we will run with three processes. We give a process four initial `branch` and `cmd` transitions (*i.e.* $k = 4$ in Figure 3.4), as well as four tasks decomposed into three sub-tasks each (i.e. $m = 3$ and $n = 4$ in Fig. 3.4) and set a synchronization barrier that forces each of the three process to be doing different tasks in order to be activated. We let BFS and DFS explore $500\,000$ states to see how often they visit the synchronization barrier. This is important because activating the barrier means having access to the potentially interesting transitions behind it. For $500\,000$ states, BFS visits the barrier **1512** times and DFS visits it **60** times. It is also important to note that backtracks can happen at any point in time- there are no guarantees that Cubicle will not backtrack, even if the proof has been running for a long time.

This is the reason why our examples in Section 3.2.1 fail. The models are too large for DFS and BFS to visit efficiently enough to help BRAB. One might suggest just letting the forward exploration run longer and letting it visit more states. Unfortunately, this leads to *too many* forward states. The irony is that too few states slow down BRAB, but so do too many states. The reason for too many states slowing down BRAB is twofold: (i) exploring too many states takes time, you do not want your forward exploration to go for hours in hopes of giving the Oracle enough information, and (ii) Cubicle has to compare every approximation with the set of forward states, so the bigger the set, the longer that takes, the slower BRAB becomes. Going back to our examples in Section 3.2.1 and Figure 3.5, there are certain models where BFS, for example, visits over a million states, and this has no impact on the result. It is also important to point out that BFS and DFS in Cubicle are optimized to account for symmetrical states. This means that for example if we have two states `X[#1] = True` and `X[#2] = True`, they are considered to be the same.

3.2.3 Buggy and unsafe models

Another problem is that, when it comes to Cubicle, models that are long and complex, like our example from Section 3.2, are sort of a no-win situation. When they are safe, a proof will take a long time, and when they are unsafe, a counter-example might also take a long time. Cubicle is designed to prove safety, and while it will give a counter-example should the system be unsafe, this can take an arbitrarily long time in huge systems. Cubicle is not specifically looking for a counter-example, it is trying to prove a system safe, meaning it is calculating everything we have mentioned earlier. This implies that, just as with backtracks, you could get hit with a trivial unsafe at any point in time.

The same goes for buggy systems, for example deadlocking ones. Cubicle is very strict in what it does- it tells you if a model is safe or unsafe according to the safety property (or properties) you asked it to check. Beyond that, Cubicle does not really care. If a model deadlocks, for example, well that is just too bad. And although a deadlock does not impact the safety proof result directly, it does impact the usefulness of a model.

Overall, we can summarize two issues facing Cubicle in its current form, and therefore two areas of interest:

- How do you visit enough states during a forward exploration to make sure that the Oracle has enough information to help BRAB
- How do you avoid running time-and-calculation-heavy proofs on systems that are potentially trivially buggy

There is also always the problem of expressiveness and language extensions - what can we model using Cubicle. This problem is never simple, be-

cause with Cubicle and its safety proving mechanism, any new language feature would need to be accompanied by a pre-image calculation, which can be arbitrarily difficult. Not to mention the underlying SMT solver.

In Chapter 4, we propose one common solution to both of these problems, called the *Cubicle Fuzzy Loop* (CFL). We will also see in Chapter 4 and subsequently in Chapters 5 and 7 that the inclusion of CFL not only allows us to address the above two issues, it also allows us to expand Cubicle's capabilities without worrying about pre-images and the SMT solver.

Chapter 4

The Cubicle Fuzzy Loop (CFL)

Contents

4.1 Fuzzing Cubicle 53
4.2 General CFL Structure 55
4.3 Fuzzing the Scheduler 61
4.4 BRAB & CFL: Experimental Results 64
4.5 Testing Models 67
4.6 Parameterized Fuzzing 69
4.7 Discussion: Heuristics, Decisions, Fuzzing . . 72
4.8 Discussion: Stability 73

The previous chapter showed that, while efficient at what it does, Cubicle does face certain limitations. This chapter describes how our integration of testing into Cubicle also provides a solution to these problems. We introduce the Cubicle Fuzzy Loop (CFL), a fuzzing-based extension for Cubicle. We show how CFL can serve as a forward-exploration technique for BRAB, as well as how it can be used to test models. We provide and discuss experimental results.

4.1 Fuzzing Cubicle

This chapter is an extended and more detailed version of our work on CFL presented in [61]. We delve into more detail about how we adapted fuzzing to Cubicle, the heuristics, as well as discussions relating to CFL.

CFL is inspired by AFL [62, 63]. AFL is a coverage-based mutational gray box fuzzer. This means that AFL takes inputs, mutates them, and checks how that affects coverage. If the mutated test led to more coverage, it is retained to be re-mutated later, else it is discarded. We retain several notions from

this that we want to incorporate into Cubicle: (i) inputs, (ii) mutation, and (iii) coverage. However, doing this is not straightforward, because Cubicle directly contradicts the possibility of new, and mutated, inputs.

Input AFL generates an input for the program it is testing. Transposing this to Cubicle means generating a state that will be considered as the initial state for the system. Cubicle models provide an initial state declaration, so we could simply generate states that correspond to the declaration. The initial formula of a model usually either fixes all variable values in a system or, if it does not, it only leaves a few variables undeclared. Therefore generating states that correspond to the initial formula is not interesting as, at most, it will only generate a handful of states, and they will all have the same form. Since our goal is to cover as diverse a state space as possible, having the same starting point for each exploration means that we become dependent *only* on the scheduler, which, as we saw in our example in Chapter 2 does not lead to the best outcomes. The alternative is to randomly generate states and start explorations from them. The problem with this is that we cannot guarantee that these states are *reachable* in the model. To be more precise, we could easily generate a consistent state, but without any guarantees that this states can be reached by executing the model starting from the model's declared initial state. Starting an execution from an unreachable state will likely add other unreachable states to the set of visited states. This will negatively impact Cubicle's invariant generation algorithm, preventing it from generating certain invariants.

Mutation Similarly to the input issue described above, we cannot mutate states for the same exact reachability reasons as before. This brings us even further away from how AFL functions.

Coverage As with AFL, if an execution ends up in a new interesting point in the model, we want to be able to get back to that point during another execution in order to explore further. Since programs are deterministic in AFL, the same inputs will bring you to the same points. However, our models are concurrent and nondeterministic: if we use the same input, we cannot guarantee that the scheduler brings us back to the same place. As a consequence, if we have no control over the scheduler, we have no control over how the model is explored.

Our solution to solve these problems is twofold. First, when we run a model exploration, we consider that any of the already visited states can serve as the initial state from which further explorations are run. This solves the problem of having reachable initial states. While this does diversify the state

space visited, it does not, however, solve the coverage problem. Our explorations still depend on the randomness of the scheduler, which, as already mentioned, is not sufficient. The only way we can affect how states are visited is by affecting the scheduler. The second part of our solution is, therefore, *fuzzing* the scheduler: from time to time, we influence how explorations are run in order to help the scheduler visit unexplored areas of the state space.

4.2 General CFL Structure

Before going into detail about how we fuzz the scheduler, we will first describe the general structure of CFL, from its environment to how it treats states.

In CFL, the states of a model are represented as dictionaries which map variables and arrays to their corresponding values. Assuming that a state of the model is noted as s, we say that the pair consisting of a transition t and its instantiated arguments $a_1, \ldots a_n$ is an *exit transition* from s if the guard of $t(a_1, \ldots a_n)$ evaluates to true in s.

CFL associates a record, called a *node*, to each visited state. Given a state s, the node associated to s contains the following fields:

- `state`: the dictionary s
- `count`: the number of times s has been visited
- `exit_num`: the number of *exit* transitions from s
- `exit_transitions`: an explicit representation of the *exit* transitions from s
- `exits_taken`: which *exit* transitions have already been taken from s
- `exit_count`: how many times each *exit* transition has been taken.

CFL uses all of this information to guide how it fuzzes the scheduler (Section 4.3).

The CFL algorithm is a specific implementation of the `FWD` function from Algorithm 2, used by the Bwd$(\mathcal{S}, U, d_{\max}, k)$ algorithm. Remember that $\mathcal{S}$ is the triplet $(\mathcal{X}, I, \tau)$ where $\mathcal{X}$ is the set of array symbols, I is a formula describing the initial states, and τ is the set of transitions, U is the unsafe state, $d_{\max}$ is the depth limit, and k is the number of processes to use. Currently, CFL does not make use of $d_{\max}$.

CFL keeps track of two sets of information: a set $\mathcal{P}$ of potential initial CFL nodes and a map $\mathcal{V}$ of visited model states to their corresponding nodes. Abstractly, CFL can be broken up into two stages: the initialization stage and the execution stage. The initialization stage can be summarized by the following steps:

Algorithm 3: Basic CFL Algorithm

```
1 function CFL(d_max, k) : begin
2     V := ∅;
3     P := init_system(d_max, k) ;
4     while not_empty(P) do
5         n := choose_node(P) ;
6         strat := choose_strategy() ;
7         V, P := explore_system(strat, n, k, V, P);
8     return V
```

- Compute the initial state s_0 of the model with k processes
- Compute the unsafe states with k processes
- Initialize the CFL node $init$ for the initial state s_0
- Add $init$ to $\mathcal{P}$ and add the mapping from s_0 to $init$ to $\mathcal{V}$

Initializing the initial state can be done in one of the three ways we described in Section 4.1. This would lead to $\mathcal{P}$ either containing one node, or multiple nodes (same as $\mathcal{V}$ containing one mapping or multiple mappings). Note that $\mathcal{P}$ and $\mathcal{V}$ are global variables and can be modified by any function.

The second stage, i.e. the execution stage, is a loop that repeats the following steps:

- Pick a node n from $\mathcal{P}$
- Pick a strategy $strat$ to explore the system (described further down)
- Apply $strat$ to n, gathering new visited states and their corresponding nodes and adding them to $\mathcal{V}$ and $\mathcal{P}$

This is repeated until CFL is stopped.

Now that we have presented how CFL works abstractly, we can go into more detail. Assuming that $\mathcal{X}$, I, τ, and U are global variables, the main procedure of the algorithm CFL($\mathsf{d_{max}}$, k) is given in Algorithm 3. We explain the procedure by detailing the auxiliary functions used.

Function init_system(k)

This function creates the original set $\mathcal{P}$ and map $\mathcal{V}$. It is given in Algorithm 4. The function takes the number of processes k as a parameter. It creates an

empty mapping s_0 (line 2), and then for each process i in the interval $[1, k]$ instantiates the quantified variables in the initial formula I with process i using lits_to_map(I, i).This instantiation creates a mapping, which is added to s_0. We do not give an explicit implementation for lits_to_map. Once s_0 has been created, a function check is called (line 5), which verifies that s_0 has a value for all model variables. If it does not, then an error is raised. After the check, a corresponding CFL node n to s_0 is computed using the function node(s_0, k), which we detail further down. The function init_system returns a set containing only s_0 and a map containing $s_0 \mapsto n$.

Algorithm 4: Initialize CFL System

```
1 function init_system(k) : begin
2     s0 := {};
3     foreach i in [1, k] do
4         s0 := s0 + lits_to_map (I,i)
5     check (s0);
6     n := node (s0, k) ;
7     return {s0 ↦ n}, {s0}
```

Function node(s, k)

This function, given in Algorithm 5, takes a model state s and the number of processes k and returns a CFL node initialized from s. It first calls a function exit_transitions (described further down) to compute a list of all possible exit transitions in state s (variable tr) on line 2. Then, node creates a CFL node, i.e. a record with the fields we described earlier. [61] [64] It uses a function map (not detailed) to iterate over the list tr, mapping each element, i.e. each exit transition, to 0.

Algorithm 5: Compute a CFL Node

```
1 function node(s, k) : begin
2     tr := exit_transitions(s, k);
3     return {
4     state = s;
5     count = 1;;
6     exit_num = length(tr);
7     exit_transitions = tr ;
8     exits_taken = ∅ ;
9     exit_count = map(tr, 0) }
```

Function exit_transitions(s, k)

The function exit_transitions is given in Algorithm 6. This function calculates the exit transitions in a state s for k processes. It takes a model state s and the number of processes k. It starts by initializing an empty list tr (lines 2). Then, for each pair of a transition t and its arguments $args$ (line 3), it computes all possible combinations of processes for the number of arguments of t (line 4). We do not detail the function all_combinations. Once that is done, it checks if t, with each combination, satisfies the state s, i.e. is possible in s. If it is, then the transition and specific argument combination are added to tr.

Algorithm 6: Calculate Exit Transitions

```
1 function exit_transitions(s, k) : begin
2     tr := [ ]
3     foreach t(args) in τ do
4         combs := all_combinations (args, k);
5         foreach c in combs do
6             if t(c) satisfies s then
7                 tr := t(c) :: tr;
8     return tr
```

Function choose_node$()$

This function selects a CFL node from $\mathcal{P}$. When a node is chosen, it is removed from $\mathcal{P}$. We do not detail this function, as in our current implementation, it simply picks a random element from $\mathcal{P}$.

Function choose_strategy$()$

This function picks an exploration strategy to apply, i.e. this function decides how the schedule will be modified. We do not detail this function either, as in CFL's current implementation, the choice of strategy is purely random.

Function explore$(strat, n, k)$

The function explore is the core function of CFL. It takes an exploration strategy $strat$, a node n, and the number of processes k. We give the function in Algorithm 7 and detail it step by step.

The function first chooses a number of steps $steps$ on line 2, and sets the current step $curr_step$ to zero. An exploration in CFL runs for a certain number of steps, randomly chosen each time. The constant bound is a limit on

Algorithm 7: Basic exploration technique

```
1  function explore(strat, n, k) : begin
2  |   steps := random_int(bound); curr_step := 0 ;
3  |   env := n ;
4  |   poss := env.exit_transitions ;
5  |   while curr_step < steps do
6  |   |   curr_step := curr_step + 1 ;
7  |   |   tr := pick_transition(poss, strat) ;
8  |   |   new_state := apply_transition(env, tr) ;
9  |   |   clean_exits(new_state, env, tr) ;
10 |   |   check_unsafe(new_state);
11 |   |   if new_state exists V then
12 |   |   |   env := modify_node(new_state) ;
13 |   |   |   poss := env.exit_transitions ;
14 |   |   else
15 |   |   |   n_1 := node(new_state, k) env := n ;
16 |   |   |   poss := n.exit_transitions ;
17 |   |   |   V(new_state) := n ;
18 |   |   |   P := P ∪ {new_state ↦ n_1}
```

how high CFL can go when choosing steps. In the current implementation, the bound is set to 100. The function initializes a variable env (line 3), which represents the environment of the current exploration. Initially, env is set to the node n. We declare a variable $poss$, which corresponds to the list of all possible exit transitions in the current environment (line 4). Then, while the current number of steps is less then the maximum, the function chooses a transition according to the chosen strategy (line 6), applies the chosen transition (line 7), does some bookkeeping (clean_exits, line 8, detailed further down), and checks that the resulting state is not unsafe (line 9). If the new state new_state has already been visited (i.e. there is a mapping in $\mathcal{V}$, line 10), then the environment is set to the already-existing node using a function modify_node (detailed further). Else, a new node n_1 and a mapping $new_state \mapsto n_1$ is added to $\mathcal{V}$ and n_1 is added to $\mathcal{P}$. We now detail the functions used in explore further.

Function pick_transition$(poss, strat)$

This function picks a transition from all possible exit transitions $poss$ to apply according to the current strategy $strat$. The function returns a transition and its arguments. We do not detail this function, as we will discuss fuzzing strategies in detail in Section 4.3.

Function apply_transition(n, tr)

This function takes a chosen transition tr and applies it to the model state stored in n. Recall that n is a CFL node, so the transition will be applied to n.state. This function is not detailed, as it simply creates a new mapping for the variables/arrays in n.state according to the transition tr and returns them.

Procedure clean_exits(s, n, tr)

This procedure, given in Algorithm 8, takes a model state s, a corresponding CFL node n and a transition tr (implicitly a transition and its arguments). When CFL encounters a state s that it has already visited, it does not need to recreate a new node, it only needs to modify the existing one.

