FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reduction Techniques for Parameterized Model Checking and Synthesis of Fault-Tolerant Distributed Algorithms

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

eingereicht von

Marijana Lazić, MSc
Matrikelnummer 01429677

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dr. Josef Widder
Zweitbetreuung: Dr. Igor Konnov

Diese Dissertation haben begutachtet:

Orna Grumberg

Javier Esparza

Josef Widder

Wien, 29. April 2019

Marijana Lazić

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Reduction Techniques for Parameterized Model Checking and Synthesis of Fault-Tolerant Distributed Algorithms

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

Marijana Lazić, MSc
Registration Number 01429677

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dr. Josef Widder
Second advisor: Dr. Igor Konnov

The dissertation has been reviewed by:

Orna Grumberg

Javier Esparza

Josef Widder

Vienna, 29th April, 2019

Marijana Lazić

Erklärung zur Verfassung der Arbeit

Marijana Lazić, MSc
Favoritenstraße 9-11
1040 Wien
Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. April 2019

Marijana Lazić

*I dedicate this thesis to the memory of Helmut Veith,
who was my advisor in the first year of my PhD studies.
Thanks to him, I fell in love with this topic,
and found my place in the verification community.
Helmut laid the foundation of my achievements
that I myself did not believe I could accomplish.*

Acknowledgements

This thesis is the final result of a four-year-long memorable journey, where hard work and pure joy have been constantly intertwined. I believe that these were the years of my biggest progress in life, and I have a need to thank all those people who contributed to this journey in any way, scientifically or privately, and there are many of them (more than I will be able to mention here).

First and the most important, the whole journey would not have even started without Helmut Veith. I am sad that I cannot personally thank him for being such an inspiring teacher, an empathic supervisor, full of support, who strove for the best for his students. A true role model, today as well. Helmut was the one who introduced me to the world of formal methods. He believed in me, even when I did not. I am grateful I had a chance to work with him even for one year. I hope he would be proud of me and my achievements today.

During several years of closely working with Helmut, both Igor Konnov and Josef Widder absorbed all his invaluable skills, so I could not have found a better advising team. This inseparable duo was always there for me, to listen to my practice talks (even in hotel lobbies or on Saturdays when it was important), to answer my questions (and there were many of them, trust me), to offer advice whenever it was needed, to think about my future career, to encourage me to believe in myself, to push me to always give my best, and much more. I am greatly indebted to them for their care and support.

Five years ago, studying computer science in Vienna was just a dream for me. Big thanks to my friends Dušan Radičanin and Nenad Savić, and to Agata Ciabattoni, who persuaded me to apply for this PhD program (together with Marko Kovačević, but he will get his paragraph later). My biggest support in the pre-PhD days was my master thesis supervisor Rozália Madarász Szilágyi, who was actually the first one to suggest applying my theoretical knowledge in this particular PhD school. She recognized my talent and persuaded me to use it wisely. I learned a lot from her, including the basics of mathematical logic, analytical thinking, and teaching. Erhard Aichinger and Nebojša Mudrinski played a big role in my first steps in science, especially in writing articles and giving talks; I fortunately had a chance to work closely with them before enrolling in these studies.

During my PhD studies I was lucky to get in touch with many brilliant scientists, from TU Wien as well as from other universities. I have been a proud member of the

FORSYTE research group. Our frequent meetings have strengthened my knowledge in formal methods and I am thankful to all of my FORSYTE colleagues for creating such a productive environment. As my unofficial co-advisor, Roderick Bloem has always been a huge support for me. He has always showed interest in my work, and our discussions lead to results that are included in this thesis. From Natalie Bertrand I learned a lot about probabilistic systems. Our fruitful discussions initiated at UC Berkeley led to great results, also included in the thesis. Ulrich Schmid inspired my interest in analysis of distributed algorithms from different perspectives. Vijay D'Silva introduced me to the world of topology in distributed systems. Besides being an inspiring colleague, Vijay has always been an inexhaustible source of useful advice. From Yoram Moses I learned a lot about epistemic logic, that allowed my passion for logic to be combined with distributed algorithms. Yoram is the most positive person I know and a highly cordial host, so working with him was a pleasure and an honor. I also enjoyed our discussions with Sergio Rajsbaum, Jérémy Ledent, and Éric Goubault, who also financially supported my visits to them. During my visit to Israel, I also had a chance to meet Sharon Shoham and Oded Padon. Our exciting discussions resulted in a paper on verification of threshold-based distributed algorithms by decomposition to decidable logics, published at CAV 2019 (that is not part of this thesis). Orna Grumberg and Javier Esparza read the first version of this thesis and gave valuable feedback, including highly encouraging comments. I particularly appreciate that they accepted to review my thesis, as I met both of them in the early stage of my studies, so they had a chance to follow my work through all its later stages.

I am grateful for the generous financial support provided by the Austrian Science Fund (FWF) through the Doctoral College LogiCS (W1255-N23), National Research Network RiSE (S11403), and project PRAVDA (P27722); and by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103).

Being a part of the doctoral school Logical Methods in Computer Science (LogiCS) was a true blessing. In the last four years, LogiCS has been like my extended family. This was particularly important for me at the beginning of my studies, as it would be for any foreign student. Anna Prianichnikova played an important role in making this research group a family. She was like a mother to us, making me feel at home from the first day in Vienna. The help of my office mates was invaluable: Ilina Stoilkovska, Jure Kukovec, Thanh Hai Tran, Labinot Bajraktari, Martin Diller, and Shqiponja Ahmetaj would always patiently answer all my naïve questions, they proofread many of my write-ups, solved many Git-related problems, etc. They have been a big support and they made my everyday life more entertaining. Still, lunch breaks were regularly reserved for a good laugh with Benjamin Kiesl, Alëna Rodionova, Gerald Berger, Anna Lukina, Ilina and the others. For a special treatment at Cafe Schrödinger and desserts for free on Fridays, I need to thank the best waitress Mara Jakešević.

Special thanks to Benjamin, Ilina and Gerald for helping me on an almost everyday basis with any kinds of issues. For example, Gerald translated the abstract of this thesis to German and spotted several typos; our race in submitting the thesis ended up

being a perfect additional motivation. Benjamin actively participated in searching for an apartment for me in Vienna, but also played a significant role in finding one in Munich; having him as a friend is a precious gift. For Ilina I would not know where to start: from helping me with verification tools, listening to my practice talks multiple times, or just always being there for me, to understand me and to comfort me.

For all the practical and legal issues, our institute's secretaries Beatrix Buhl, Juliane Auerböck, and Eva Nedoma were lifesavers. They would spend hours calling MA35 or reading Austrian (and lately, German) laws, to help me apply for residence permits or other visas. Another lifesaver is our group's system administrator Toni Pisjak, who was so kind to solve not only all the troubles with my office PC in record time, but even with my private laptop. I would sometimes come to his office with a trivial question, and he would always take it very seriously.

Last but not the least, I would like to thank to my family members. My sister Nataša was my childhood role model and today she is my perfect example of a strong woman. I thank her for teaching me to fight for myself.

This whole journey would be unimaginable without my husband Marko. I can never forget that he moved to Vienna because of me and then spent months alone waiting for me to return from research stays, summer and winter schools, and conferences. Nevertheless, he has always been excited to hear about the experience I gained during these trips, and has always been proud of me. I appreciate his special way of dealing with my impostor syndrome: he would mockingly agree that my doubts are justified and encourage me to just keep "pretending". Marko has been my source of energy, my complement, my best friend. These years have been wonderful thanks to him.

Let me finish this with a few words in Serbian, to let my parents know how grateful I am for their whole-life unconditional love and support, for raising me in the spirit of true values, and for giving me wings and letting me fly freely. I am what I am today thanks to them!

Dragi mama i tata, od srca sam vam zahvalna za naizmernu ljubav i podršku koju ste mi u životu pružili. Hvala vam što ste me vaspitali da cenim prave vrednosti i da budem dobar čovek. Hvala vam što me nikada niste sputavali već ste mi dali krila i pustili me da letim slobodno. Ja sam danas ovo što jesam zahvaljujući vama!

An immense thanks to all of these incredible people. I am truly blessed for having you in my life!

Marijana Lazić
Vienna, April 2019

Kurzfassung

Verteilte Algorithmen finden zahlreiche Anwendungen im täglichen Leben, in eingebetteten Systemen der Automobil- und Flugzeugindustrie, in Computer Netzwerken, sowie dem Internet der Dinge. Die zentrale Idee ist Zuverlässigkeit durch Replikation zu erreichen und sicherzustellen, dass alle korrekten Replika als Einheit agieren, selbst wenn manche Replika ausfallen. Dadurch ist die korrekte Operation des Systems zuverlässiger als die Operation seiner einzelnen Teile. Fehlertolerante Algorithmen sind typisch für jene Anwendungen, welche die höchste Ausfallsicherheit verlangen, da in diesen Anwendungen menschliches Leben gefährdet wird, wie zum Beispiel in der Automobil- und Flugindustrie, und selbst unwahrscheinliche Ausfälle dieser Systeme unakzeptabel sind. Darum ist die Korrektheit fehlertoleranter Systeme von zentraler Bedeutung.

Neue Anwendungsgebiete wie Cloud Computing und Blockchain Technologien schaffen neue Motivation zur Untersuchung fehlertoleranter Algorithmen. Mit einer immer größeren Anzahl an involvierten Computern sind Fehler eher die Norm als die Ausnahme. Dadurch wird Fehlertoleranz nur wirtschaftlichen Notwendigkeit und somit auch die Korrektheit der Mechanismen zur Fehlertoleranz.

Um mit praktischen verteilten Systemen, welche aus tausenden Komponenten bestehen, umgehen zu können, bedarf es Techniken zur Analyse fehlertoleranter verteilter Algorithmen für jede Systemgröße, das heißt für jede mögliche Anzahl an Komponenten und jede zulässige Anzahl an Defekten. Wir verwenden spezielle Variablen n, f und t, genannt *Parameter*, um entsprechend die Anzahl der Prozesse, die Anzahl der Defekte, und eine Oberschranke für die Anzahl der Defekte zu beschreiben. Wir betrachten einen Algorithmus als korrekt, wenn dieser seine Spezifikation unter jeder Wahl von n, f, t erfüllt, wobei wir Vorbedingungen, wie zum Beispiel $n > 3t$, die sogenannte *Resilience Condition*, erlauben. Die Verifikation dieser generellen Korrektheit nennt man *parameterisierte Verifikation*, und die Synthese korrekter Algorithmen aus deren Spezifikationen wird *parameterisierte Synthese* genannt.

Das Schlussfolgern über die Korrektheit von fehlertoleranten verteilten Algorithmen (FTDAs) ist durch deren nichtdeterministisches Verhalten aufgrund von Defekten, Nebenläufigkeit und Nachrichtenverzögerung eine immens schwierige Aufgabe. Der klassische Ansatz zur Feststellung von Korrektheit besteht in mathematischen Beweisen durch Bleistift und Papier, welche offensichtlich menschlichen Einfallsreichtum und großen manuellen Aufwand erfordern. Da verteilte Algorithmen komplexes Verhalten aufweisen

und da diese für Menschen schwer verständlich sind, gibt es nur sehr wenige Werkzeuge, um logische Fehler in fehlertoleranten verteilten Algorithmen aufzuspüren, insbesondere in der Entwurfsphase.

In dieser Dissertation befassen wir uns mit der Klasse von asynchronen FTDAs, wobei Prozesse durch Message Passing kommunizieren und die empfangenen Nachrichten mit sogenannten *Tresholds* vergleichen. Zum Beispiel mag es einem Prozess nur dann erlaubt sein, eine Operation auszuführen, wenn dieser von der Mehrheit der Prozesse im System eine Nachricht empfangen hat, d.h. mindestens $\frac{n}{2}$ Nachrichten. Tresholds sind üblicherweise Linearkombinationen von Parametern und daher Parameterisierungen des Programmcodes. Für solche FTDAs sind die meisten existierenden Verifikations- und Synthesetechniken entweder nur für kleine Instanzen geeignet, zum Beispiel mit vier Prozessen von denen einer ausfallen darf, oder ungeeignet für (i) parameterisierten Programmcode, (ii) arithmetische Resilience Conditions, oder (iii) die Spezifikation verteilter Algorithmen.