Algorithm 8: Clean Exit Transitions

```
1 procedure clean_exits(s, n, tr) : begin
2     temp := {env with
        exits_taken = env.exits_taken ∪ tr;
        exit_count(tr) = env.exit_count(tr) + 1};
3     V(env) := temp;
4     if temp.exits_taken ≠ temp.exits then
5         P := P ∪ {temp}
```

When modifying an existing node, CFL needs to:

- Add the transitions tr to the field exits_taken (if it was not already added)
- Increment how many times tr has been taken.

Then it needs to modify the mapping in $\mathcal{V}$ for s. Recall that when a node is chosen from $\mathcal{P}$, it is removed from the set. This procedure checks if there are still exit transitions that have not been explored (line 4) for this node. If so, then the node is added back to $\mathcal{P}$.

Function check_unsafe(s)

This function checks whether the state s contains an unsafe state. If it contains an unsafe state, then CFL does one of two things:

- If CFL is running for BRAB, it stops completely and informs the user that an unsafe state was reached. It also generates a trace.
- If CFL is running as a standalone tool, it raises a warning that an unsafe state was reached, but does not stop its execution.

Function modify_node(s)

This function, given in Algorithm 9, is only called when CFL encounters a model state that it has already seen. Recall that nodes keep track of how often a model state has been visited. This means that when a model state is re-visited, CFL has to modify its node to increment the counter. The function modify_node takes a model state s, finds its associated record (node) in $\mathcal{V}$, and creates a new, temporary node $node$ with all of the same values, except with an incremented count (line 2). Then it modifies the mapping of s in $\mathcal{V}$ to set it to $node$ (line 3). It then returns $node$.

Algorithm 9: Modify Node

```
1 function modify_node(s) : begin
2     node := {V(s) with count = node.count + 1} ;
3     V(s) := node;
4     return node
```

4.3 Fuzzing the Scheduler

Having described CFL's general structure, we can now describe how we treat the scheduler. We established in Section 4.1 that we have no choice other than to fuzz the scheduler, since we cannot touch the states. However, fuzzing the scheduler can mean many things, all of which we tested experimentally before settling on our solution.

Suggestion 1: Randomly generate schedules Randomly generating schedules means randomly generating sequences of transitions for Cubicle to execute. This, unfortunately, does not work mainly due to Cubicle transitions having requirements. Similarly to how fuzzing with randomly-generated inputs does not work very well because many end up syntactically invalid, randomly generated transition sequences also end up invalid. There are no guarantees that the transitions can be taken, meaning that there are too many useless sequences.

Suggestion 2: Take existing (correct) sequences, and change random steps This would work similarly to mutation-based fuzzers. You have a correct input and you mutate it to create a new input. There are two ways this can be done:

- change a random step $step$, leaving the rest of the sequence unchanged, and re-execute the sequence from the beginning

- pick a step $step$, change it, and re-execute the sequence from the beginning, calculating new valid steps when you get to $step$

There are multiple reasons as to why this either does not work, or is inefficient. The first way does not work simply because you are replacing a transition randomly, so you are once again faced with the problem of invalid requirements. The second way will either not work or work inefficiently. If $step$ is chosen randomly, then you run the risk of facing the same invalid requirement problem. However, if $step$ is picked from valid, possible transitions, then this method works, but uselessly reruns (i.e. recalculates) the same parts of the sequence multiple times, bringing in no new information. The biggest problem with suggestion 2 however, in both cases, is having to rerun the sequence to get to the chosen $step$, which directly contradicts Section 4.1, where we said that we will retain visited states to use as inputs. So suggestion 2 does not work however you spin it. The only remaining solution, and the one we implement in CFL, is to fuzz the scheduler by altering how it makes decisions when exploring the system.

Having settled on "decision-altering" as our chosen technique to fuzz the scheduler, the only missing piece is the heuristics behind the technique, i.e. what possible ways could the scheduler explore the system. Currently, CFL implements **six** exploration techniques.

Technique 1: Random Exploration This is the simplest exploration technique and is basically randomly choosing and applying transitions. Note that CFL randomly chooses transitions only from the **possible** transitions in a state, not from all transitions in the system. When this technique is picked, CFL randomly chooses a number of steps to execute and then applies random transitions to the starting node for that many steps. CFL can stop prematurely if it encounters a deadlock. The reason for this technique is that throughout our experiments, we noted that random execution, on its own, is often fairly efficient in exploring the system. It is also rather fast, since it does not spend time calculating anything. Overall, while it provides no guarantees on how diversely it will explore a model, it does provide a good basis for further exploration.

Technique 2: Process Sequences Interesting behaviors in models can sometimes be linked to the fact that it is the same process acting multiple steps in a row. While we can assume that with random exploration sequences of the same process would eventually appear, this becomes less likely the bigger the system and the more processes there are. We therefore include this technique, where CFL applies transitions to only one process consecutively. CFL selects a random process, picks a number of steps,and moves only that

process forward for that amount of steps. This stops prematurely if the chosen process has no more moves (note that this is not equivalent to a system deadlock, since CFL is only looking at one process).

Technique 3: Weighted Decision The goal of CFL is to explore the system as diversely as possible, while potentially forgoing exhaustivity. This implies that it needs to privilege coverage, i.e. visiting states it has never seen before. With this technique, CFL grades potential steps using the following criteria:

1. This step will lead me to a never visited state
2. This step means taking a transition that has never been taken by anyone globally
3. This step means taking a transition never taken from this node

These criteria are in order of importance - being able to visit a state that has never been visited will outweigh the rest.

Technique 4: Maximizing Randomness This technique aims at visiting the states that lead to greater uncertainty in how the system will behave. It privileges states that have the most potential outcomes, i.e. states that have the most *exits*. Since the goal is to cover as diverse a space as possible, the logic is that if you target states that have more exits, you will cover vaster state spaces. Assume that CFL is in a state s. It chooses a random transition from all **possible** transitions in s. It applies that transition and calculates how much randomness the resulting state will bring. If the result is greater than the randomness provided by s, then CFL will agree to that transition and move to the new state. Otherwise, the transition might be rejected. CFL will always accept transitions that lead to greater randomness. However, always rejecting states with equal or lower randomness means running the risk of ignoring parts of the system. This is why a certain percentage of the time (set by the user), CFL will accept transitions that lead it to states with equal to or lower randomness than s.

Technique 5: Limited BFS This technique is fairly straightforward: CFL picks a very small random depth and runs BFS. This acts as an occasional countermeasure to CFL's more depth-oriented explorations.

Technique 6: Unused Exit This technique serves a more cleanup purpose by simply picking one exit that has never been taken and covering it.

4.4 BRAB & CFL: Experimental Results

As described in Section 3, BRAB utilizes a forward exploration to construct a set of seen states to use as an Oracle. This method hinges on the set of forward visited states, $\mathcal{M}$, being diverse enough and including crucial states which would allow the Oracle to correctly decide if an approximation is good or not. The whole idea of CFL is that it sacrifices exhaustivity in order to explore a larger variety of states. We decided to plug CFL into Cubicle not only as a standalone testing tool, but as the Oracle for BRAB.

We run our benchmarks on the examples discussed in Chapter 3. All of our examples have 25 transitions to represent depth, branching, and pipelining, as well as one synchronization transition which requires that processes be in different configurations throughout the model.

We compare several forward exploration strategies with CFL: (i) Cubicle's existing BFS and DFS strategies, both optimized for speed, (ii) a random exploration strategy, i.e. one that starts at the initial state and randomly chooses transitions, and (iii) CMurphi, an enumerative model checker [65] developed on top of Murφ, only used here to efficiently visit the state space. The results of this comparison, excluding CMurphi, can be seen in Figure 4.1. We discuss CMurphi separately further down.

Each strategy is run for three processes and has the same amount of time allocated for its forward exploration, noted in the Forward Time column. We then compare how many states were visited (States column) and whether Cubicle was able to prove safety before hitting the timeout criteria (Safe column). The total time (forward + proof) is noted in the Total Time column for each strategy. Each example was timed out after 5 minutes. This was chosen due to the time taken using CFL, as well as the number of proof nodes generated by Cubicle within those 5 minutes, compared in Figure 4.2. The values underlined and in bold are where Cubicle was successful in proving safety. We can see that the number of nodes for the timed out examples is much higher than is necessary for Cubicle in the cases where it quickly proves safety.

The reason CMurphi is excluded from Table 4.1 is due to the fact that we were unable to find an option that would force CMurphi to run for the allocated time. For each of our models, CMurphi raised the following error: "Internal Error: Too many active states." For the sake of fairness, we rerun CFL, manually setting the limit for each model to how many states were visited by CMurphi. The results for this are seen in Figure 4.3. The results for CFL all have the form `X/Y`. This is due to CFL's innate randomness. Two executions will not necessarily have the same results, especially if the allocated time/number of states to visit is low and the model is large. For example, in Table 4.3, Dekker was run 10 times, and all 10 times CFL managed to visit enough states to help Cubicle quickly prove safety. However, on a model like Germanish4, which is

Model	Forward Time	BFS			DFS			Random			CFL		
		States	Safe	Total Time	States	Safe	Total Time	States	Safe	Total Time	States	Safe	Total Time
Dekker	10s	466K	T.O.	-	605K	Yes	12.72s	266K	Yes	11.74s	120K	Yes	10.61s
Germanish	10s	424K	T.O.	-	593K	Yes	12.91s	261K	Yes	11.94s	120K	Yes	10.78s
Germanish2	10s	315K	T.O.	-	515K	Yes	12.26s	244K	Yes	11.92s	115K	Yes	10.75s
Germanish4	10s	287K	T.O.	-	547K	Yes	14.54s	186K	T.O.	-	110K	Yes	11s
German	10s	312K	T.O.	-	547K	Yes	16.25s	207K	Yes	13.55	107K	Yes	12.23s
German_Baukus	10s	359K	T.O.	-	591L	Yes	14.82s	207K	Yes	12.93s	105K	Yes	12s
German_CTC	50s	1 429K	T.O.	-	2 010K	Yes	62.81s	505K	T.O.	-	265K	Yes	55.17s
German_pfs	10s	416K	T.O.	-	431K	Yes	17.37s	174K	Yes	12.69s	100K	Yes	13.11s
Szymanski_at	10s	372K	T.O.	-	534K	T.O.	-	155K	Yes	11.92s	105K	Yes	11.60s
Szymanski_na	10s	270K	T.O.	-	483K	T.O.	-	270K	T.O.	-	100K	Yes	12.50s
Bakery_lamport	40s	1 565K	T.O.	-	2038K	T.O.	-	650K	T.O.	-	230K	Yes	42.59s
Flash_no_data	40s	862K	T.O.	-	1 048K	T.O.	-	273K	T.O.	-	140K	Yes	43.32s

Figure 4.1: Comparing CFL with Different Forward Strategies.

Model	BFS	DFS	Random	CFL
Dekker	6904	**4**	**4**	**4**
Germanish	889	**4**	**4**	**4**
Germanish2	1770	**4**	**4**	**4**
Germanish4	2415	**20**	3255	**20**
German	2862	**41**	**41**	**41**
German_Baukus	2170	**41**	**41**	**41**
German_CTC	1500	**61**	1231	**60**
German_pfs	1121	**44**	**44**	**44**
Szymanski_at	2861	174	**33**	**33**
Szymanski_na	2061	210	510	**43**
Bakery_lamport	779	2189	230	**16**
Flash_no_data	1329	61	1227	**37**

Figure 4.2: Number of Generated Proof Nodes for Each Strategy

Model	CMurphi		CFL	
	States	**Safe**	**States**	**Safe**
Dekker	48K	T.O.	48K	10/10
Germanish	48K	T.O.	48K	10/10
Germanish2	39K	T.O.	39K	10/10
Germanish4	39K	T.O.	39K	7/10
German	33K	T.O.	33K	6/10
German_Baukus	33K	T.O.	33K	7/10
German_CTC	24K	T.O.	24K	0
German_pfs	33K	T.O.	33K	6/10
Szymanski_at	32K	T.O.	32K	3/10
Szymanski_na	26K	T.O.	26K	2/10
Bakery_lamport	32K	T.O.	32K	1/10
Flash_no_data	21K	T.O.	21K	3/10

Figure 4.3: Comparison with CMurphi

longer and more complex, running CFL 10 times only led to seven quick successes. We will discuss this further in Section 4.8.

4.5 Testing Models

As we mentioned earlier, large and unsafe models are just as problematic as large and safe models. To demonstrate this fact, we take our previous models from Section 4.4, and we introduce trivial unsafe states. We make sure that these unsafe states appear in the part of the model that comes after the synchronization barrier. We then run Cubicle (without BRAB) and CMurphi on these examples in order to compare the results. We set the timeout to five minutes. The results are given in Figure 4.4. Both Cubicle and CMurphi fail at finding the unsafe state. The reason why CMurphi fails is the same as before, where it raised an internal error: "Internal Error: Too many active states.". The timeout results for Cubicle are not surprising, as the MCMT framework that Cubicle implements is made to *prove* safety. There is nothing in MCMT that makes it efficient at quickly finding that a model is unsafe. Showing that a model is unsafe implies finding a path from the unsafe state to the initial state. In large models, this path can be arbitrarily long and complex. This is where CFL comes in.

Model	**Backward**	**CMurphi**
Dekker	T.O.	O.M.
Germanish	T.O.	O.M.
Germanish2	T.O.	O.M.
Germanish4	T.O.	O.M.
German	T.O.	O.M.
German_Baukus	T.O.	O.M.
German_CTC	T.O.	O.M.
German_pfs	T.O.	O.M.
Szymanski_at	T.O.	O.M.
Szymanski_na	T.O.	O.M.
Bakery_lamport	T.O.	O.M.
Flash_no_data	T.O.	O.M.

Figure 4.4: Unsafe: Backward and CMurphi

CFL can be used for more than just running the forward exploration for BRAB. We can use it as a standalone tool to test models. Not only that, it also allows us to improve Cubicle's error-detection. For example, it is very easy to

extend CFL to *deadlock detection*, something that MCMT also does not do. We discuss deadlocks further down. We run CFL in a standalone fashion on the same models. The results can be seen in Figure 4.5. Once again timeout was set to five minutes.

Model	**CFL (standalone)**
Dekker	0.3s
Germanish	0.7s
Germanish2	0.2s
Germanish4	0.7s
German	0.4s
German_Baukus	0.4s
German_CTC	0.5s
German_pfs	0.3s
Szymanski_at	2s
Szymanski_na	2s
Bakery_lamport	1.5s
Flash_no_data	3s

Figure 4.5: Unsafe: Backward vs. CFL

We can see that CFL is quickly able to find the unsafe states that we injected, with even the largest model - the dataless Flash - taking only three seconds.

We use CFL to introduce deadlock detection into Cubicle. Within MCMT, the only viable way to detect a deadlock would be to know in advance what the deadlocking state is, and check whether it is reachable or not. This method is less than ideal in real life. A deadlock in CFL is defined as a state where there are no more exit transitions possible. Every time CFL calculates $poss$ from Algorithm 7, if that number is equal to zero, then CFL raises a warning that it has encountered a state that is impossible to leave. This means that CFL can detect multiple deadlocks, or it can detect the same deadlock multiple times, if it revisits the deadlocking state more than once. In its current implementation, CFL does not take into account whether it has already signaled a state as deadlocking, it simply notes every state (and path) that contains a deadlock.

We artificially deadlocked our examples from Section 4.4. We run CFL with

Model	CFL
Dekker	0.1ms
Germanish	0.5s
Germanish2	0.2s
Germanish4	0.5s
German	0.4s
German_Baukus	0.4s
German_CTC	0.4s
German_pfs	1s
Szymanski_at	2s
Szymanski_na	0.6s
Bakery_lamport	2s
Flash_no_data	4s

Figure 4.6: Deadlock Detection

three processes to see if it can detect the deadlocks, and, if so, how long it takes. The results are given in Figure 4.6. As we can see, even on huge models, finding deadlocks does not take CFL too long.

4.6 Parameterized Fuzzing

As mentioned numerous times before, Cubicle is a parameterized model checker and proves safety for n processes. However, CFL uses a **fixed** number of processes, provided by the user. One of our main goals is to instrument CFL so that it is capable of **independently** deciding how many processes it needs to explore a model without missing any *behaviors*. When discussing system behaviors in [66], Leslie Lamport states "*Formally, we define a behavior to be a sequence of states, where a state is an assignment of values to variables. We specify a system by specifying a set of possible behaviors—the ones representing a correct execution of the system.*" Unfortunately, according to this definition of "behavior", any time CFL decides to add a process, new behaviors are generated. This makes it difficult to decide when to stop, as new array values do not necessarily imply actually new *behavior*. If we take the example in Figure 4.7, adding processes does nothing to change the overall behavior of the system. Since we cannot easily focus on variables and values, we try the only other alternative: transitions.

```
type t = A | B

array X[proc] : t

init(i) { X[i] = A }

transition t1(i)
{ X[i] := B }
```

Figure 4.7: Same Behaviors with Multiple Processes

The easiest way to define a new behavior with regards to transitions is a transition being possible with $n+1$ processes, while being impossible with n processes. This is too simple, however, because a process's behavior is not only defined by the transitions it takes, but also the order in which it does. Sometimes we want to focus more on the path instead of the destination. Analyzing all combinations of all paths would be too expensive, so instead we focus on pairs of transitions, i.e. "is it possible to go from transition `t1` to transition `t2` directly".

Lets look at the example in Figure 4.8. In this example, all transitions are eventually possible with three processes. However, three processes never allow for the sequence `t3 -> t4`. If we have three processes #1, #2, and #3 and we took transition `t3`, given the requirements and the actions, we get the state:

```
P2[#1] = False && P1[#2] = A && P2[#2] = True && P1[#1] = B
```

for #1 and #2 and values we do not know for #3. If we now want to take transition `t4`. In our resulting state from `t3`, `P1[#1] = B`, so we will say that #1 corresponds to `i` in `t4`'s parameters. We needs a `P2[j] = False` and since #2's is equal to `True`, we can only assume that `P2[#3] = False`. That still leaves the last condition: `P2[k] = False`, and with three processes, we are out of options. However, if we had four processes, the transition would be possible.

This is the current logic behind how CFL decides the number of processes it needs. It first preprocesses the model to set the minimum number m of processes needed depending on how many parameters the unsafe states and transitions have. For Figure 4.8, the minimum number is three. Then CFL runs Technique 5 from Section 4.3. The reason is that this technique is supposed to cover transitions better since it targets states according to how many transitions leave them. It starts with the minimum number of processes, and runs

heuristic.cub

```
type t = A | B | C

array P1[proc] : t
array P2[proc] : bool

init(i) { P1[i] = A && P2[i] = True }

transition t1 (i)
requires { P1[i] = A }
{ P2[i] := False }

transition t2 (i j)
requires { P2[i] = False && P1[j] = A}
{ P2[j] := True;
  P1[i] := C }

transition t3 (i j)
requires { P2[i] = False && P1[j] = A }
{ P2[j] := True;
  P1[i] := B; }

transition t4 (i j k)
requires { P1[i] = B && P2[j] = False
             && P2[k] = False }
{ P1[i] := A ;}
```

Figure 4.8: Cubicle Code to Test Fuzzer Parameterization

the exploration for a fixed number of steps. While it runs, it counts how many passages there were between each pair of transitions. When it is done, it saves the result, and does the same exact thing but with $m+1$ processes. Then, it compares the results for m and $m+1$. If a pair of transitions goes from 0 to anything greater than zero, CFL stores the results for $m+1$, and reruns with $m+2$, repeating the same processes. If nothing new appeared between two values of m, then CFL keeps the smaller of the two. If ask CFL to evaluate the example in Figure 4.8, it returns fours processes.

This is the preliminary work to render CFL parameterized. This current implementation works well for small and medium-sized models. This method suffers on larger models, because the forward exploration used to evaluate behaviors is very limited.

4.7 Discussion: Heuristics, Decisions, Fuzzing

The six exploration strategies given in Section 4.3 are heuristics used to explore the state space using CFL. And while they are the only ones currently implemented in CFL, because they are heuristics, there is no limit on what could be added as an exploration technique. For example, heuristics that focus more on the processes themselves could be useful to diversify state space exploration as well as testing model behaviors. For example processes could be introduced, or "killed". We discuss this further in Chapter 5 with the implementation of threads in CFL.

Another way to influence CFL is to directly act on how it picks both nodes from $\mathcal{P}$ and strategies to apply. Currently, both of these things are done purely randomly. If we take our inspiration, AFL, when it picks inputs to mutate, it privileges those which have not yet been touched. A potential optimization to CFL is setting priorities to the nodes in $\mathcal{P}$. Several direct ways to prioritize nodes are:

- Pick nodes according to how often they have been used
- Have the user set interesting states and pick nodes that contain those states
- Have the user tag important transitions and then prioritize nodes either according to which nodes would allow these transitions to be executed or which nodes are the results of these transitions

Influencing how strategies are picked is more complicated and would generally require more computation, slowing down CFL. A basic way to influence strategy selection would be either privileging certain strategies (user-defined) or making sure that every strategy is chosen equally. A less basic way would

be analyzing which strategies are more effective under which circumstances, and choosing accordingly.

In its current form, CFL sits between white box and gray box fuzzing. This is due to it having initial knowledge of all transitions in the system, i.e. the program structure. What brings it closer to gray box fuzzing is that it does not use a systematic exploration of the system. If we take AFL for example, it knows when it has covered something new, but it does not know what remains to be covered. Whereas with CFL, if we look at it from a transition perspective, it is aware of the model structure. That being said, if we look at CFL from a state perspective, it is closer to AFL and therefore gray box fuzzers, since it knows the states that it has visited, but has no knowledge of which states remain unseen. Getting closer to white box fuzzing is costly, as the more computationally intensive CFL becomes, the slower it gets. As is, CFL is slower than BFS and DFS in how many states it visits in the same amount of time. CFL makes up for that by visiting more interesting states.

4.8 Discussion: Stability

In Section 4.4, we saw that we had to rerun CFL multiple times when comparing it with Murphi. This is due to CFL's inherent randomness. Nodes are picked randomly, strategies are picked randomly, transitions are sometimes picked and applied randomly. Overall, CFL is significantly dependent on *random*.

Lets extract a couple of examples from our previous tables and run some experiments to illustrate the issue with this. We focus on five examples: Dekker, Germanish, German, Szymanski_at, and Flash_no_data. The reason we have picked these five is because they differ in size and complexity, ranging from simple (Dekker) to very complex (Flash_no_data). Note that these are the examples from our benchmarks, meaning they are all prefixed with our concurrency pattern from Chapter 3.

In our BRAB benchmarks, we let BFS, DFS, and CFL run for a set number of seconds. If we lower that number, our results change. For the purposes of this section, we will only be focusing on DFS and CFL, since BFS could not handle any of the examples.

For these experiments, we classify a "success" as Cubicle being able to prove safety.

We run DFS and CFL for **one second** on Dekker and they both succeed. We rerun them and they succeed again. DFS, being deterministic, will always succeed if it visits the same amount of states in the allocated time period. We ignore potential hardware discrepancies and assume that the same time rerun multiple times will always result in the same result for DFS. But what about CFL? We rerun Dekker again. DFS succeeds, but suddenly CFL fails. If we run CFL 100 times on Dekker, it will succeed **99** times. But it will fail once.

If we run CFL on Dekker for two seconds, CFL succeeds 100 out of 100 times.

Moving on to Germanish. We redo the same experiment, giving CFL and DFS **one** second. This time, DFS fails and CFL succeeds. We rerun CFL 100 times. Out of those 100, it fails 11 times. We run CFL for two seconds on Germanish- it succeeds 96 out of 100 times.

Skipping directly to Flash_no_data, running CFL for one second only results in one success out of 40 tries. CFL succeeds in two seconds, but only **two** out of 10 times.

We stated earlier that CFL was slower than BFS and DFS in visiting states, so maybe time is a bad comparison, maybe we should instead focus on the number of states, ignoring how long it takes to visit them.

We restart our experiments. We tell CFL to visit 5000 states with Dekker, which allows it to succeed only **one** out of 10 times. If we tell it to visit 50000 states, it succeeds 100 out of 100 times. DFS fails even with 52000 states, and since we know that DFS is deterministic, there is no point in rerunning it to see if it will succeed with 52000 states during a different run.

For Germanish, if we give CFL 5000 states, it succeeds **four** out of 10 times. However, if we give it 60000 states, it succeeds 100 out of 100 times. For German, 5000 states results in **two** successes out of 10, whereas 100000 states results in 100 out of 100 successes.

Szymanski_at and Flash_no_data are another story. Both models are very large. Running CFL for 5000 states with Szymanski_at gives one success out of 10 runs, whereas Flash_no_data only succeeds once out of 45 runs. And contrary to the previous three, these models are so big that even getting succeeding 100 out of 100 times does not guarantee anything, since running them for another 100 times occasionally leads to failures. All of this is to illustrate CFL's biggest hurdle and its most promising future work (parameterization aside): **stability**.

Fuzzers, like any testing technique, offer no guarantees. Rerunning the same tests will not necessarily lead to the same results. Running many tests, or running fuzzers for a long time, is optimal. This can, unfortunately, also be applied to CFL. The same amount of time, the same number of visited states, and the same model do not guarantee the same results. The easiest way to mitigate this problem is to let CFL run for a longer time. In the context of BRAB, this is not ideal, since it directly influences the invariant generation. However, if CFL is running purely to test a model, it is less troublesome.

The interesting thing about this problem is that, reproducibility of results aside, CFL is capable of succeeding with very little visited states. Managing to prove Flash_no_data with 5000 states is impressive, considering that the model has 95 transitions.

The only way to tackle CFL's stability is to tackle how strategies are chosen. If we analyze what led to CFL succeeding with so few states in these

particular cases, perhaps we could extract a pattern or a rule/heuristic to apply. We mentioned in Section 4.7 that influencing how exploration techniques are chosen would slow CFL down. However, in CFL's current state, we do not see another way of tackling stability.

Chapter 5

CFL-Based Threads

Contents

5.1 Primitives 79

5.2 Cubicle vs. Primitives 80

5.3 Language Extensions & Semantics 82

5.3.1 Distinguishing Threads and Processes 82

5.3.2 Initial States and Unsafe States 83

5.3.3 Locks and Reentrant Locks 84

5.3.4 Conditions 87

5.3.5 Semaphores 88

5.3.6 Restrictions 89

5.4 Examples 89

5.4.1 Producer-Consumer 89

5.4.2 Dining Philosophers 91

5.4.3 Other Examples 97

Now that we have introduced CFL and shown how it allows us to test Cubicle models and explore state space more efficiently for BRAB, we can focus on its other big advantage: CFL makes extending Cubicle easier. Extending Cubicle (without CFL) is never straightforward. Adding new features can only be done if these features can be handled by the underlying SMT solver and if we know how to calculate their pre-images. Both of these tasks are, sadly, nontrivial. CFL, on the other hand, has no such limitations as it simply executes forward. This is not to say that adding things to CFL is completely trivial- we still need to account for Cubicle's underlying semantics. But it is more straightforward than the classic Cubicle approach. It is with this in mind that we decided to add a significant new language feature into Cubicle- threads.

One of the most widely used techniques for concurrent programming is based on *execution threads* (or simply *threads*) [67, 68]. Within a program, a thread is the smallest sequence of instructions that can be scheduled and run concurrently with other threads. Program threads share memory and resources, which enables fast communication between them, improving application performance.

Multithreaded programming is difficult and error-prone. There are two main reasons for this: shared data between threads and nondeterminism introduced by the scheduler. This subtle combination can lead to issues such as:

- Race conditions: multiple threads access a share resource simultaneously
- Synchronization problems: the order the threads execute in needs to be controlled for consistent results
- Deadlocks: a group of threads is blocked because each one is waiting for another to release a resource it needs

To avoid these problems, developers implement specific design patterns for concurrent programs (*rendezvous*, *barriers*, *read-write locks*, etc.) which require mastering appropriate synchronization primitives like *mutexes*, *conditions*, and *semaphores* [67].

Unfortunately, no matter how careful people are when writing concurrent programs, there are always bugs. Many bugs are often due to subtle interleavings between threads. This means that testing concurrent programs means executing these very precise interleavings, which implies having control over the scheduler. It is generally not possible to control the scheduler, so people often settle for re-executing the same code with the same input data, hoping that something different will happen, or they introduce primitives like `sleep` or `yield` into their code. But even with all of that, testing concurrent programs remains difficult.

We think that a way to check these types of programs is to abstract away many implementation details (present in programming languages) and focus solely on the **essential** aspects of a concurrent algorithm, namely the logic and the complex interleavings. In other words, what we want to be able to do is write a concurrent program in Cubicle, test it using CFL until no weird behaviors remain, and then proceed to actually coding it. This should eliminate any bugs that are introduced during the conception phase of the algorithm.

5.1 Primitives

As mentioned, concurrent programs often have the same reoccurring synchronization mechanisms: simple locks (mutex), conditions, and semaphores. *Locks* are a basic synchronization primitive where only one thread can have access to a resource at a time. The operations associated to locks are *acquire* and *release*. In our case, we consider blocking locks: if a thread tries to acquire a lock held by someone else, it is blocked and suspended until the lock is released. The lock follows two rules:

Acquire Rule: For a thread t attempting to acquire a lock:

- If the lock is free, t acquires the lock and proceeds.
- Otherwise, t is blocked (i.e. suspended until it can gain access to the lock)

Release Rule: For a thread t releasing a lock:

- If there are suspended threads waiting, pick one to give the lock to
- Otherwise, the lock is free

Conditions are mechanisms that allow threads to wait for a specific condition to be true before continuing. A *condition* consists of a lock and an associated wait-queue. When a thread acquires the lock, it checks if the necessary condition to continue is satisfied. If it is not, the thread releases the lock and joins the wait-queue (going dormant). When a thread is eventually in a state that meets the condition's requirements, it can signal one of (or all of) the other threads to wake them (it) up. Conditions follow the following rules:

Acquire and Wait Rule: For a thread t attempting to acquire a condition:

- Acquire a condition lock
- If the condition is true, proceed to the next rule
- Otherwise, release the lock and join the wait-queue (t is blocked)

Signal Rule: For a thread t that has acquired a condition which is true

- Notify one or all waiting threads and proceed

Semaphores maintain a counter to symbolize how many accesses to a resource are possible. This allows multiple threads to gain access to the same resource. The semaphore's counter is a non-negative number symbolizing how many possible accesses/resources there are. Each time a thread acquires the semaphore, the counter is checked- if it is greater than zero than the thread gains access to the resource, else, if it is equal to zero, the thread is suspended. When a thread releases a semaphore, the counter is incremented, and a random suspended thread (if there is any) is woken up. Summing up, semaphores following the following rules:

Acquire Rule: For a thread t attempting to acquire a condition:

- If the semaphore counter is greater than zero, decrement the counter

by one and continue
  - Otherwise, thread t is blocked until the counter becomes positive

Release Rule: For a thread t releasing a semaphore
  - Increment the counter by one
  - Wake up one of the suspended threads (if there are any)

We will now explain that, while it is possible to encode most of these constructs in Cubicle's current language, it is necessary to introduce built-in primitives.

5.2 Cubicle vs. Primitives

Most of the features mentioned in Section 5.1 can be encoded in Cubicle's current language. For instance, encoding a semaphore would mean having an integer variable `S` and an array `WaitS` that maps processes to a boolean, indicating whether or not that process is waiting for S. But it would also mean adding a user-defined type `status` and an array `Status` to indicate whether a process is alive or suspended. Overall, encoding a semaphore, and its corresponding rules, as described in Section 5.1, would look like Fig. 5.1.

Each semaphore would require four transitions:

- One transition to acquire a semaphore when there are still free resources.
- One transition to try to acquire a semaphore when there are no available resources. This transition has to suspend the process and register that that process is now waiting for the semaphore.
- One transition to release a semaphore that no one is waiting on.
- One transition to release a semaphore that someone is waiting on. This transition picks a random waiting process.

Ignoring the obvious problem of modeling complexity, i.e. that this encoding unnecessarily clutters up a model adding unnecessary interleavings to check, this method of treating the desired concurrency primitives has two main issues.