In dieser Dissertation entwickeln wir Techniken zur parameterisierten Analyse dieser Klasse von FTDAs, welche den erwähnten Herausforderungen gerecht werden. Diese Techniken basieren auf *Reduktion*, das heißt auf dem Schließen über Abhängigkeiten von nebenläufig ausgeführten Ereignissen. Der Beitrag dieser Dissertation is dreifältig:

1. Wir präsentieren ein komplettes Framework zur parametersierten Verifikation von temporalen Eigenschaften dieser speziellen Klasse von FTDAs, insbesondere eine Methode zur Zertifizierung der Korrektheit eines Algorithmus für jede Systemgröße. Unsere Methode adressiert sowohl Safety- als auch Liveness-Spezifikationen von Algorithmen.

2. Wir erarbeiten eine automatisierte Technik zur Synthese von FTDAs, welche korrekt für jede Systemgröße ist und nicht nur für jene fixierter Größe. Für ein gegebenes sogenanntes Skeletts eines Algorithmus unserer Klasse erkennen wir automatisch Tresholds, parameterisiert in der Anzahl der Prozesse und der Anzahl der Defekte, welche korrekte Algorithmen liefern.

3. Zusätzlich erweitern wir die Klasse der betrachteten Algorithmen um die Fähigkeit, probabilistische Übergänge auszuüben, wie zum Beispiel Münzwürfe und Spezifikationen, welche mit Wahrscheinlichkeit 1 gelten. Wir reduzieren das Problem der Verifikation solcher *randomisierter FTDAs* auf das bekannte Szenario. Dadurch entwicklen wir die erste automatisierte Methode zur parameterisierten Verifikation von randomisierten, fehlertoleranten, verteilen Algorithmen und deren probabilistischen Eigenschaften.

Abstract

Distributed algorithms have many applications in everyday life, as well as in avionic and automotive embedded systems, computer networks, and the Internet of Things. The central idea is to achieve dependability by replication, and to ensure that all correct replicas behave as one, even if some of the replicas fail. In this way, the correct operation of the system is more reliable than the correct operation of its parts. Fault-tolerant algorithms are typically used in applications where highest reliability is required because human lives might be at risk, such as in automotive or avionic industries, and even unlikely failures of the system are not acceptable. Therefore, correctness of fault-tolerant algorithms is of utmost importance.

New application domains, such as cloud computing or blockchain technologies, provide new motivation to study fault-tolerant algorithms. With the huge number of computers involved, faults are the norm rather than the exception. Consequently, fault-tolerance becomes an economic necessity, and so does the correctness of mechanisms for fault-tolerance. In order to deal with real-life distributed systems with thousands of components, we need techniques for analyzing fault-tolerant distributed algorithms for every system size, that is, for every number of components and every admissible number of faults.

We use special variables n, f and t, called *parameters*, to describe the number of processes in the system, the number of faults, and an upper bound on the number of faults, respectively. An algorithm is considered to be correct if it satisfies its specifications for every n, f, t, under a certain prerequisite, e.g., $n > 3t$, called *resilience condition*. Verifying such general correctness is called *parameterized verification*, and synthesizing correct algorithms from their specifications is in this case called *parameterized synthesis*.

Reasoning about correctness of fault-tolerant distributed algorithms (FTDAs) is an immensely difficult task, due to the non-deterministic behavior caused by the presence of faults, concurrency, message delays, etc. The classical approach toward correctness consists of pencil-and-paper mathematical proofs, that require human ingenuity and huge manual efforts. As distributed algorithms show complex behavior, and are difficult to understand for human engineers, there is only very limited tool support to catch logical errors in fault-tolerant distributed algorithms, especially at design time.

In this thesis we focus on the class of asynchronous FTDAs where processes communicate by message passing and compare numbers of received messages against so-called

thresholds. For instance, a process might be allowed to perform an action only if it has received messages from a majority of processes in the system, i.e., at least $\frac{n}{2}$ messages. Thresholds are typically linear combinations of parameters, and thus they constitute a parameterization of the code itself. For asynchronous FTDAs, most existing verification and synthesis techniques are either only able to deal with small instances, for example, with four processes among which one might fail, or they cannot deal with (i) parameterized code, (ii) arithmetic resilience conditions, or (iii) specifications of distributed algorithms.

In this thesis we develop techniques for the parameterized analysis of this class of FTDAs that can cope with all the mentioned challenges. More specifically, our methods are based on *reduction* techniques, that is, on reasoning about dependency of concurrently executed events. The contribution of this thesis is threefold:

1. We present a complete framework for the parameterized verification of temporal properties of this particular class of FTDAs, that is, a method for certifying the correctness of an algorithm for any system size. Our method addresses both safety and liveness specifications of algorithms.

2. We introduce an automated technique for synthesizing FTDAs that are correct for any system size, rather than for a fixed size. In fact, for a given so-called skeleton of an algorithm from our class, we automatically detect thresholds, parameterized in the number of processes in total and the number of faults, that yield correct algorithms.

3. Additionally, we extend the class of the considered algorithms, by allowing probabilistic transitions, such as coin tosses, and specifications that hold with probability 1. We reduce the problem of verification of such *randomized FTDAs* to the non-probabilistic setting. In this way, we present the first automated method for parameterized verification of randomized fault-tolerant distributed algorithms and their associated probabilistic properties.

Contents

Thesis Publications

This thesis is a collection and an extension of the following work, where I contributed as a co-author: one journal paper [KLVW17a], two conference papers [KLVW17b, LKWB17], a conference submission [BKLW18], and partially using the reasoning outlined in a workshop position paper [DLW18]. Throughout the thesis, we will state which chapter is based on which work, but we will also quote the papers in verbatim without explicitly referring to them.

[KLVW17a] Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. Para^2: Parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. *Formal Methods in System Design*, 51(2):270–307, 2017

[KLVW17b] Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, pages 719–734, 2017

[LKWB17] Marijana Lazić, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *OPODIS*, volume 95 of *LIPIcs*, pages 32:1–32:20, 2017

[BKLW18] Nathalie Bertrand, Igor Konnov, Marijana Lazić, and Josef Widder. Verification of Randomized Distributed Algorithms under Round-Rigid Adversaries. *HAL*, hal-01925533, Nov 2018

[DLW18] Cezara Drăgoi, Marijana Lazić, and Josef Widder. Communication-closed layers as paradigm for distributed systems: A manifesto. In *Sinteza*, pages 131–138, 2018

Individual contribution. All the results presented in this theses have been conducted in a team that typically consists of my advisers and myself, occasionally including an external collaborator. As this was an equally distributed joint collaboration, followed by collective discussions, it is impossible to precisely determine the role of each participant. This is expressed by the alphabetic order of authors in all the papers mentioned above except in [LKWB17], where I am the first author, which emphasizes my contribution.