The first is that Cubicle has no notion of "*specific process executing a specific task*" - a transition modifying an array indexed by a process does not mean that *that* specific process called the transition, it just means that something happened in the system that applied that transition to that process. This is directly contradicting to what we want with the concurrency primitives, because we need to know precisely who acquired and released what.

semaphore.cub

```
type status = Awake | Suspend

var S : int
array WaitS[proc] : bool
array Status[proc] : status

init(i) { WaitS[i] = False && Status[i] = Awake
          && S = (*available resources*)}

transition acquire (i)
requires { Status[i] = Awake && S > 0 }
{ S := S - 1 }

transition suspend (i)
requires { Status[i] = Awake && S = 0 }
{ Status[i] = Suspended;
  WaitS[i] := True }

transition release (i)
requires { Status[i] = Awake &&
           forall_other j. WaitS[j] = False }
{ S := S + 1}

transition awaken (i j)
requires { Status[i] = Awake && WaitS[j] = True }
{ Status[j] := Awake;
  WaitS[j] := False }
```

Figure 5.1: Encoding a Semaphore in Cubicle

The second problem is the lack of actual waiting queues in Cubicle. Earlier we simulated a process waiting for a semaphore by adding a `WaitS` array. However, waiting means being *blocked*, except Cubicle has no notion of *blocking* processes. In the same vein as not knowing which process does what, processes are either active in the system, or they do not exist. There is no intermediate status of *temporarily incapable of doing anything*, also because "doing something" means nothing for processes in Cubicle. The two issues come hand in hand- in order to block processes we need to be able to distin-

guish them and their actions. And by definition, Cubicle processes are indistinguishable, replicated components.

We therefore introduce threads to Cubicle to solve this problem without breaking the existing process mechanism. Threads in Cubicle bring with them the notion of *who* does what, and along with that a corresponding active and suspended status. This allows us to add the built-in concurrency primitives discussed in Section 5.1.

5.3 Language Extensions & Semantics

The built-in primitives that we include are (blocking) locks, reentrant locks, conditions, and semaphores. Along with these, we introduce threads into Cubicle. Threads come with an internal state that indicates their status (whether they are alive or suspended) and their *subtype*. Subtypes are an easy way to tell threads that they can only do certain actions. For example, in a typical Producer-Consumer example, some threads execute the producer procedures, and some execute the consumer procedures. If all threads do one of the two, that leads to a system deadlock, since there are either no values being produced or no values being consumed.

5.3.1 Distinguishing Threads and Processes

We stated earlier that processes in Cubicle are indistinguishable, and that Cubicle has no notion of "process `i` executed action `A`". Even in CFL, Cubicle's core semantics apply, meaning that when CFL applies a transition to a process, that does not entail that the action was completed by the process. This concept does not really work with threads, as there needs to be a specific way to distinguish threads and the actions they take. To solve this problem we modify Cubicle's semantics for threads and introduce an *actor* for each transition. Syntactically, our transitions now look like Figure 5.2 below.

```
transition transition_name ([i])
requires { guard }
{ actions }

transition transition_name ([i] j ...)
requires { guard }
{ actions }
```

Figure 5.2: Cubicle Transition Actor Syntax

The thread that executes the transition is marked in square brackets. Each transition can only have one actor, but the transition itself can affect multiple processes, as seen in the second transition declaration.

Another element needed to distinguish threads is thread types. Imagine a simple Producer-Consumer algorithm. Some threads will produce, and some threads will consume. Except in CFL, since all threads are indistinguishable, they could all decide to become producers (similarly, consumers). We introduce process subtypes into Cubicle, as seen in Figure 5.3 Currently a process

```
type subtype1 < proc
type subtype2 < proc

array X[proc] : bool

init (i) { X[i] = False }

transition t1 (i : subtype1)
{ X[i] := True }

transition t2([i : subtype2] j)
{ X[i] := True }
```

Figure 5.3: Cubicle Proc Subtype Syntax

can only be declared as a subtype of the global type `proc`: subtypes cannot be subtypes of other subtypes. When CFL initializes a system with declared subtypes, it forces the system to have a minimum of processes that corresponds to the number of declared subtypes. This is to make sure that there is at least one process of each subtype. For example, if CFL was manually initialized with three processes on the example in Figure 5.3, the first process would have subtype `subtype1`, the second process would be `subtype2` and the third process would have a subtype chosen at random. It is important to note that when subtypes are declared, CFL by default will start assigning them to processes. All subtyped processes are still type `proc`, so if a transition does not force a subtype in its parameters, then that transition can be applied to any process.

5.3.2 Initial States and Unsafe States

Thread primitives do not have any impact on the initial state or the unsafe state. By default, all locking mechanisms (locks, reentrant locks, semaphores,

conditions) start off unlocked with no owner. Currently, this cannot be changed in the initial state. The **only** thing that can be declared in the initial state is the amount of resources associated with a semaphore. Unsafe states also remain untouched by our primitives. We plan on letting unsafe states describe the states of locking mechanisms in the future.

5.3.3 Locks and Reentrant Locks

The first primitives we introduce into Cubicle are (blocking) locks and reentrant locks, which are declared with the types `lock` and `rlock`, respectively. We allow variables and arrays to have lock types. An example is given in Figure 5.4.

```
var A : lock
var B : rlock
array C[proc] : lock
array D[proc] : rlock
```

Figure 5.4: Cubicle Lock Declarations

Here we have two locks, `A` and `C`, and two reentrant locks `B` and `D`. These now built-in types come with two actions: `acquire` and `release`. We give an example in Figure 5.5.

```
transition t1 ([i])
{ acquire(A, i) }

transition t3 ([i] j)
{ acquire(B[j], i) }

transition t4 ([i])
{ release(A,i) }

transition t5 ([i] j)
{ release(B[j],i) }
```

Figure 5.5: Cubicle Acquiring and Releasing Locks

Internally, the only difference between a `lock` and a `rlock` is that `rlock` keeps a counter to how many times it has been locked by a thread. CFL keeps a waiting queue of suspended processes per lock (rlock). The only restriction is that a process cannot try to release a lock that does not belong to it. Doing so will provoke an error.

We show how Cubicle (CFL) treats locks with a simpler example than the ones used to show the syntax. This example is given in Figure 5.6

```
var X : lock
var Y : rlock

(*init omitted*)

transition t1([i])
{ acquire(X,i) }

transition t2([i])
{ acquire(Y,i) }

transition t3([i])
{ release(X,i) }

transition t4([i])
{ release(Y,i) }
```

Figure 5.6: Cubicle Acquiring and Releasing Locks

If we have three threads `#1` , #2, and #3, then CFL will initialize the following environment:

```
#1 :  active
#2 :  active
#3 :  active
X : unlocked
Y : unlocked
X queue:  {}
Y queue:  {}
```

We can now have the following sequence of actions:

(Step 1) `#1` takes transition `t1`

```
#1 :  active
#2 :  active
#3 :  active
X : locked by #1
Y : unlocked
X queue:  {}
Y queue:  {}
```

(Step 2) #2 takes transition t2

```
#1 :  active
#2 :  active
#3 :  active
X : locked by #1
Y : locked 1 time(s) by #2
X queue:  {}
Y queue:  {}
```

(Step 3) #2 takes transition t2 again

```
#1 :  active
#2 :  active
#3 :  active
X : locked by #1
Y : locked 2 time(s) by #2
X queue:  {}
Y queue:  {}
```

(Step 4) #3 takes transition t1

```
#1 :  active
#2 :  active
#3 :  suspended
X : locked by #1
Y : locked 2 time(s) by #2
X queue:  {}
Y queue:  { #3 }
```

(Step 5) #1 takes transition t3. This automatically passes the lock to the suspended, waiting #3

```
#1 :  active
#2 :  active
```

```
#3 :  active
X : locked by #3
Y : locked 2 time(s) by #2
X queue:  {}
Y queue:  {}
```

(Step 6) #2 takes transition t4 twice. This fully liberates Y, which was locked twice by #2

```
#1 :  active
#2 :  active
#3 :  active
X : locked by #3
Y : unlocked
X queue:  {}
Y queue:  { }
```

5.3.4 Conditions

Declaring a condition is the same as declaring a lock, except the type is called `condition`

```
var C : condition
```

Conditions are like locks that threads can release and acquire. This process has the same syntax as before:

```
acquire(C,i)
release(C,i)
```

However, conditions also contain a `wait` operation as well as `notify` and `notify_all` .

Wait When a thread acquires a condition, it can either continue its execution if the condition is true, or wait for the condition to become true. In Cubicle, the conditions to be satisfied are included in a transition's requirements and not in the acquire primitive. For example, in Python, we could have a condition `c` and the associated `wait_for`

```
with c:
 c.wait_for(lambda : x > 0)
 x -= 1
```

which states that if a certain variable `x` is greater than zero, then the thread can continue and decrement `x`, else it has to wait for that condition to be true.

In Cubicle with conditions this translates into two transitions: one where the condition is valid and one where it is not. The equivalent to the Python code in Cubicle is given in Figure 5.7.

```
transition wait_ok([i])
requires { X > 0 }
{ X := X - 1 }

transition wait_not_ok([i])
requires { X <= 0 }
{ wait(C,i) }
```

Figure 5.7: Cubicle Equivalent of Python Wait

Notify and Notify_all When threads attempt to acquire conditions that are locked by another thread **or** they reach a `wait` that they cannot pass, they are suspended and added to the corresponding condition's wait queue. The operations `notify` and `notify_all` exist so that threads that hold the condition lock can wake up suspended threads. The operation `notify` wakes up a random process from the waiting queue, and the operation `notify_all` wakes up all processes in the queue. The syntax is as straightforward as the other operations:

```
notify(C,i)
notify_all(C,i)
```

Note that process `i` has to be the owner of the condition lock `C` to be able to notify others.

5.3.5 Semaphores

Semaphores are counters with an implicit wait-queue. A semaphore is initialized to an integer greater than or equal to zero and every time a thread acquires the semaphore, the counter is decremented, whereas every time a semaphore is released the counter is incremented. If a thread tries to acquire a semaphore whose counter is equal to zero, then that thread is suspended. When a semaphore is released, if there are any suspended waiting threads, they are awakened. Figure 5.8 gives an example of semaphores in Cubicle. The `acquire` and `release` syntax is the same as the for locks and conditions. In fact both semaphores and conditions are built on top of the basic lock structure.

```
var S : semaphore

init() { S = 1 }

transition t1 ([i])
{ acquire(S,i) }

transition t1 ([i])
{ release(S,i) }
```

Figure 5.8: Cubicle Semaphores

5.3.6 Restrictions

The biggest restriction in implementing threads in Cubicle comes from us wanting to respect Cubicle's semantics. In Chapter 2, we mentioned that the actions in Cubicle's transitions are *atomic* and there are no sequences. This does not translate well to locking mechanisms, since if we write

```
release(L,i)
acquire(L,j)
```

we want `i` to release lock `L` before `j` tries to acquire it. However in Cubicle, there is no order to actions, so we have no guarantees here. This is why we impose that there only be **one** thread-based primitive per transition. While this does complicate models a bit, it respects Cubicle's semantics. In that vein, transitions that contain thread-based primitives **must** contain an actor process. Otherwise Cubicle (CFL) rejects the transition.

5.4 Examples

In order to test our extension and CFL, we model several examples of concurrent algorithms.

5.4.1 Producer-Consumer

We take the example of a Producer-Consumer from the book *Principles of Concurrency Programming* by M. Ben-Ari [69]. This example is what originally led us to adding subtyping to processes for CFL, because the algorithm kept deadlocking when it should not. The algorithm can be found in Fig. 4.8 on page 59 in the book. We give it in Algorithm 10.

Algorithm 10: Producer-Consumer Algorithm

```
   Var: n: semaphore;
   Var: s: binary semaphore;
1  Procedure producer
2      repeat
3          produce;
4          wait(s);
5          append;
6          signal(s);
7          signal(n);
8      until forever;
9  Procedure consumer
10     repeat
11         wait(n);
12         wait(s);
13         take;
14         signal(s);
15         consume;
16     until forever;
17 Begin (main program);
18 n := 0;
19 s := 1;
20 cobegin;
21 producer, consumer;
22 coend;
```

This Producer-Consumer example is linked to a buffer (operations *append*, *take* lines 5, 13). The **wait** operations stand for acquiring the semaphore and **signal** for releasing it. In this algorithm there are two semaphores, n and s. The semaphore n stands for the data produced and available for consumption by the consumer. The semaphore s guarantees mutual exclusion - the producer and consumer will not be accessing the buffer at the same time. When the producer produces something (line 3), it acquires the mutual exclusion semaphore s. It adds what it produced to the buffer (line 5) and releases s. Then it releases n, signaling that it is possible to consume something. The consumer waits for n to be greater than zero. If so, it acquires n (line 11), and then acquires s (line 12), i.e. access to the buffer. It takes the data from the buffer (line 13) and releases s. This process repeats indefinitely.

For simplicity, we omit the buffer element as well as the *produce* and *consume* actions (lines 3 and 15) in our Cubicle model described further, focusing

only on the sequence of waits and signals. We give the Cubicle model in Figure 5.9.

Our Cubicle model contains eight transitions, one for each `wait` and `signal` appearing in Algorithm 10. The transitions `wait_produce`, `wait_put_loop`, and `enter_put` correspond to lines 4, 6, and 7 in Algorithm 10, respectively. The transitions `wait_consume`, `wait_get_loop`, and `enter_get` correspond to lines 11, 12, and 14 in Algorithm 10, respectively. The transitions `run_produce` and `run_consume` start each thread off depending on its subtype (`producer` or `consumer`). Without these two transitions (and the subtype) the algorithm deadlocks incorrectly for trivial reasons. The array `PC` is a program counter keeping track of threads' actions.

If we run CFL with two threads on our example, it runs smoothly without any issues. The book states that if you inverse the two waits on lines 11 and 12 n Algorithm 10, the whole system deadlocks. We want to test CFL to see if it detects the deadlock when we switch the two around. We modify the `wait_consume` and `wait_get_loop` transitions in Fig. 5.9 by switching the values in the **acquire** construct. We now run CFL again for two threads. This time it signals that it has reached a deadlock. If we look at CFL's log file, we see the generated deadlock trace:

$$\texttt{Init} \rightarrow \texttt{run_consume(\#1)} \rightarrow \texttt{wait_consume(\#1)} \rightarrow$$
$$\texttt{run_produce(\#2)} \rightarrow \texttt{wait_produce(\#2)} \rightarrow \texttt{wait_get_loop(\#1)}$$

This trace is short enough to manually understand what happened. Thread `#1`, the consumer, grabbed s before the producer (`#1`), so when the producer started its run, it was blocked. Then, #2 grabbed n, a semaphore that was set to 0, meaning it blocked itself and consequently the whole system. But sometimes, as we will see with our next example, error traces are not that easy to interpret.

5.4.2 Dining Philosophers

The Dining Philosophers [70] is a classic synchronization problem in concurrent programming. Five philosophers are sitting around a table with an eating utensil between each pair of philosophers. In order to eat, a philosopher has to be in possession of the utensil to his right and to his left. The problem is managing access to utensils without blocking every philosopher. In the spirit of Cubicle's parameterized nature, we slightly modify the example, and instead of five philosophers and five utensils, we have *N* philosophers and *N* utensils. We also forgo the distinction between left and right and consider that utensils are all in one pile. We encode this with one semaphore originally set to *N*-1, a counter `Utensil` set to *N*, and a condition that allows philosophers to move if `Utensil` is greater than zero (otherwise they have to wait).

```
type get < proc
type put < proc

type t = None
        | Produce_Wait | Append_1 | Append_2
        | Consume_Wait | Consume_Wait_S | Take


var S : semaphore (*binary semaphore*)
var N : semaphore
array PC[proc] : t

init(i) {PC[i] = None && S = 1 && N = 0}

(*---- Run ----*)

transition run_produce(i:put)
requires { PC[i] = None }
{ PC[i] := Produce_Wait }

transition run_consume(i:get)
requires { PC[i] = None }
{ PC[i] := Consume_Wait }

(*---- Put ----*)

transition wait_produce([i])
requires { PC[i] = Produce_Wait }
{ PC[i] := Append_1; acquire(S,i) }

transition wait_put_loop([i])
requires { PC[i] = Append_1}
{ PC[i] := Append_2; release(S,i); }

transition enter_put([i])
requires { PC[i] = Append_2 }
{ PC[i] := None; release(N,i); }

(*---- Get ----*)

transition wait_consume([i])
requires { PC[i] = Consume_Wait }
{ PC[i] := Consume_Wait_S ;
  acquire(N,i); }

transition wait_get_loop([i])
requires { PC[i] = Consume_Wait_S }
{ PC[i] := Take;
  acquire(S,i);}

transition enter_get([i])
requires { PC[i] = Take }
{ PC[i] := None;
  release(S,i) }
```

Figure 5.9: Producer-Consumer in Cubicle

The interesting aspect of modeling the Dining Philosophers is that we can model two different versions, one with semaphores and one without. The difference is that the version without semaphores should deadlock. It is a good way to check that CFL is capable of finding thread-based deadlocks.

No Semaphores We give the Python code for our version of the Dining Philosophers in Figure 5.10. This version has no semaphores. It has one condition, and 5 philosophers. Each philosopher tries to acquire an eating utensil (the Get comments on lines 11, 19) and then when the philosopher has acquired two eating utensils, he can release them and notify everyone else of that fact. The Python code omits several technical details to make the code easier to read.

We give the Cubicle equivalent in Figure 5.11

The only new feature in the Cubicle code that we have not seen yet is `SYS_PROCS`. This is a new built-in constant in CFL that automatically sets the number of processes for the system as the number which was passed to the command line. This avoids having to hard-code values in Cubicle models. Looking at the Cubicle code, or the Python code for that matter, it is not obvious that this can lead to a deadlock. If we run CFL on it, it quickly spits out a deadlocking trace (in 0.067 seconds). CFL will note all of the deadlocking paths that it encountered in a separately generated file. There is a possibility of paths repeating, since CFL might cover the same areas multiple times. Currently we do not filter our repeat paths and CFL simply writes everything to the file. One of the paths generated by CFL in our example is the following:

```
Init → start(#5)→ get1_lock(#5) → get1_continue(#5) →
get1_release(#5)→ start(#4) → get1_lock(#4) → start(#2) →
get1_lock(#2) → start(#1)→ get1_lock(#1)→ start(#3) →
get1_lock(#3)→ get1_continue(#4) → get2_lock(#5) →
get1_release(#4) → get1_continue(#2) → get2_lock(#4)→
get1_release(#2) → get1_continue(#1) → get2_lock(#2) →
get1_release(#1) → get2_lock(#1) → get1_continue(#3) →
get1_release(#3)→ get2_wait(#2) → get2_lock(#3) →
get2_wait(#4) → get2_wait(#5) → get2_wait(#4) → get2_wait(#1)
```

Unlike the trace in Section 5.4.1, this path is hard to interpret. There are 31 steps in total and 5 implicated processes, so trying to re-execute this manually would most likely lead to errors. We discuss in Chapter 6 our solution to interpreting CFL's long error traces.

For the sake of comparison, we can now try the version with semaphores. This version is given in Figure 5.12. The only difference with the previous version is the addition of

```
var S : semaphore
```

```
n = 5
x = n
c = threading.Condition()

def run(i):
for k in range(3):
#s.acquire()

# Get1_lock
with c:
# Get1:
c.wait_for(lambda : x > 0)
x -= 1
# ReleaseGet1
print(i,':get1')

# Get2_lock
with c:
# Get2
c.wait_for(lambda : x > 0)
x -= 1
# Release_Get2
print(i,':get2')
with c:
x += 2
c.notify_all()
print(i,':release')

P = [ threading.Thread(target=run, args=(i,)) for
i in range(n)]
for i in range(n):
P[i].start()

for i in range(n):
P[i].join()

```

Figure 5.10: Dining Philosophers in Python (no semaphore)

which is set in the initial state to `SYS_PROCS - 1`, i.e. N-1 as mentioned earlier. We add the extra step of acquiring the semaphore in the `start` transition. The semaphore is released in the newly added transition `release_sem`. Remember that we cannot add our `release` for the semaphore `S` to transition `done`, because transition `done` already contains a `release` for condition `C`. Now, no matter how long we run CFL for, it does not show any deadlocking traces.

```
type s = Idle | Done
         | Get1 | Get1_lock | ReleaseGet1
         | Get2 | Get2_lock | ReleaseGet2

array P[proc] : s

var X : int
var C : condition

init(i) { P[i] = Idle && X = SYS_PROCS }

transition start([i])
requires { P[i] = Idle }
{ P[i] := Get1_lock }

transition get1_lock([i])
requires {P[i] = Get1_lock }
{ acquire(C,i);
  P[i] := Get1 }

transition get1_continue([i])
requires { P[i] = Get1 && X > 0 }
{ P[i] := ReleaseGet1;
  X := X - 1 }

transition get1_wait([i])
requires { P[i] = Get1 && X = 0 }
{ wait(C, i) }

transition get1_release([i])
requires { P[i] = ReleaseGet1 }
{ release(C,i);
  P[i] := Get2_lock }

transition get2_lock([i])
requires {P[i] = Get2_lock }
{ acquire(C,i);
  P[i] := Get2 }

transition get2_continue([i])
requires { P[i] = Get2 && X > 0 }
{ P[i] := ReleaseGet2;
  X := X - 1 }

transition get2_wait([i])
requires { P[i] = Get2 && X = 0 }
{ wait(C, i) }

transition get2_release([i])
requires { P[i] = ReleaseGet2 }
{ notify_all(C,i);
  X := X + 2;
  P[i] := Done }

transition done([i])
requires { P[i] = Done }
{ release(C,i);
  P[i] := Idle }
```

Figure 5.11: Dining Philosophers in Cubicle (no semaphores)

```
type s = Idle | Done | ReleaseSem | Get1 | Get1_lock | ReleaseGet1
         | Get2 | Get2_lock | ReleaseGet2

array P[proc] : s
var X : int
var C : condition
var S : semaphore

init(i) { P[i] = Idle && X = SYS_PROCS && S = SYS_PROCS - 1 }

transition start([i])
requires { P[i] = Idle }
{ P[i] := Get1_lock;
  acquire(S,i);}

transition get1_lock([i])
requires {P[i] = Get1_lock }
{ acquire(C,i);
  P[i] := Get1 }

transition get1_continue([i])
requires { P[i] = Get1 && X > 0 }
{ P[i] := ReleaseGet1;
  X := X - 1 }

transition get1_wait([i])
requires { P[i] = Get1 && X = 0 }
{ wait(C, i) }

transition get1_release([i])
requires { P[i] = ReleaseGet1 }
{ release(C,i);
  P[i] := Get2_lock }

transition get2_lock([i])
requires {P[i] = Get2_lock }
{ acquire(C,i); P[i] := Get2 }

transition get2_continue([i])
requires { P[i] = Get2 && X > 0 }
{ P[i] := ReleaseGet2; X := X - 1 }

transition get2_wait([i])
requires { P[i] = Get2 && X = 0 }
{ wait(C, i) }

transition get2_release([i])
requires { P[i] = ReleaseGet2 }
{ notify_all(C,i);
  X := X + 2; P[i] := Done }

transition done([i])
requires { P[i] = Done }
{ release(C,i); P[i] := ReleaseSem }

transition release_sem([i])
requires {P[i] = ReleaseSem }
{ release(S,i); P[i] := Idle }
```

Figure 5.12: Dining Philosophers in Cubicle (with semaphores)

5.4.3 Other Examples

We have coded other classic examples in our extended CFL-powered Cubicle.

- The Sleeping Barber [31]: the Sleeping Barber is a synchronization problem described by Dijkstra. In this problem there is a barbershop with a barber. If there are no customers, then the barber stays asleep. If a customer walks in and the barber is sleeping, they wake up the barber. If a customer walks in and the barber is working on someone, then the customer tries to sit in the waiting room. If no seats are available, the customer leaves. As soon as the barber has no more customers left to treat, he falls back asleep. We encode the problem with three semaphores (one of which is a binary semaphore).

- The Cigarette Smokers Problem [71]: the Cigarette Smokers problem consists of four threads: three threads are classified as smokers and one is the non-smoker. Each of the smokers has an infinite supply of either matches, tobacco, or paper, and they need all three to form a cigarette to smoke. The non-smoking agent has access to all three supplies. When a smoker wants to smoke, he signals to the non-smoker that he would like to smoke and the supplies needed. The non-smoker than produces those ingredients, allowing the smoker to create his cigarette. We encode the problem as given in [72] with four semaphores and a lock(mutex).

- Sense-reversing Barrier: in the sense-reversing barrier, there are two global variables- Sense and Count. The Count variable is initialized with the number of threads, and the Sense barrier is used to indicate which barrier threads are facing. Each time a thread reaches a barrier, it decrements Count. When Count is at zero, the barrier opens, Count is reset, and the Sense variable switches.

Chapter 6

Interactivity, Execution, and Debugging

Contents

6.1 Interpreter Commands 99
6.2 Usage Example 102

While having the ability to model various types of systems is interesting and undoubtedly good for Cubicle, as we saw in Chapter 5, the error traces generated can be arbitrarily complex and hard to interpret. Being able to re-execute the traces step-by-step is crucial in understanding what caused them. It is also always fun to be able to play around with a system and just see what happens It is with this in mind that we introduced the topic of this chapter- an interpreter and a debugging layer for Cubicle.

6.1 Interpreter Commands

The interpreter in Cubicle can be used on any Cubicle model with the command

```
cubicle -interpreter model.cub
```

where `model.cub` is the Cubicle file. By default, the interpreter starts with three processes (or threads). It is possible to change this by using the command line option

```
-interpret-proc n,
```

where n is greater than zero. The interpreter does not currently preprocess models to force a minimum number of processes.

The interpreter is launched in the terminal and resembles Figure 6.1.

```
*******************************************************************************************************
                                               Cubicle
                                        Interpreter & Debugger
*******************************************************************************************************
Usage:
Executing transitions:
        transition <name>(<args>) : apply a transition
        transition <name>(<args>); <name>(<args>)... : apply a sequence of transitions

Other Commands:
        help : display this list
        status : show current environment
        execute : run random execution
        execute <N> <depth> : execute <depth> transitions <N> times. Looks for unsafe state
        all : show possible transitions
        random : pick a random transition and apply it
        unsafe : check if current state is unsafe
        reset : reset the environment [global system state, trace logs, backtracking info]
Debug Commands:
        flag <int> : set how often debugger remembers states for easier backtracking
        trace  : show trace
        replay : replay entire trace, waits for user OK after each step
        backtrack <int> : backtrack environment to Step <int>
        rerun <int> <int> : rerun trace between two steps, waits for user OK after each step
        why <transition call> : explain which values interfere with transition application
        dhelp : show debug help
>
```

Figure 6.1: Interpreter Welcome Screen

The interpreter's welcome screen lists a few basic commands, as well as usage instructions. The interpreter has a global environment that corresponds to a state in the system. A user can interact with the system in several ways:

> transition <name> (parameters)
lets a user execute a transition. The transitions requirements have to be valid in the current environment

> transition <name> (parameters); <name> (parameters); ...
lets a user execute a sequence of transitions. When executing a sequence of transitions, if one of the transitions fails to execute, all of the transitions are rolled back, and there is no effect on the environment

> all
lists all possible transitions in current environment

> random
picks a random, valid transition and apply it

> execute
starting from the current environment, runs a random execution, choosing transitions at random. The execution will stop when it encounters a deadlock or an unsafe state

> `execute_<strat>`
starting from the current environment, executes the system using one of the following CF exploration techniques:

- `proc`: process sequences
- `weight`: weighted decision
- `max`: maximizing randomness
- `bfs`: limited BFS

CFL's last technique (Unused Exit), is not implemented, since it has no meaning in the context of the interpreter.

> `status`
shows the current environment

`reset`
resets the interpreter's environment to the initial state

• `unsafe`
checks whether the current environment is unsafe

When the interpreter executes a model, it annotates each visited state as a *Step*. The initial state is Step 0. It also makes note of how many transitions were possible before a Step, and how many are possible after. This information is used in the debugging layer of the interpreter. This layer consists of the following commands:

> `trace`
shows the execution trace

> `why <transition_name (parameters)>`
explains why a certain transition is not possible in the current state

> `flag <int>`
tells the interpreter to remember states every `<int>` steps By default `flag` is set to one, so every state is remembered

> `replay`
replays the entire trace of memorized steps, highlighting the changes in the environment between each step

> `rerun <int_start> <int_stop>`
rerun (i.e. potentially recalculate) the trace between steps `<int_start>` and `<int_stop>`. The step <int_start> needs to be a memorized state. This step is useful if you want to replay details of steps whose states were not originally memorized

> `backtrack <int>`
sets the interpreter environment to the state corresponding to step `<int>`

6.2 Usage Example

We will be use the deadlocking version of our producer-consumer algorithm from Chapter 5 to illustrate the features we described in the previous section.

We start the interpreter with two processes:

```
cubicle -interpreter -interpret-proc 2 ben-ari.cub
```

We then immediately type `status` to see what the environment looks like, shown in Figure 6.2.

```
> status
Final Environment
==========================================================
#1 : type: put; status: active
#2 : type: get; status: active
S : 1
N : 0
PC[#1] : None
PC[#2] : None
----------------------------------------------------------
Lock Queues:
----------------------------------------------------------
Condition wait pools:
----------------------------------------------------------
Semaphore wait lists:
S : { }
N : { }
----------------------------------------------------------
>
```

Figure 6.2: Interpreter Producer-Consumer Initial Status Screen

The status screen first lists the processes (or threads) in the system, as well as their subtype, and their status. The subtype and status are direct consequences of the thread implementation from Chapter 5. If the model contains no subtype declarations, then the type will always be `proc`. Then the screen shows us `S` and `N`, which were the two semaphores, as well as an array `PC` to indicate where in the model each thread is. Next it shows three queues: `Lock`, `Condition`, and `Semaphore`. Even if the model contains none of these, the status screen will still show them, they will just have nothing in them.

Now that we have started the interpreter and visualized the initial state, we can start playing with the system.

Figure 6.3 shows the effect of various commands:

- `all` listed the initially possible transitions `run_consume(#2)`, `run_produce(#1)`
- We try to execute a sequence of transitions `run_consume(#2)` and `run_consume(#1)`. The interpreter rejects this sequence because `#1` is not a consumer.
- We ask the interpreter to explain why `run_produce(#1)` is impossible. It replies the transition is not blocked and is possible.
- We ask why `wait_consume(#2)` is blocked, to which the interpreter replies that this is due to `PC[#2]` having the wrong value.

```
> all
Possible transitions:
transition run_consume(#2)
transition run_produce(#1)
> transition run_consume(#2); run_consume(#1)
Process #1 should be of get subtype
> why run_produce(#1)
Transition run_produce(#1) NOT blocked
> why wait_consume(#2)
Transition wait_consume(#2) blocked because following reqs are false:
        PC[#2] = Consume_Wait
>
```

Figure 6.3: Interpreter Helper Functions

We then let the interpreter run on our buggy model and get the result shown in Figure 6.4

```
> execute
WARNING: Deadlock reached
Total entries: 6
Total visited: 6
State seen most often: 39696945750868566 [1 time(s)]
>
```

Figure 6.4: Interpreter Results for Buggy Producer-Consumer

The interpreter warns us about the deadlock and lists:

- Total entries: how many *unique* states were visited
- Total visited: how many states were visited in total, including repeats
- State seen most often: states in the interpreter, and in CFL in general, are hashed for easier comparison. This field shows the hash of the state and how many times it was seen.