Still, as a mathematician by training, I mostly contributed to the development of the framework and the theoretical results, with the emphasis on the reduction techniques that are the central concept of this thesis. Implementation and experimental evaluation were done by my co-authors, and I only contributed conceptually to this.

CHAPTER 1

Introduction

In the era of modern technologies and high tech development, each of us is more or less dependent on complex computer systems. These systems are progressively becoming distributed. Our everyday activities depend on the Internet, where geographically distant computers communicate with each other. Even seemingly simple tasks, like reading newspapers online or having long-distance video calls, require sophisticated distributed algorithms. We expect them to work correctly, but often they do not. Let us take an example of a distributed problem which can be found in a daily routine such as having chat with several users via a social network. If some of the users, or server, got disconnected, we expect that the other users can continue to chat and that the system provides history information to other users, including messages sent by the disconnected users.

Moreover, in complex distributed systems like cloud services or databases, failures of individual components are a commonplace phenomenon [Net10]. We often encounter environmental faults, like power outages, memory corruption, or network misconfiguration, as well as software bugs caused by the programmer's oversight. This problem calls for fault-tolerance mechanisms that can deal with different types of failures.

For example, the different components of Amazon cloud storage service S3, are spread all over the United States and Europe, and need to be aware of the states of each other. In July 2008 [Tea08], message corruption caused problems in server-to-server communication. The existing fault-tolerance mechanism of Amazon S3 was not able to deal with this kind of faults. Consequently, when the corruption occurred, it spread throughout the system. To recover, Amazon had to take the service offline for more than seven hours. This caused a vast financial damage, and therefore motivated more investments in verification of fault-tolerant distributed systems.

Furthermore, currently we observe increasing interest in blockchain technology [Nak08, Ext15, GOS16, Buc16], where a large number of persons, companies, etc., are interested in participating in a market place. These systems have to ensure fault tolerance, e.g., if a majority of the participants are honest, the system should work as expected. Besides, they provide probabilistic guarantees, which demands analysis of randomized distributed algorithms for any number of participants.

Such examples of distributed algorithms, whose failures can cause a huge financial loss, or put human lives at risk, like in the case of airplanes, cars, and medical devices, teach us that the correctness of fault-tolerant distributed algorithms is of immense importance.

Replication is a classic approach to make computing systems more reliable. In order to avoid a single point of failure, one introduces multiple processes in a distributed system. Then, if some of these processes fail (e.g., by crashing or deviating from their expected behavior) the distributed system as a whole should stay operational. For this purpose one uses fault-tolerant distributed algorithms (FTDAs). These algorithms have been extensively studied in distributed computing theory [Lyn96, AW04], and they found application in safety critical systems (e.g., automotive or aeronautic industry). With the recent advent of data centers, cloud computing, and blockchain technologies, we observe growing interest in fault-tolerant distributed algorithms, and their correctness, also for more mainstream computer science applications [LBC16, PTP$^+$16, DHV$^+$14, DHZ16, PMP$^+$16, WWP$^+$15, HHK$^+$15].

The classical approach for certifying correctness of algorithms in distributed computing literature are hand-written mathematical proofs. One downside of this approach is that these proofs are written for specific algorithms, and thus cannot be easily generalized. Moreover, they are usually highly complex, as they require reasoning about faults, concurrency, non-determinism, and message delays. Consequently, even published algorithms and their proofs can contain bugs [LR93]. Establishing correctness manually demands human ingenuity, huge efforts and a lot of time.

To overcome the problem of hand-written proofs, one may consider automated verification. However, designing automated techniques for verification of fault-tolerant distributed algorithms is a notably difficult task. One problem comes from the scale. The existing automated techniques often cannot cope with complex distributed systems, and therefore focus on small instances, with a fixed number of processes, typically less than 10 [JKS$^+$13b, Gme15]. Large realistic systems, containing thousands of participants, require techniques that do not depend on the number of components, ideally they should work for an unbounded number of components. This precisely formalizes the need for *parameterized* analysis of distributed algorithms. In other words, we generally introduce parameters n, t and f, where n represents the total number of processes in the system, f is the number of faulty processes, and t is the upper bound on the number of faults. It is natural that one cannot expect correct behavior of a system, if, for example, all processes are faulty, formally, if $n = f$. We consider an algorithm to be correct if it satisfies its specifications for any values of parameters provided that, for example, less than a third of processes might fail, that is, $n > 3t$. This requirement is called *resilience condition.*

Checking if an arbitrary distributed system satisfies an arbitrary temporal formula in the parameterized case, is known to be undecidable [BJK$^+$15]. These theoretical restrictions thus introduce a trade-off between generality and the degree of automation. In this work we give priority to automation, and focus on one class of distributed algorithms. We explain characteristics of this class in Section 1.1, and illustrate a typical example of an FTDA of interest. The problem we address in this thesis is presented in Section 1.2. The existing work on this topic, using different approaches, is summarized in Section 1.3. For solving the problem, we employ a method called reduction, that is described in Section 1.4. Finally, a guide through the main body of the thesis is given in Section 1.5.

1.1 Fault-tolerant Distributed Algorithms

A fault-tolerant distributed system is a collection of independent sequentially computing components, called processes, that cooperate to achieve a common goal in the presence of failures. A protocol that defines the behavior of each correct process in such a system, is given as a fault-tolerant distributed algorithm (FTDA) [Lyn96, AW04].

There are two common means of communication in a distributed system:

- *message passing*, that is, processes send messages to each other over communication channels that connect pairs of processes, or
- *shared memory*, that is, processes store data in a memory that is accessible to other processes (not necessarily all).

Regarding timing assumptions, the literature distinguishes three models:

- *synchronous model* [Lyn96], where processes operate in lockstep. Every execution of the system is partitioned in rounds, and in each round processes exchange messages (send and receive them) and perform local computations. We say that individual executions of processes are fully synchronized. Although convenient for analysis, this model is not achievable in practice.
- *asynchronous model* [FLP85] allows more realistic modeling, where processes are arbitrarily interleaved with no upper bound on the time taken for message delivery and no upper bound on the speeds of individual processes. Even in a fault-free algorithm, with a fixed input, asynchronous modeling gives rise to multiple executions, depending on the speeds of processes and message deliveries.
- *partially synchronous model* [DDS87, DLS88], which is a more restricted asynchronous model, where processes are interleaved with certain limits on message delays and/or the relative process speeds.