In the case of our buggy example, the interpreter visited only six states, and they were all unique. If multiple states have the same visit count, and this visit count is greater than every other visit count, the interpreter chooses one at random. We ask the interpreter for the trace, which in our case corresponds to Figure 6.5.

```
> trace
Applied transitions:
---
  pre: possible transitions before
  post: possible transitions after
  MANUAL: transition applied manually, pre/post not calculated
  **Step <int>: resulting env stored and accessible
---
**Step 1[pre: 2, post: 2]: transition run_produce(#1)
**Step 2[pre: 2, post: 2]: transition run_consume(#2)
**Step 3[pre: 2, post: 2]: transition wait_consume(#2)
**Step 4[pre: 2, post: 1]: transition wait_get_loop(#2)
**Step 5[pre: 1, post: 0]: transition wait_produce(#1)
>
```

Figure 6.5: Interpreter Trace

As we can see in Figure 6.5, every step is highlighted in green. This is because we never set `flag`, so it kept the default value of one. Had we changed `flag`, and set it to two for example, we would have gotten a result like Figure 6.6

```
> trace
Applied transitions:
---
  pre: possible transitions before
  post: possible transitions after
  MANUAL: transition applied manually, pre/post not calculated
  **Step <int>: resulting env stored and accessible
---
Step 1[pre: 2, post: 2]: transition run_consume(#2)
**Step 2[pre: 2, post: 2]: transition wait_consume(#2)
Step 3[pre: 2, post: 1]: transition wait_get_loop(#2)
**Step 4[pre: 1, post: 1]: transition run_produce(#1)
Step 5[pre: 1, post: 0]: transition wait_produce(#1)
>
```

Figure 6.6: Alternative Trace

We can now replay the trace. It is easy in our case because the interpreter kept track of every step. The interpreter will print each transition taken and the resulting environment, highlighting what changed between each step, like in Figure 6.7.

```
Starting from Step 1, post transition run_consume(#2)
  #1 : process active
  #2 : process active
  S : 1
  N : 0
  PC[#1] : None
  PC[#2] : Consume_Wait
  Lock Queues:
  Condition wait pools:
  Semaphore wait lists:
   S : {  }
   N : {  }
------------------------------------------------------

Press enter to continue each time

After Step 2, transition wait_consume(#2)
  #1 : process active
  #2 : process active
  S : 0
  N : 0
  PC[#1] : None
  PC[#2] : Consume_Wait_S
  Lock Queues:
  Condition wait pools:
  Semaphore wait lists:
   S : {  }
   N : {  }
------------------------------------------------------
```

Figure 6.7: Interpreter Replay

We introduce a (reachable) unsafe state to our model:

```
unsafe (i) { PC[i] = Consume_Wait_S }
```

We reset the interpreter and ask it to execute again. This time it reaches an unsafe state before reaching the deadlock, as shown in Figure 6.8

```
> execute
WARNING: Unsafe state reached. Do you wish to continue? (y/n)
```

Figure 6.8: Reaching an Unsafe State

When the interpreter reaches an unsafe state during execution, it asks the user whether they want to continue executing or stop. Unsafe states are

not checked automatically when transitions are executed manually, as in Figure 6.9. We specifically need to ask the interpreter if the current environment is unsafe. This could be changed in future versions of the interpreter.

```
> all
Possible transitions:
transition run_consume(#2)
transition run_produce(#1)
> transition run_consume(#2)
> all
Possible transitions:
transition wait_consume(#2)
transition run_produce(#1)
> transition wait_consume(#2)
> unsafe
WARNING: Current state is unsafe
>
```

Figure 6.9: Alternative Trace

Chapter 7

Test Case: Tenderbake

Contents

7.1 Blockchains & Consensus 107
7.2 Tenderbake 113
7.3 Modeling & Testing a Simple Tenderbake . . 114

The work in this chapter is based on our work published in [73], where we formalize Tenderbake using TLA$^+$.

Tenderbake [21] is a consensus algorithm designed by Nomadic Labs for the Tezos blockchain [74]. Tenderbake is inspired by Tendermint [75], which in turn is an adaptation of PBFT [76] (described later in section 7.1). We first give brief overviews of blockchains and consensus algorithms, necessary to understand Tenderbake. We do not delve into all aspects of consensus and blockchains, but focus only on those necessary to understand Tenderbake. We then introduce the Tenderbake algorithm itself.We show how our new extensions allow us to model and test Tenderbake *incrementally* and how this is crucial for Cubicle.

7.1 Blockchains & Consensus

A blockchain is, simply put, a book of records. In this analogy, each page in the book contains a list of transactions, financial or otherwise, and the pages are numbered, so you know if a page has been ripped out or added. Page numbers cannot be modified, and neither can the contents of a page. In the context of blockchains, this is called a **ledger**. At the same time, there isn't one single instance of this ledger. The ledger is distributed among the people appearing in it - anyone who interacts with it has their own personal copy. This means that when something is added to one ledger, it has to be added to

every single copy. Because of these two aspects, blockchains are sometimes referred to by their more direct name – *Distributed Ledger Technology*.

Ledger In lieu of pages, a blockchain has blocks. Every block contains a pointer to its predecessor, resulting in the chain aspect. And similarly to how it is not possible to rip a page out of a paper ledger (or add a page) due to page numbers being a dead giveaway that something has been changed, the link between blocks and their predecessors ensures that a block cannot be added or removed without significant computational power (i.e. the whole blockchain needing to be rebuilt). This results in one of the biggest features of blockchains in general - any existing information stored within is immutable. It also results in traceability - anything stored in a blockchain can be traced back in time to when it was added.

Typically a page in a book of records would contain the page number and the date the page was added, along with a list of its transactions. A block in a blockchain contains the same information, but organized differently.

Internally, a block has a header and a body of contents. The body of contents is a list of the transactions that are stored within the block. Headers, however, may vary from blockchain to blockchain, e.g. Ethereum [77] block headers differ from Bitcoin's [78], although the general contents can be boiled down to what is shown in Figure 7.1.

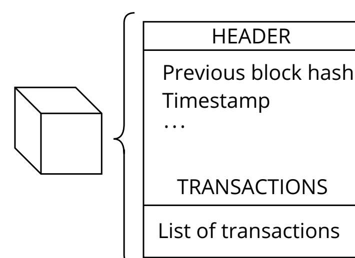

Figure 7.1: General Structure

Block headers contain the hash of their parent block, serving as the pointer mentioned earlier. Storing the hash of the block is what makes it impossible to alter previous blocks: any change would modify the hash, breaking the link. The header also stores a timestamp to indicate when a block was created, similarly to how a page in a record book would have a date.

Distributed As opposed to a centralized system, where there is only one central controlling party, the blockchain is a distributed system, meaning that the blockchain is replicated and each party has their own copy, and there is no controlling entity. This makes blockchains a Peer-to-Peer (P2P) network, as seen in Fig. 7.2. Here, each peer in the network has their own local copy of the blockchain.

In a centralized system, such as a bank, any operations are verified/executed by the bank itself. In the distributed version, where the peers each have a local copy of the bank and the information stored within, any operations have to be verified by the members of the network, and they have to agree

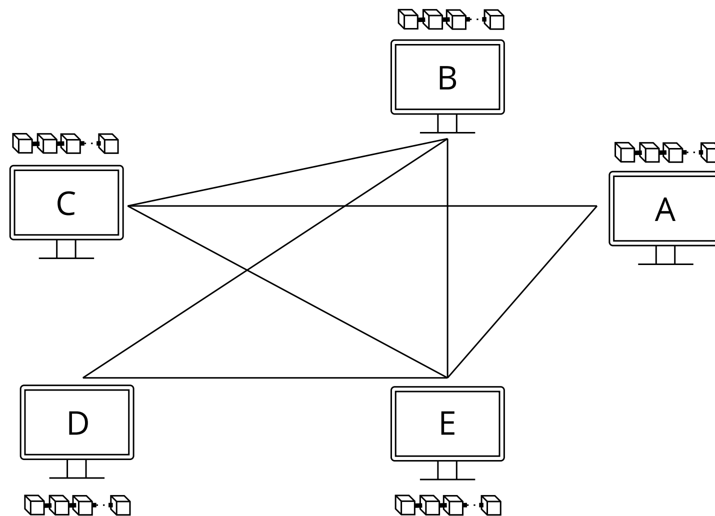

Figure 7.2: P2P Network

on the validity of said operations. Applied to the blockchain, all of this provides fault-tolerance. Since the blockchain is replicated, a peer failing doesn't lead to the loss of any information. At the same time, the P2P network makes the blockchain more resistant to network issues, such as server downtime.

Assuming that there is a bookkeeper, adding a page to a book of records when there is one centralized book, is easy. Even if multiple people have their own local copies, in a centralized system it would still be straightforward - the main book is modified, and everyone else updates their local copy based off of that. Multiple people having their own versions of the book with no controlling party is much harder - now when something is added to one book, the information needs to be passed on and added to every other book. Assuming that there are no problems passing on information, the only remaining issue is: what happens if person A doesn't agree with person B on which page to add?
The blockchain is replicated and distributed among multiple peers. The issue lies in that, while everyone maintains their own version, the versions need to match. The blockchain would be useless if everyone paraded around with their own vision of the world. Matching versions is harder when you include the P2P network. This is because not everyone on the network sees the same transactions, so when person A proposes a block with transaction X, it's entirely plausible that person B didn't even know transaction X existed. This means that the blockchain needs to have a mechanism that allows peers to agree on what they're adding to the chain. This is called **consensus**.

Blockchains need a consensus mechanism to allow peers to decide which blocks to add. If peers cannot agree, the network runs the risk of having multiple versions of the blockchain. This can lead to several problems, for example

double-spending, where a peer spends the same money twice, because different versions of the blockchain indicate that the money has not yet been spent.

Consensus existed before blockchains and is a key part of distributed systems, allowing processes to reach mutual decisions, for example when choosing a value. In general, consensus is reached when the following conditions are satisfied:

- **Agreement** All correct processes agree on the same value
- **Validity** The agreed upon value was proposed by a correct process
- **Termination** Each correct process eventually decides on a value

The first two conditions are *safety* properties, while the third is a *liveness* property.

Ideal systems vs. Blockchains In theory, consensus seems simple enough - every process needs to vote, and the majority wins. Issues with consensus start to arise when the system is no longer ideal, for various reasons:

- **Network Issues**: A network can have any number of issues. All of these lead to problems that will impact a blockchain consensus decision, namely message loss and message delay.
- **Process Failures - stopping failures**: A process can stop responding at any moment. There are no guarantees that the process will restart later on.
- **Process Failures - Byzantine**: Processes can exhibit arbitrary behaviors [79]

Blockchain-related articles often shove all process failures under the term "Byzantine", and then distinguish between it being a stopping failure or a process not following the protocol.

Asynchronous Networks vs. Synchronous Networks There are two main system models: synchronous and asynchronous. The following assumptions can be made in a synchronous system:

- Messages are bound: the maximum time a message can take to arrive is known, and this bound is global across the entire system
- Process speed is bound: the time it takes a process to do something has a lower and an upper bound

Synchronous systems, where processes move in synchronized rounds, aren't our main focus, since the blockchain exists on an asynchronous network.

The above assumptions do not apply to asynchronous networks. These networks have no bounds on anything: processes can take however long they want to complete a task, and messages can take a very long time to arrive. Consensus in its general form, as mentioned earlier, is impossible to solve in an asynchronous network [80]. The possibility of a process crashing means that the system always starts out undecided, and the arbitrary message delay, that cannot be distinguished from a process crash in an asynchronous network, means that the system can remain undecided indefinitely. This equals to the **Termination** property never being satisfied.

The widespread usage of asynchronous systems requires adapting to the impossibility result in [80], with solutions being safe and probabilistic (since Termination cannot be guaranteed).

Partially Asynchronous Systems A class that exists between the two main models is partially asynchronous systems, also referred to as *timing-based* systems [79]. In this type of system, the notion of time and bounds exists, and processes have access to information concerning time, although it might not be 100% exact.

Probabilistic vs. Immediate Finality The impossibility result in [80] means that in a fully asynchronous system, Termination cannot be guaranteed. This implies that in the case of an asynchronous blockchain network, the guarantee that a block has been added to the blockchain is only *probabilistic*. Probabilistic finality, such as in Bitcoin, means that the chances that a block is permanently in the blockchain grow with each new block added to the chain. In such a blockchain, if a block b is added, then every block $b+1$ adds to the probability that block b will remain in the blockchain, and the blockchain won't change to a version where b is no longer part of the overall chain. In Bitcoin, this translates to having to wait for six new blocks to be added on top of block b to be sure that nothing will be reverted. This is however a general assumption, and technically nothing stops the blockchain from reverting even after 6 blocks.

Immediate finality tries to guarantee that once a block has been added to a blockchain, you're sure the block will remain in the blockchain. Immediate finality is usually in n blocks, meaning that if you add a block b to the chain, after n blocks on top of it, you can be sure that no matter what, b will remain in the blockchain.

The algorithm that we're interested in, Tenderbake, is from this family of immediate finality consensus algorithms, and is based off of **PBFT** - Practical

Byzantine Fault Tolerance [76].

PBFT Byzantine fault tolerance (BFT) is what allows consensus algorithms to account for nodes that don't follow the protocol and is derived from the Byzantine Generals' Problem [81]. PBFT [76] is an optimization of BFT. In PBFT, there is one leader node to propose an action, and the rest are secondary nodes and will vote on the proposed action. If the leader node fails, any of the secondary nodes can be promoted to replace it. The idea is that all of the honest, protocol-following nodes will achieve consensus together by voting on a result. PBFT holds up as long as the number of Byzantine nodes (in the sense of not following the protocol for whatever reason) is less than $\frac{1}{3}$ of the total number of nodes. PBFT implements two voting rounds: nodes will not cast any final votes in the second round of voting before seeing the majority (called a quorum) decision from the first round of voting. Two voting rounds also protects the network from issues following the leader node changing (i.e. secondary nodes seeing different proposals). A correct execution of PBFT can be summarized as:

- Phase *Request*: a client sends a request to the leader node
- Phase *Pre-prepare*: the leader broadcasts the request to the secondary nodes
- Phase *Prepare*: when the secondary nodes receive the request, they verify its validity. If it is valid, they broadcast Prepare messages to all other secondary nodes. If it is not valid, they do nothing. This is the first round of voting.
- Phase *Commit*: upon receiving Prepare messages from $\frac{2}{3}$ of the secondary nodes, secondary nodes broadcast Commit messages. This is the second round of voting.
- Phase *Result*: the client sees the result

A correct execution of PBFT can be seen in Fig. 7.3 below. The leader, Node 1, receives a request from a Client. The leader then passes on this request to every other node (pre-prepare). Upon receiving that message, the other nodes check the message, and if it's valid, they vote for the first time (prepare). This vote is sent to everyone, including the leader. Note that Node 3 is faulty and does not send anything. After the nodes receive enough prepare messages, they send a second vote (commit) to everyone else. Again, Node 3 does not do anything. Finally, once every node has received enough commit messages, it sends a reply to the client. Node 3 does not send a reply, but since more than $\frac{2}{3}$ of all nodes sent a reply, the client accepts the result.

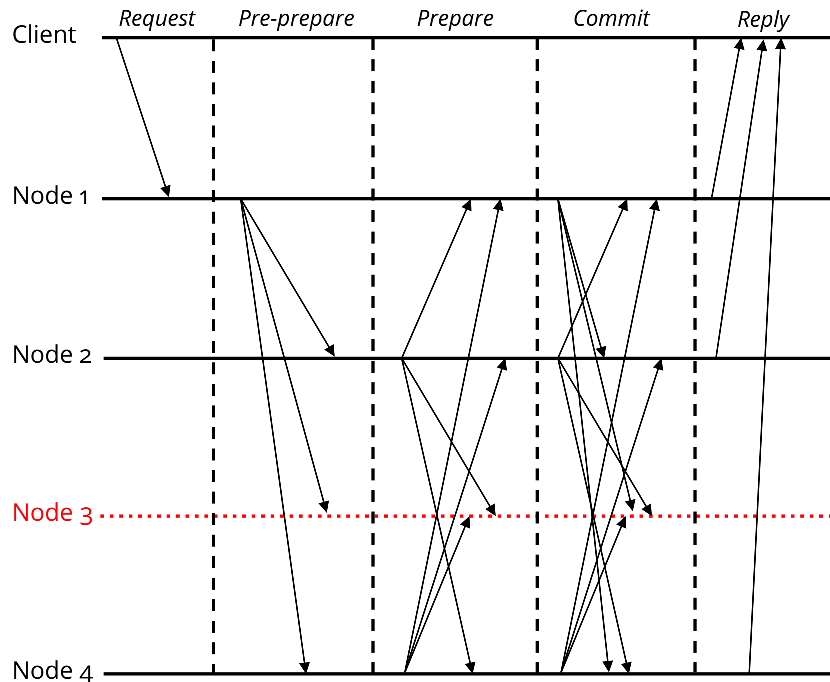

Figure 7.3: A normal execution of PBFT with one faulty node

PBFT offers immediate finality as opposed to probabilistic. As stated in [76], PBFT provides both safety and liveness (assuming a certain number of faulty nodes). However, in order to circumvent [80], PBFT sacrifices asynchronous assumptions when if comes to liveness. Time bounds on message delays are introduced: it is assumed that a message will eventually not take longer than a certain time t to arrive if the sender keeps re-transmitting it until it is received, and, if the message loss is due to network issues, the network is eventually repaired. The authors state that this is a rather weak synchrony assumption, but is likely to be true in a real-life setting. This makes the algorithm partially asynchronous.

7.2 Tenderbake

When discussing Tenderbake, we refer to participants in the algorithm as *bakers*, following Tezos's terminology.

Being a PBFT-style algorithm, Tenderbake resembles what was described in Section 7.1. Contrary to the described PBFT however, in Tenderbake, there is no client sending a request to a leader node. The leader node, or the proposing baker in this context, decides on the proposal (in this case a block) to broadcast. Tenderbake, therefore, consists of the following three phases:

- Phase 1: A (unique) baker proposes a new block (*Proposal*) to be added

to the blockchain

- Phase 2: The bakers vote for the proposal and wait for a quorum of *preendorsement*) votes
- Phase 3: If/when a quorum is obtained, bakers vote a 2nd time for the proposal and wait for a quorum of (*endorsement*) votes.

These three phases together form a round. Tenderbake is based on the concept of rounds and levels. Each block in the blockchain corresponds to a level. Tenderbake has to go through a full round in order to add a block and go up a level. The proposing baker is chosen depending on the round/level.

PBFT offers absolute finality. In Tenderbake, this feature shows up as well, as immediate finality in two blocks. A block is considered to be final after there have been two blocks added on top of it. So if a block is level l, the moment there is a block level $l + 2$, block l is considered final.

Failures and asynchrony In theory, the algorithm runs smoothly and rounds never fail: one baker proposes, the others vote, the blockchain goes up a level, a new proposer is chosen, and the cycle continues. A round in Tenderbake only lasts a certain amount of time. Bakers use their internal clocks along with block timestamps to calculate what round they're in and how long they have before a timeout. On the one hand, this circumvents the algorithm never converging- if there is nothing to stop a round, then hypothetically, it could go on forever, for example if messages are lost. Time forces bakers to move on. On the other hand, time also means that the algorithm might not run as smoothly, because any phase can now fail:

- No proposal was received (message lost, the proposing baker crashed, ...)
- No preendorsement quorum was reached (network issues)
- No endorsement quorum was reached (network issues)

If one of the three phases fails for a baker, then the whole round fails. And if a round fails, the baker has to start a new round, because the goal is to, ultimately, choose a block to add. However, one baker failing doesn't mean that another baker failed, leading to bakers all potentially having different versions of the consensus. The addition of internal clocks also leads to potential clock drift, since bakers might have slight variations of the current time.

7.3 Modeling & Testing a Simple Tenderbake

Contrary to our work in [73], we only want to focus on the actual bakers and the algorithm they follow. We do not take into account how Tenderbake is im-

pacted by Tezos's architecture, nor the notion of *time*. This is because our goal is to highlight how CFL changes our modeling process and renders it more *incremental*.

Each baker in Tenderbake runs the same automaton, given in Figure 7.4. This automaton represents the evolution of a baker's state and the actions performed by the baker in the three possible consensus phases. Bakers exist in three possible states: (i) no proposal (**NP**), (ii) collecting preendorsements (**CP**), and (iii) collecting endorsements (**CE**).

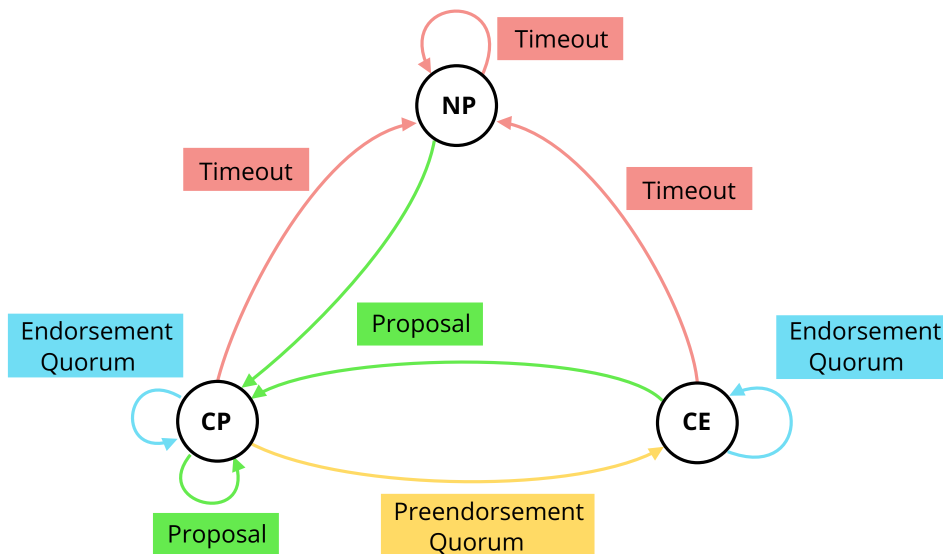

Figure 7.4: Baker automaton

The default state is **NP** - the baker has no proposals to consider/treat. When a baker receives a proposal, they automatically move to state **CP**, meaning the second phase of the algorithm - collecting preendorsements for the proposal. If the baker observes a preendorsement quorum, the baker moves on to state **CE**. Note however that a baker can always go (back) to state **CP** from any state, or remain in state **CP**. This is because bakers get proposals from the network all the time. An endorsement quorum transition can be triggered in any of the collection phases. However an endorsement quorum doesn't cause a baker to change states, it only modifies a baker's internal state to record which block received all necessary votes.
It is important to reiterate that this is a simplified version of Tenderbake. The real version, formalized in [73] contains how bakers interact with the network and how quorums are observed. For the purposes of this chapter, we omit all of those details.

The issue with Cubicle is that when you only have the proof mechanism available, it is very hard to incrementally grow models. A safety proof can take

an arbitrarily long amount of time, and restarting every time you make a small change is tedious and time-consuming. CFL, on the other hand, does not face these problems. We want to show how using CFL allows us to grow our model incrementally, all while verifying it at each step.

We start with an extremely bare-bones version of Tenderbake, where there is only one system-wide proposal that everyone votes on, and when the proposal is accepted, the blockchain grows. To summarize:

- We have a type to describe a baker's internal state
- We have a type to describe a vote
- We have an array mapping a baker to their state and an array mapping a baker to their vote
- Each baker can receive a proposal, we do not care what it is, just that it exists. In the basic version we assume that everyone got the same proposal
- Each baker has a local copy of their blockchain, which we only want to know the length of (i.e. the level)

Now the only subtlety that we want to keep, even in the bare bones version of Tenderbake, is the *quorum*. For this we introduce a new built-in primitive into CFL, `count`. The primitive `count` takes the following form:

```
count(an_Array, a_Value)
```

This will return the number of times `a_Value` appears in the array `an_Array`. We can also write

```
count(an_Array, _)
```

which will simply return how many elements there are in total in the array. We use `count` to simulate quorums. In our basic version of Tenderbake, we insist on a full quorum - i.e. everyone has voted in favor. We will use our previously introduced built-in constant `SYS_PROCS` to avoid hard-coding how many processes are in the system. We give the model in Figure 7.5. Our model has two user-defined types, `state` and `vote`. It has four arrays:

- `BakerState` for each baker's internal state
- `BakerVote` for each baker's vote
- `LocalLevel` for the length of each baker's blockchain copy
- `ProposalExists` to note which bakers have received a proposal

tenderbakeVo.cub

```
type state = NP | CP | CE

type vote = None | PreEndorse | Endorse

array BakerState[proc] : state
array BakerVote[proc] : vote
array LocalLevel[proc] : int

array ProposalExists[proc] : bool

init(i) { BakerState[i] = NP &&
          BakerVote[i] = None &&
          ProposalExists[i] = False &&
          LocalLevel[i] = 0 }

unsafe(i j) { LocalLevel[i] > LocalLevel[j] + 2 }

transition receive_proposal(i)
requires { ProposalExists[i] = False }
{ ProposalExists[i] := True }

transition preendorse(i)
requires { BakerVote[i] = None &&
           ProposalExists[i] = True }
{ BakerVote[i] := PreEndorse;
  BakerState[i] := CP }


transition preendorse_quorum(i)
requires { count(BakerVote, PreEndorse) = SYS_PROCS }
{ BakerVote[i] := Endorse;
  BakerState[i] := CE }


transition endorsement_quorum(i)
requires { count(BakerVote, Endorse) = SYS_PROCS }
{ LocalLevel[i] := LocalLevel[i] + 1;
  BakerState[i] := NP;
  BakerVote[i] := None;
  ProposalExists[i] := False }
```

Figure 7.5: Basic Version of Tenderbake

Initially all bakers are in state NP, have not voted anything, have no proposals, and their blockchain is at length 0. We declare an unsafe state as two bakers that have two blockchain lengths that have a difference of at least two blocks, since Tenderbake offers finality in two blocks.

We then have four transitions:

- receive_proposal: simulates a baker receiving a proposal from the network
- preendorse: a baker preendorses the proposal. This can only be done if the baker has not voted on anything yet
- preendorse_quorum: if everyone has voted on the proposal, a baker can now endorse it
- endorsement_quorum: once everyone has endorsed a proposal, a baker can add a new block to their chain, incrementing the length and resetting their internal variables

We run CFL with three processes to keep our traces easier to read. CFL tells us that the model has a deadlock in eight steps:

receive_proposal(#2) → receive_proposal(#3) → preendorse(#2) → preendorse(#3) → receive_proposal(#1) → preendorse(#1) → preendorse_quorum(#3)

Looking at the trace, we can come to the conclusion that the issue is in the baker changing its vote in preendorse_quorum, effectively blocking everyone else from seeing a quorum. So we need quorums to be separate from votes in order to allow bakers to move independently. We can use Cubicle's matrix notation to simulate a baker communicating to everyone that they had voted. Note that count can also be used on matrices, in the following form:

```
count(A[i], a_Value)
```

We give a new version of our model in Figure 7.6.

Our modified version replaces BakerVote with Endorse and PreEndorse. These are two matrices that simulate a baker "sending a message" to other bakers to indicate that they have (pre)endorsed the proposal.
In the transition preendorse, a baker sets their vote to true for everyone. So for example if we have three bakers #1, #2, and #3, and preendorse is being taken by #1, then the following PreEndorse indexes would be modified, being set to True:

```
PreEndorse[#1, #1]
PreEndorse[#2, #1]
PreEndorse[#3, #1]
```

tenderbakeV1.cub

```
type state = NP | CP | CE

array BakerState[proc] : state
array LocalLevel[proc] : int
array PreEndorse[proc,proc] : bool
array Endorse[proc,proc] : bool
array ProposalExists[proc] : bool

init(i j) { BakerState[i] = NP  &&
            ProposalExists[i] = False &&
            LocalLevel[i] = 0 &&
            PreEndorse[i,j] = False &&
            Endorse[i,j] = False }

unsafe(i j) { LocalLevel[i] > LocalLevel[j] + 2 }

transition receive_proposal(i)
requires {  ProposalExists[i] = False }
{ ProposalExists[i] := True }

transition preendorse(i)
requires { BakerState[i] = NP &&
           ProposalExists[i] = True }
{ PreEndorse[l,m] := case | m = i : True
                          | _ : PreEndorse[l,m];
  BakerState[i] := CP }

transition preendorse_quorum(i)
requires { BakerState[i] = CP &&
           count(PreEndorse[i], True) = SYS_PROCS }
{ Endorse[l,m] := case | m = i : True
                       | _ : Endorse[l,m];
  PreEndorse[l,m] := case | l = i : False
                          | _ : PreEndorse[l,m];
  BakerState[i] := CE }

transition endorsement_quorum(i)
requires { BakerState[i] = CE &&
           count(Endorse[i], True) = SYS_PROCS }
{ LocalLevel[i] := LocalLevel[i] + 1;
  BakerState[i] := NP;
  Endorse[l,m] := case | l = i : False
                       | _ : Endorse[l,m];
  ProposalExists[i] := False }
```

Figure 7.6: New Version of Tenderbake

The same thing happens for `Endorse` in `preendorse_quorum]`. Inversely, when a baker sees a quorum, they reset the values. So again, if we have three bakers and `endorsement_quorum` is being taken by #1, then the following `Endorse` indexes would be modified, being set to `False`:

```
Endorse[#1, #1]
Endorse[#1, #2]
Endorse[#1, #3]
```

This time, if we let CFL run, it does not find any bugs or unsafe states.

We can now complicate our model a bit. We replace our full quorum with 2/3. If we run CFL with three bakers on this new, barely modified version, it finds a couple of deadlocks:

```
receive_proposal(#2) → receive_proposal(#1) →
receive_proposal(#3) → preendorse(#2) → preendorse(#1) →
preendorse_quorum(#3) → preendorse(#3)
```

and

```
receive_proposal(#2) → receive_proposal(#1) →
receive_proposal(#3) → preendorse(#2) → preendorse(#1) →
preendorse(#3)
```

The common element is the three `preendorse` transitions. The mistake comes from setting the quorum *equal* to 2/3, and not greater than. Once we fix that, we can rerun CFL. This time we get something more interesting - an unsafe state. Except the trace ranges from 50 to 90 steps, depending on the execution. But we know that there is an unsafe trace possible, so we can try to get to it manually by using the interpreter.

If we play around with the interpreter, we can quickly come to a conclusion. The reason our model reaches an unsafe state fairly quickly is because we never force all bakers to move. The whole execution could be two bakers voting and growing their blockchains, while the third never moves. What happens in the actual Tenderbake is that proposals are accepted depending on things like the level of the blockchain bakers have or the contents of the proposal. The proposer also changes with every level of the blockchain and round of the consensus algorithm. It is also not that dangerous for bakers to have differences in their blockchain levels - someone could have disconnected and fallen behind, but they can easily catch up by requesting the blockchain from someone. What is dangerous in Tenderbake is when two bakers are at the same level, but the contents of their blocks two levels back is different. Since Tenderbake offers finality in two blocks, that means once there are two blocks in front of a block b, everyone should have the same block b.

```
type state = NP | CP | CE

array BakerState[proc] : state
array LocalLevel[proc] : int

array PreEndorse[proc,proc] : bool
array Endorse[proc,proc] : bool
array ProposalExists[proc] : bool
array Proposal[proc] : proc
array ProposalEndorse[proc,proc] : bool
array ProposalPreEndorse[proc,proc] : bool
```

Figure 7.7: Declarations for Third Version of Tenderbake

We complicate our model further by adding new arrays, as shown in Figure 7.7. We add `Proposal`, `ProposalEndorse`, and `ProposalPreEndorse`. We now let proposals vary. Do to Cubicle's being parameterized by processes, and that still being the case in CFL, we keep `Proposals` as `proc` types, so we can use them in transition arguments. That being said, we could add a subtype declaration as with threads in Chapter 5 to clearly distinguish between "proposal" procs and "baker" procs. We modify the transitions as shown in Figure 7.8. We add `count` for `ProposalEndorse` and `ProposalPreEndorse`, as well as a verification for `Proposal`. This way bakers vote, but they only vote on their specific proposal and check that the others voted on the same proposal before accepting quorums.