In this work we focus on asynchronous algorithms with the message passing model of communication. Processes are fully *symmetric*, that is, they do not have process IDs, and the message passing model is restricted in such a way that correct processes can only broadcast messages, that is, send to all. The events of receiving such a message are not synchronized. Moreover, processes keep track of the number of received messages and compare it against so-called *thresholds*. A process is allowed to progress only if its number of received message has reached the threshold. FTDAs with these characteristics are called *threshold-based FTDAs*. Typical examples of such algorithms are folklore reliable broadcast (FRB) [CT96], consistent broadcast (STRB) [ST87b], asynchronous Byzantine agreement (ABA) [BT85], condition-based consensus (CBC) [MMPR03], non-blocking atomic commitment (NBAC and NBACC [Ray97] and NBACG [Gue02]), one-step consensus with zero degradation (CF1S [DS06]), consensus in one communication step (C1CS [BGMR01]), and one-step Byzantine asynchronous consensus (BOSCO [SvR08]).

Threshold-based FTDAs. Faulty behavior in distributed systems can appear in many different forms [GKS^{+}14, Gme15]. For example, processes may just stop functioning (*crash faults*), may fail to send messages to certain participants (*omission faults*), or they may send messages with erroneous content, or even conflicting information to different processes (*Byzantine faults*). Many algorithms achieve fault-tolerance by using *threshold guards* that, for instance, ensure that a process waits until it has received an acknowledgment from a majority of its peers.

In asynchronous systems, processes do not have access to the global state. Thus, they progress based on their local states and received messages. If the modeling does not allow faults, we often encounter simple threshold guards, such as existential guards:

```
if received <m> from some processes
then action(m);
```

and universal guards:

```
if received <m> from all processes
then action(m);
```

Nevertheless, as discussed in [GKS^{+}14], these guards do not tolerate faults. Namely, if one uses existential guards in the presence of Byzantine faults, and the sender of a message $<\mathtt{m}>$ is faulty, then the content of the message might be erroneous, and a receiver would make its local computation based on this misleading information; If one uses universal guards, and a faulty process omits to send $<\mathtt{m}>$, then this message will never be received, and thus every correct process will wait forever as the guard prevents it from progressing. Therefore, it is crucial to be able to express different conditions, for example, that a process should wait until it receives at least one message from a correct process, or until it receives messages from majority of processes. Algorithms that use threshold guards are called *threshold-based algorithms*, and in this thesis we develop techniques for analysis of threshold-based FTDAs.

```
 1  var $myval_i \in \{0,1\}$
 2  var $accept_i \in \{false, true\} \leftarrow false$
 3
 4  while true do (in one step)
 5    if $myval_i = 1$
 6       and not sent ECHO before
 7    then send ECHO to all
 8
 9    if received ECHO from at least $t+1$ distinct processes
10       and not sent ECHO before
11    then send ECHO to all
12
13    if received ECHO from at least $n-t$ distinct processes
14    then $accept_i \leftarrow true$
15  od
```

Figure 1.1: Pseudo code of a distributed algorithm for a correct process i. A distributed system consists of n processes, at most $t < n/3$ of which are Byzantine faulty.

Example FTDA. A typical example of a threshold-based fault-tolerant distributed algorithm is reliable broadcast. Figure 1.1 shows a pseudo code inspired by the reliable broadcast protocol from [ST87b], which is used as building block of many fault-tolerant distributed systems. It tolerates Byzantine faults, that is, every correct process follows the pseudo code, and no assumption is made about faulty processes. Note that parameters appear already in the pseudo code: the system contains n processes, and at most t of them are faulty. We allow at most $t < n/3$ faulty processes, and this is called *resilience condition.*

The statements from lines 9 and 13 are called *threshold guards*, and the expressions $t+1$ and $n-t$ from these lines are *thresholds.* Waiting for messages from $n-t$ processes ensures that if all correct processes send messages, then faulty processes cannot prevent progress. Similarly, waiting for $t+1$ messages ensures that at least one message was sent by a correct process.

Each correct process i has initial value 0 or 1 (line 1). Processes with initial value 1 broadcast the ECHO message (lines 5-7), i.e., send ECHO to all processes including themselves, exactly once. Independently of its initial value, every process that has received at least $t+1$ messages from its peers, but has not sent ECHO, also broadcasts the ECHO message (lines 9-11). Every correct process that has collected at least $n-t$ ECHO messages, sets its accept value to true, or simply, accepts (lines 13-14). We assume that every message that is sent, eventually it is received, and this is called *fairness assumption.*

Example specifications. This algorithm is correct if for every value of n and t such that $n > 3t$, the following holds:

Unforgeability If all correct processes have initial value 0, then no correct process ever accepts.

Correctness [1] If all correct processes have initial value 1, then there is a correct process that eventually accepts.

Relay If a correct process accepts, then eventually all correct processes accept.

Does the algorithm satisfy the specifications? Next we consider three cases that illustrate the role of threshold guards and the resilience condition.

Let us first consider the fault-free case, i.e., when $t = 0$. Our threshold guards $t + 1$ and $n - t$ become existential ($t + 1 = 1$) and universal ($n - t = n$), respectively. We analyze behavior of the parameterized system in the two possible cases: (i) when all processes have initial value 0, and (ii) when $k \geq 1$ processes have initial value 1.

(i) If no process has initial value 1, then no process ever sends the ECHO message, and therefore no process ever accepts. This proves unforgeability.

(ii) If there are exactly $k \geq 1$ processes with 1 as their initial value, each of them will send the ECHO message. Then, by our assumption that every sent message is eventually received, each process eventually receives at least k ECHO messages. As $k \geq 1$, every process will reach the threshold $t + 1 = 1$ in line 9, and thus each of the n processes will send ECHO (even those $n - k$ processes whose initial value is 0). Thus, again by the fairness assumption, every process eventually receives n ECHO messages, and accepts. For $k = n$, this proves correctness, and in general, it implicitly proves relay.

This confirms that existential and universal guards are powerful enough in the case with no faults.