If we let CFL run, it seems to run without issues. Note that we removed the the unsafe property. If we stop CFL and look at the environment, it is not ideal. Notable, we get:

```
 BakerState[#1] :  NP
 BakerState[#2] :  CE
 BakerState[#3] :  CP
LocalLevel[#1] :  5033
LocalLevel[#2] :  5011
LocalLevel[#3] :  5036
```

This circles back to our very first problem: bakers never reset their proposal votes, so they can be reused for later cases, leading to large gaps between levels. We can, once again, try to fix this by changing our model. The takeaway from this process is that pure Cubicle at this point would be struggling - we are in a system with eight arrays, which is a lot if we recall the original benchmarks from Chapter 2. The problem is that with complex algorithms

```
transition set_proposal(i j)
requires { ProposalExists[i] = False }
{ Proposal[i] := j;
  ProposalExists[i] := True}

transition preendorse(i j)
requires { BakerState[i] = NP &&
           ProposalExists[i] = True &&
           Proposal[i] = j }
{ PreEndorse[l,m] := case | m = i : True
                          | _ : PreEndorse[l,m];
  ProposalPreEndorse[j,i] := True;
  BakerState[i] := CP }


transition preendorse_quorum(i j)
requires { BakerState[i] = CP &&
           Proposal[i] = j &&
           count(PreEndorse[i], True) >= 2/3 &&
           count(ProposalPreEndorse[j], True) >= 2/3
}
{ Endorse[l,m] := case | m = i : True
                       | _ : Endorse[l,m];
  PreEndorse[l,m] := case | l = i : False
                          | _ : PreEndorse[l,m];
  ProposalEndorse[j,i] := True;
  BakerState[i] := CE }


transition endorsement_quorum(i j)
requires { BakerState[i] = CE &&
           Proposal[i] = j &&
           count(Endorse[i], True) >= 2/3 &&
           count(ProposalEndorse[j], True) >= 2/3
           }
{ LocalLevel[i] := LocalLevel[i] + 1;
  BakerState[i] := NP;
  Endorse[l,m] := case | l = i : False
                       | _ : Endorse[l,m];
  ProposalExists[i] := False }
```

Figure 7.8: Transition for Third Version of Tenderbake

like Tenderbake, a natural approach is to build it up piece by piece, modifying it as you encounter bugs or issues in the behavior. Even in our examples where we stayed in a fairly simple subset of Tenderbake, we still encountered multiple issues that forced us to modify our model. This process with only Cubicle is hard, because the only way to "check" behavior is to run a safety proof. At early stages of modeling, we might not have enough components modeled to express the actual safety property that we want, and finding an intermediate one might not be easy. And proofs are arbitrarily complicated - what might appear to be an easy intermediate safety property may end up forcing Cubicle to run for a long time. Overall, we view CFL as a tool in the development process. Being able to quickly test, modify, and adapt a model is crucial.

Chapter 8

Related Work

Fuzzing has often intersected with formal methods in various forms. It has been applied to check SAT solvers: in [82] the authors introduce a fuzzer to "attack" their SAT solver by giving the solver random inputs and observing its behavior. In a similar fashion, fuzzers have also been applied to SMT solvers, with grammar-based black box fuzzers being used to generate syntactically valid formulas [83, 84, 85]. Similarly, mutation-based fuzzing has also been explored for SMT solvers [86]. Fuzzing has also been used to check model checkers: in [87] the authors propose a fuzzer to automatically generate verification tasks for model checkers.

Fuzzing has been mixed with model checking in various ways. Bounded model checking (BMC) has been used to generate initial seeds for fuzzers [88, 89, 90], under the assumption that BMC would be able to give the fuzzer more interesting and effective seeds than manually created seeds. Similarly, BMC has also been used to generate seeds for paths a fuzzer is not capable of finding on its own [91]: if there is a branch in the code not covered by the fuzzer, BMC is used to create a model, i.e. a set of assignments to variables, for that branch. BMC has also been combined with gray box fuzzing to find vulnerabilities in concurrent programs [92]. Model checking has also served as the inspiration to test Linear-time Temporal Logic (LTL) properties for C++ programs using fuzzing [93]. Parameterized programs are a bit more complicated. To our knowledge, no previous works combine fuzzing with parameterized model checking. In [94], the authors use existing unit tests to generate parameterized tests, which are then passed to a fuzzer. Since the tests are parameterized, the fuzzer simply sets the parameters each time it runs a test. In [95], the authors create a fuzzer that parameterizes the actual fuzzing stages - it adapts to the program it is being run on to optimally select values and strategies.

Fuzzers have been applied to concurrency problems, although less. In [96], the authors propose a fuzzer to detect concurrency bugs in memory. They take into account program input and thread scheduling. In [97], the authors

introduce a fuzzer to deal with kernel concurrency issues by decomposing thread interleavings and mutating the individual parts. The authors in [98] propose a fuzzer that controls thread interleavings and introduces buggy input to see how the program will reacts.

When it comes to blockchains, fuzzing has mainly been used to test smart contracts. Smart contracts are programs that live on the blockchain with which users (human or computer) can interact. Due to being on the blockchain, undoing the actions of a smart contract or modifying a contract after its been deployed is nearly impossible. Fuzzing has been applied to Ethereum smart contracts [99, 100] but it has also been used to find consensus-related bugs in Ethereum [101].

Chapter 9

Conclusion & Perspectives

In this thesis we presented an extension of Cubicle called the Cubicle Fuzzy Loop (CFL). This extension has allowed us to integrate testing into Cubicle. This method has reinforced Cubicle's existing capabilities and introduced new ones. We showed how CFL acts as an extremely efficient forward exploration algorithm for BRAB, allowing us to tackle larger systems that Cubicle struggled with previously. At the same time, being a testing technique, CFL enables quick debugging and execution of models. It has let us adopt an incremental approach to modeling larger examples, and given Cubicle an interactive interpreter, making it more user-friendly. It has also introduced deadlock detection to Cubicle, as well as a more expressive and easier-to-extend language.

The most obvious line of work is taking what we introduced in pure CFL, namely threads, and developing the proof part, so that Cubicle can generate safety proofs and invariants for those features. This requires adding a pre-image calculation and adapting the SMT solver, as well as figuring out how to correctly approximate states that contain thread information. We also want to focus on bettering CFL by improving its stability. As discussed in Chapter 4, CFL is capable of visiting all necessary parts of the system in very few states, but it is not stable. And while this is normal because of its random nature, ideally we want to develop heuristics to make CFL's performance more consistent. In the same vein is developing the parameterized aspect of CFL to keep it as close to Cubicle as possible. Overall, the possibilities to extend CFL are endless, even from an input language point of view. CFL is a good basis for more interactivity and user-friendliness when it comes to Cubicle, and we could build off of that. We could use the feedback aspect of CFL as inspiration to make Cubicle's safety proofs more interactive or "open", in the sense that Cubicle could provide feedback to the user about the proof its running.

Bibliography

[1] Code by the Numbers: How Many Lines of Code in Popular Programs, Apps, and Video Games? https://en.softonic.com/articles/programs-lines-code?

[2] Lines of Code: Microsoft XP. https://www.facebook.com/windows/posts/155741344475532.

[3] Fast-forward — comparing a 1980s supercomputer to the modern smartphone. https://blog.adobe.com/en/publish/2022/11/08/fast-forward-comparing-1980s-supercomputer-to-modern-smartphone.

[4] Google Is 2 Billion Lines of Code—And It's All in One Place. https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/.

[5] John C Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th international conference on software engineering*, pages 547–550, 2002.

[6] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[7] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[8] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on logic of programs*, pages 52–71. Springer, 1981.

[9] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[10] Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[11] Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[12] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards SMT model checking of array-based systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin Heidelberg, 2008.

[13] Silvio Ghilardi and Silvio Ranise. MCMT: A model checker modulo theories. In *IJCAR*, pages 22–29, 2010.

[14] Sylvain Conchon, Alain Mebsout, and Fatiha Zaïdi. Vérification de systèmes paramétrés avec Cubicle. In *JFLA*, Aussois, France, February 2013.

[15] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems: Tool paper. In *CAV*, CAV'12, pages 718–724, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Cc (x): Semantic combination of congruence closure with solvable theories. *Electronic Notes in Theoretical Computer Science*, 198(2):51–69, 2008.

[17] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Invariants for finite instances and beyond. In *2013 Formal Methods in Computer-Aided Design*, pages 61–68. IEEE, 2013.

[18] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[19] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[20] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[21] Lăcrămioara Astefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci Piergiovanni, and Eugen Zalinescu. Tenderbake - a solution to dynamic repeated consensus for blockchains. In *Fourth International Symposium on Foundations and Applications of Blockchain*, 2021.

[22] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions.

In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*, pages 145–157. Springer, 2007.

[23] Parosh Aziz Abdulla, A Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. *Handbook of model checking*, pages 685–725, 2018.

[24] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24-April 1, 2007. Proceedings 13*, pages 721–736. Springer, 2007.

[25] Steven M German and A Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 39(3):675–735, 1992.

[26] Silvio Ghilardi and Silvio Ranise. *MCMT: A Model Checker Modulo Theories*, pages 22–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[27] David L Dill, Andreas J Drexler, Alan J Hu, and C Han Yang. Protocol verification as a hardware design aid. In *ICCD*, volume 92, pages 522–525. Citeseer, 1992.

[28] Alain Mebsout. *Inférence d'invariants pour le model checking de systèmes paramétrés*. PhD thesis, Paris 11, 2014.

[29] Cubicle Model Examples. https://github.com/cubicle-model-checker/cubicle/tree/master/examples.

[30] Edsger W. Dijkstra. Over de sequetialiteit van procesbeschrijvingen [on the sequentiality of process descriptions]. https://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html.

[31] Edsger W. Dijkstra. Cooperating sequential processes, technical report ewd-123. 1965.

[32] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*, pages 289–294. Springer, 2001.

[33] Alain J Martin. A new generalization of dekker's algorithm for mutual exclusion. 1985.

[34] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, aug 1974.

[35] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*. Springer, 2007.

[36] Boleslaw K Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 621–626, 1988.

[37] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4):273–298, 1986.

[38] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[39] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys (CSUR)*, 29(1):82–126, 1997.

[40] Hanan Shukur, Subhi Zeebaree, Rizgar Zebari, Omar Ahmed, Lailan Haji, and Dildar Abdulqader. Cache coherence protocols in distributed systems. *Journal of Applied Science and Technology Trends*, 1(3):92–97, 2020.

[41] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97. Springer, 2001.

[42] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *Verification, Model Checking, and Abstract Interpretation: Third International Workshop, VMCAI 2002 Venice, Italy, January 21–22, 2002 Revised Papers 3*, pages 317–330. Springer, 2002.

[43] Ching-Tsun Chou, Phanindra K Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings 5*, pages 382–398. Springer, 2004.

[44] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, et al. The stanford flash multiprocessor. In *Proceedings of the 21ST annual international symposium on Computer architecture*, pages 302–313, 1994.

[45] Fuzz Testing of Application Reliability. https://pages.cs.wisc.edu/~bart/fuzz/.

[46] Justin Forrester and Barton Miller. An empirical study of the robustness of windows NT applications using random testing. In *4th USENIX Windows Systems Symposium (4th USENIX Windows Systems Symposium)*, Seattle, WA, August 2000. USENIX Association.

[47] Patrice Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.

[48] Michael Howard and Steve Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[49] Microsoft SDL Practices. https://www.microsoft.com/en-us/securityengineering/sdl/practices.

[50] ClusterFuzz. https://google.github.io/clusterfuzz/.

[51] Android Goes All-in on Fuzzing. https://security.googleblog.com/2023/08/android-goes-all-in-on-fuzzing.html.

[52] Guided in-process fuzzing of Chrome components. https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html.

[53] Kostya Serebryany. {OSS-Fuzz}-google's continuous fuzzing service for open source software. 2017.

[54] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142. IEEE, 2021.

[55] OSS-Fuzz. https://google.github.io/oss-fuzz/.

[56] DevSecOps Fundamentals Guidebook. https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOpsActivitesToolsGuidebookTables.pdf.

[57] Mayhem. https://info.forallsecure.com/rs/112-FGI-163/images/br-mayhem-for-code.pdf.

[58] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 4, 2007.

[59] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[60] Hamad Al Salem and Jia Song. A review on grammar-based fuzzing techniques. *International Journal of Computer Science & Security (IJCSS)*, 13(3):114–123, 2019.

[61] Sylvain Conchon and Alexandrina Korneva. The cubicle fuzzy loop : A fuzzing-based extension for the cubicle model checker). to appear in *21st edition of the International Conference on Software Engineering and Formal Methods (SEFM 2023)*.

[62] Michał Zalewski. American fuzzy lop-whitepaper, 2016.

[63] American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[64] Sylvain Conchon, Alexandrina Korneva, and Fatiha Zaïdi. Verifying smart contracts with cubicle. In *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I 3*, pages 312–324. Springer, 2020.

[65] Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci, and Marisa Venturini Zilli. Exploiting transition locality in automatic verification of finite-state concurrent systems. *STTT*, 6(4):320–341, 2004.

[66] Leslie Lamport. Specifying concurrent systems with tla+. *Calculational System Design*, pages 183–247, April 1999.

[67] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[68] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[69] Mordechai Ben-Ari. *Principles of concurrent programming*. Prentice Hall Professional Technical Reference, 1982.

[70] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[71] David Parnas. On a solution to the cigarette smoker's problem (without conditional statements). *Communications of the ACM*, 18:181–183, 03 1975.

[72] Cigarette Smoker's Problem. https://www.cs.umd.edu/~hollings/cs412/s96/synch/smokers.html.

[73] Sylvain Conchon, Alexandrina Korneva, Çagdas Bozman, Mohamed Iguernlala, and Alain Mebsout. Formally documenting tenderbake (short paper). In *3rd International Workshop on Formal Methods for Blockchains (FMBC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[74] LM Goodman. Tezos—a self-amending crypto-ledger white paper. *URL: https://www. tezos. com/static/papers/white paper. pdf*, 2014.

[75] Jae Kwon and Ethan Buchman. Cosmos whitepaper, 2019.

[76] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[77] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[78] Satoshi Nakamoto. Bitcoin : A peer-to-peer electronic cash system. 2009.

[79] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[80] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[81] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.

[82] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of sat and qbf solvers. In *Theory and Applications of Satisfiability Testing–SAT 2010: 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 13*, pages 44–57. Springer, 2010.

[83] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5, 2009.

[84] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. Stringfuzz: A fuzzer for string solvers. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, pages 45–51. Springer, 2018.

[85] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, 2020.

[86] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox

mutational fuzzing. In *Proceedings of the 28th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 701–712, 2020.

[87] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. Finding and understanding bugs in software model checkers. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 763–773, 2019.

[88] Yixiao Yang. Improve model testing by integrating bounded model checking and coverage guided fuzzing. *Electronics*, 12(7):1573, 2023.

[89] Ravindra Metta, Raveendra Kumar Medicherla, and Samarjit Chakraborty. Bmc+ fuzz: Efficient and effective test generation. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1419–1424. IEEE, 2022.

[90] Animesh Basak Chowdhury and Raveendra Kumar Medicherla. Verifuzz: Program aware fuzzing: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25*, pages 244–249. Springer, 2019.

[91] Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. Fusebmc v4: Smart seed generation for hybrid fuzzing: (competition contribution). In *Fundamental Approaches to Software Engineering: 25th International Conference, FASE 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*, pages 336–340. Springer International Publishing Cham, 2022.

[92] Fatimah K Aljaafari, Rafael Menezes, Edoardo Manino, Fedor Shmarov, Mustafa A Mustafa, and Lucas C Cordeiro. Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs. *IEEE Access*, 10:121365–121384, 2022.

[93] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. Linear-time temporal logic guided greybox fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1343–1355, 2022.

[94] Alexander Kampmann and Andreas Zeller. Carving parameterized unit tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 248–249. IEEE, 2019.

[95] Ivica Nikolić, Radu Mantu, Shiqi Shen, and Prateek Saxena. Refined grey-box fuzzing with sivo. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 18th International Conference, DIMVA 2021, Virtual Event, July 14–16, 2021, Proceedings 18*, pages 106–129. Springer, 2021.

[96] Nischai Vinesh and M Sethumadhavan. Confuzz—a concurrency fuzzer. In *First International Conference on Sustainable Technologies for Computational Intelligence: Proceedings of ICTSCI 2019*, pages 667–691. Springer, 2020.

[97] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2104–2121. IEEE Computer Society, 2023.

[98] Youngjoo Ko, Bin Zhu, and Jong Kim. Fuzzing with automatically controlled interleavings to detect concurrency bugs. *Journal of Systems and Software*, 191:111379, 2022.

[99] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020.

[100] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.

[101] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 349–365, 2021.