Next we consider the system in the presence of faults, but with respect to the resilience condition $n > 3t$. Let us analyze a particular instance of the system, that is, let us assume that the system consists of, for example, four processes and exactly one of them is faulty, that is, $n = 4$ and $t = 1$. We show (i) that the system satisfies unforgeability, even if the faulty process sends the ECHO message to the others. Moreover, we show (ii) that a slight modification of a threshold can violate unforgeability.

(i) Assume that the three correct processes have initial value 0, but the faulty one sends ECHO to some or all of them. The threshold guard $t + 1 = 2$ will never be reached, and thus, no other process will ever broadcast ECHO, nor accept.

[1] We kept the name of this specification, *correctness*, as it is originally introduced in [ST87b], but one should not confuse it with the notion of a *correct system*, which means that the system satisfies all its specifications, in this case: unforgeability, correctness, and relay.

(ii) To illustrate the importance of the threshold guards, we can set the threshold from the line 9 to t instead of $t+1$, and give the same example, where one faulty process sends the ECHO message to the correct ones that have initial values 0. They will eventually receive this message, which would be enough for their own broadcast, and implicitly later for accepting. This would violate unforgeability.

Here we also stress the role of the resilience condition $n > 3t$. Assume it is only slightly changed, that is, assume we have $n \geq 3t$ instead of $n > 3t$. Then, the algorithm is not correct. Let us take the simplest example, with $n = 3$ and $t = 1$. Note that unforgeability and correctness are still satisfied, but relay can be violated. Assume that two correct process are i and j, where i has initial value 0, and j has 1. If the Byzantine process sends ECHO to j, but no message to i, then j eventually receives two messages (from itself and from the faulty one) and accepts, but i never receives more than one message (from j), and thus never accepts. This violates relay.

We have seen that parameterized verification of distributed algorithms highly depends on the interplay between resilience conditions, precise formulation of specifications, and parameter values.

1.2 Problem statement

In this thesis we focus on verification and synthesis of fault-tolerant distributed algorithms. To formalize threshold-based fault-tolerant distributed algorithms we use *threshold automata*. The notion has been introduced for the first time in [KVW14], and it represents an abstraction of the original system. Threshold automata are edge-labeled graphs, where nodes represent local states of individual processes, and edges represent transitions that can be executed by a process only if the threshold guard labeling the edge is true. A threshold automaton captures the behavior of one process, and the distributed system is modeled as a system of typically n copies of a threshold automaton.

A parameterized distributed system $M(n,t,f)$ is now defined as follows:

$$M(n,t,f) = \underbrace{P(n,t,f) \parallel P(n,t,f) \parallel \ldots \parallel P(n,t,f)}_{n-f \text{ times}} \parallel \underbrace{\mathit{Faulty} \parallel \ldots \parallel \mathit{Faulty}}_{f \text{ times}} \tag{1.1}$$

It is a parallel composition of n individual sequentially computing processes, that either follow the process code $P(n,t,f)$ if they are correct, or deviate from it in case of a faulty process (Faulty). For instance, $P(n,t,f)$ can be a formalization of the code from Figure 1.1. We are interested in asynchronous timing model, as described in Section 1.1.

Such systems should satisfy temporal properties. Every specification is either safety, or liveness, or their combination [AS87, Lam77]. Intuitively, safety means that nothing bad can ever happen in the system, and liveness means that something good eventually happens. In the example from Section 1.1, unforgeability is a safety property, while correctness and relay are liveness properties. Classical verification methods often focus on safety properties, but for fault-tolerant distributed algorithms liveness is as important as safety. It is a folklore knowledge that designing a safe fault-tolerant distributed algorithm is trivial: *just do nothing*; e.g., by never committing transactions, one cannot commit them in inconsistent order. Hence, a technique that verifies only safety may establish the "correctness" of a distributed algorithm that never does anything useful. To achieve trust in correctness of a distributed algorithm, we need tools that verify both safety and liveness. A formal definition of the system correctness is given as follows:

Given a process code $P(n, t, f)$, a temporal formula φ, and a resilience condition $RC(n, t, f)$, the system $M(n, t, f)$ defined by $P(n, t, f)$ as in Eq. (1.1) is *correct* if it holds that

$$M(n, t, f) \models \varphi \tag{1.2}$$

for all values of parameters n, t and f that satisfy $RC(n, t, f)$.

In this thesis we address four challenges about correctness of distributed algorithms according to Definition (1.2):

C1 Check if an asynchronous system $M(n, t, f)$ is correct for a resilience condition RC, if φ is a **reachability** property;

C2 Check if an asynchronous system $M(n, t, f)$ is correct for an RC, if φ is a **safety or liveness** property that can be encoded in a fragment of linear temporal logic with operators F and G;

C3 If a process code is only partially defined, **synthesize** the full definition of $P(n, t, f)$, such that the asynchronous system $M(n, t, f)$ defined by $P(n, t, f)$ is correct for a given resilience condition and a formula in LTL as in $C2$;

C4 Check if a **randomized** asynchronous system $M_{\text{rand}}(n, t, f)$ satisfies an LTL formula as in $C2$, possibly with **probability 1**.

We successfully deal with these four challenges in Chapters 3–6. Moreover, we design an automated tool that either confirms correctness, or otherwise it generates concrete values of parameters, i.e., a fixed system, and an execution in it, that demonstrates violation of φ. The tool is called Byzantine Model Checker (ByMC [KW18]).

The dependence diagram of the four problems is depicted in Figure 1.2. Every solution to a challenge is based on the solution to the preceding challenge in the diagram.

1.3 State of the Art

Model checking. Verification is a general notion that includes different formal methods, such as theorem proving [BM83], abstract interpretation [CC77], and model

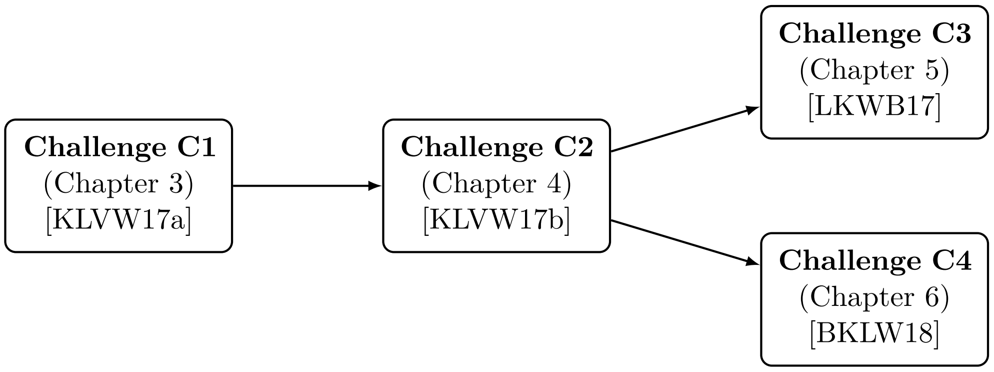


Figure 1.2: A schematic representation of the four parameterized model checking challenges addressed in this thesis.

checking [CHVB18]. We focus on *model checking*, that is in [CGP99] defined as an efficient search procedure used to determine if a specification is true on a transition system, that is, the transition system is *checked* to see whether it is a *model* of the specification [CGP99]. Formally, given a transition system M and a property φ, model checking is concerned with the question whether it holds that $M \models \varphi$.

Model checking has been introduced in the early eighties independently in [CE81] and [QS82], but it still receives well deserved attention [CGP99, BK08, GV08, CHVB18]. The state-transition system is a formalization of, for instance, a protocol or a circuit design[2]. A property is given in a *temporal logic*, that is an extension of propositional logic that allows us to describe the behavior of a system over time, e.g., an event eventually happens, or an event keeps appearing until another event happens, etc.

The early model checking techniques [CE81, QS82] searched through all system states in an exhaustive manner. Thus, they dealt with finite state systems, and they faced the problem of the combinatorial state explosion, which is the most evident for concurrent systems. Even if we have only a fixed finite number of processes, all possible interleavings form a massive global search space. The number of global states grows exponentially with the number of processes. If our goal is automated verification of huge databases or blockchains, we need more tuned techniques. We need to exploit the features of the transition system.

Effective methods that increase efficiency of model checking are abstraction [CGL94, GS97], binary decision diagrams [BCM$^+$90], satisfiability solvers [BCCZ99, Bra12], abstraction refinement using SMT solvers [CGJ$^+$03, BLR11].

[2]In the model checking literature, as well as in this thesis, we say that a transition system is a *model*, in the sense of formalizing (modeling) the protocol or the circuit. This should not be confused with a transition system being a *logical model* of a formula.

Parameterized model checking. When we fix values of n, t and f in a parameterized system $M(n,t,f)$, we typically get a finite state system. Conventional model checkers are able to verify correctness only for small instances of systems. A method that checks correctness of a system $M(n,t,f)$ according to the definition (1.2), that is, a method for checking whether the system satisfies the specification for all values of these parameters, is called *parameterized model checking.* As we consider an infinite family of systems, parameterized model checking is exceptionally hard. Moreover, it happens to be undecidable for many computational models, which is summarized in [BJK$^+$15].

A common method examines cutoffs [EN95, CTTV04, KKW10, AHH13, MSB17], that ensure that checking systems up to a specific size is necessary and sufficient for the correctness of the system for all sizes. This idea allows us to use existing non-parameterized techniques on small systems in order to check the parameterized problem. In other words, the existence of a cutoff for a system implies decidability of the paramaterized model checking problem for that system [BJK$^+$15, Prop. 3.5]. Nonetheless, in order to solve this problem, one needs to know (i) whether there is a cutoff, (ii) the size of the cutoff, and (iii) whether the cutoff is small enough to be checked in practice.

The first undecidability result for the parameterized model checking of concurrent systems was obtained in [AK86]. Their idea lies in reducing the parameterized model checking problem to the non-halting problem of Turing machines. Extensions of this idea could also be applied to uniform concurrent systems, that is, those where each of n finite-state processes is independent of the parameter n. Decidability of such systems depends on the means of communication, timing assumptions, as well as on the topology of the underlying communication graph.

For instance, undecidability of parameterized model checking problem for token rings where tokens contain information is presented in [Suz88]. Token passing systems communicate by passing a token along an edge of the underlying graph. For tokens that do not contain information, there exist decidability results of the parameterized model checking problem [EN95, EN03, CTTV04, AJKR14] for special fragments of temporal logic that do not contain the nexttime operator.

Imposing certain restrictions on the specifications [EK03, EFM99, GS92, AKR$^+$18], when focusing on the parameterized systems where processes communicate via pairwise rendezvous or by broadcasting messages, implies decidability for safety properties, as well as for liveness for pairwise rendezvous, and undecidability for liveness otherwise. In asynchronous shared-memory systems with one distinguished leader and any number of identical contributors, safety and liveness are decidable [DEGM15, EGM16]. Slight deviations from this setting lead to undecidability [Esp14].

Another natural technique for parameterized model checking is based on induction. It is requires network invariants [WL89, KM95], that is, the properties such that holds on a system with n processes, that it also hold on the analogous system with $n+1$ processes. Numerous extensions of this technique have been developed for parameterized systems defined by network grammars [BCG89, SG89, CGJ95, CGJ97, KZ10].

As described in the survey [BJK$^+$15], the state-of-the-art techniques for parameterized model checking [EN95, PXZ02, EK03, CTV08] do not focus on fault-tolerant distributed algorithms.

Fault-tolerant distributed algorithms. Design, implementation, and verification of distributed systems constitutes an active research area [vGBR16, KAB$^+$07, BDMS13, DHZ16, LBC16, PTP$^+$16]. Although distributed algorithms show complex behavior, and are difficult to understand for human engineers, there is only very limited tool support to catch logical errors in fault-tolerant distributed algorithms at design time.

One approach is to encode these algorithms in TLA+ [TLA], and use the TLC model checker to automatically find bugs in small instances, i.e., in distributed systems containing, e.g., three processes. Large distributed systems (e.g., clouds) need guarantees for *all* numbers of processes. These guarantees are typically given using hand-written mathematical proofs. In principle, these proofs could be encoded and machine-checked using the TLAPS proof system [CDLM10], PVS [LR93], Isabelle [CBM09], Coq [LBC16], Nuprl [RGBC15], or similar systems; but this requires human expertise that cannot be easily automated and repeated for other benchmarks.

Ensuring correctness of the implementation is an open challenge: As the implementations are done by hand [OO14, PTP$^+$16], the connection between the specification and the implementation is informal, such that there is no formal argument about the correctness of the implementation. There is a mature theory regarding mathematical proof methods, which found their way into formal frameworks like I/O Automata [Lyn96] and TLA+ [Lam02]. Recent approaches [WWP$^+$15, LBC16, HHK$^+$17, SWT18] provide tool support to establish correctness of implementations, by manually constructing proofs with an interactive theorem prover. Although, if successful, this approach provides a machine-checkable proof [CDLM10, BBJ$^+$16a], it requires considerable manual efforts from the user. A logic for distributed consensus algorithms in the HO Model [CS09] was introduced in [DHV$^+$14]. It allows one to automatically check the invariants (for safety) and ranking functions (for liveness), that is, the manual effort is reduced to finding right invariants and ranking functions. Model checking of distributed algorithms promises a higher degree of automation. For consensus algorithms in the HO Model, the results of [MSB17] reduce the verification to checking small systems of five or seven processes.

Another methodology that has a better degree of automation, has been introduced in [PMP$^+$16], supported by the interactive verification tool Ivy [MP18]. It has been extended in [PHL$^+$18, PHM$^+$18] in order to check both safety and liveness of a large class of distributed algorithms that can be expresses in the effectively-propositional fragment (EPR) of first order logic. Expressing algorithms in EPR is not necessarily straightforward, as the quantifier alternation graph might contain cycles, whose elimination requires human expertise. The user specifies the distributed algorithm and suggests an invariant, and receives guidance in the form of a counterexample depicted graphically. By inspecting the counterexample, that is typically small as EPR has the finite model property, the user suggests another invariant and proceeds. This technique has been successfully used for verification of several variations of Paxos.

Recently, parameterized model checking of FTDAs in the synchronous setting has been achieved in [ARS$^+$18] for safety and liveness using abstraction, and in [SKWZ18] using bounded model checking only for safety. More generally, in the parameterized case, going from safety to liveness is not straightforward, as exemplified by [FKP16]. There are systems where safety is decidable and liveness is not [EFM99].

Threshold-based FTDAs. As a starting point for the work of this thesis, we have used the parameterized model checking technique from [KVW14], that is based on bounded model checking. Namely, the authors use acceleration in order to find a bounded diameter of the system, and check all executions up to the computed length. This can only be applied for reachability properties, and moreover, enumerating all executions of a specific length is in general inefficient. This work is an adaptation of the approach based on data and counter abstraction, used in [JKS$^+$13a, Gme15]. This was the first successful parameterized model checking technique for both safety and liveness properties of fault-tolerant distributed algorithms. Although it allows checking of any formula, this method proved to be impractical for large benchmarks, as the abstraction introduced spurious counterexamples that had to be manually examined.

1.4 Methodological Approach

Given a sequential piece of code, once the input is fixed, the control flow of the code with the evaluation of variables defines *a single* execution (if we ignore a lot of details in modern compiler and processor design, such as, code optimization, caching, etc.). In sharp contrast, a distributed system consists of multiple processes each with its local control flow. In asynchronous systems, processes run independently, so that already all possible interleavings of steps of the distributed processes induce a typically huge execution space rather than a single execution. Furthermore, we are interested in the case where these processes communicate by message passing, without restricting delays of their messages. This additionally increases the execution space.

As already observed by Lamport [Lam78], distributed computations thus induce a partial order — the so-called happens before relation — of events in a distributed system. Roughly speaking, the event of sending a message m happens before the event of receiving m, and for each process p, local events at p are ordered according to the temporal order of their occurrences. The happens before relation is the transitive closure of the send-receive relation for all messages and all processes, and the local order of events for each process. As a result, if in an execution events e_1 and e_2 happend directly one after the other at two distinct processes p and q, respectively, e_1 and e_2 may still be independent (not ordered according to happened before). That is, neither the local control flows, nor the messages impose an order of e_1 and e_2 so that swapping e_1 and e_2 leads to a different execution, which (i) entails the same happens before relation, and (ii) is locally indistinguishable for processes p and q (and all other processes in the system).

The enormously large execution spaces of asynchronous algorithms make understanding and reasoning about the executions hard, both for humans and computers. An important idea to handle this complexity is to structure the reasoning along the induced partial orders. Due to the mentioned partial order and indistinguishability arguments, the happens before relation can be understood as an equivalence relation between executions: all executions that have the same happens before relation over their events can be seen as falling into the same class. For many interesting specifications, it is sufficient to check a representative execution from each class. Here we distinguish — very roughly — two approaches. In *partial-order reduction*, while searching the execution space, executions that are "similar" to ones searched before are pruned [God90, Val91, Pel93]. In *reduction* one proves a priori that every execution can be represented by an execution of specific form [Lip75, EF82]. Then verification procedures only need to consider executions of these specific forms.

To the best of our knowledge, Lipton [Lip75] was the first to highlight reduction as a proof method for concurrent systems. In his theory, processes execute sequences of statements, for instance, one process may be the sequence of statements A, B, C, and another process may be the sequence X, Y. Then, concurrent executions are interleaved sequences of statements. For example, A, X, B, Y, C is an execution, as well as X, A, B, C, Y. In the latter execution, the sequence A, B, C is said to be executed atomically.

Lipton considered the classic semaphore operation $P(s)$ and $V(s)$, for semaphore s. Then if p's code is $P(s), B, V(s)$, he proves that all executions can be reduced to ones where $P(s), B, V(s)$ occurs as uninterupted (atomic) block: Intuitively, if an execution is interleaved with an event A' at a different process p', that is, $P(s), A', B, V(s)$, then Lipton proves that $P(s)$ can always be moved to the right, that is, $A', P(s), B, V(s)$ is also an execution within the mentioned block. Such an operation is called a *right mover*. Similarly, all V operations are called *left movers*. Thus, Lipton's reduction consists in identifying large blocks of process code between a P operation and its matching V operation that can be "moved" together. Then for verification (of reachability properties) it is sufficient to consider the executions where these blocks are executed atomically.

Reduction techniques for threshold-based FTDAs. In this thesis, we focus on the class of fault-tolerant distributed algorithms described in Section 1.1, and the four parameterized model checking challenges from Section 1.2. The dependence diagram of the four problems is depicted in Figure 1.2. In all the four points we exploit the structure of the system given by a threshold automaton, and develop new reduction techniques that fit to such systems. The dependence diagram of the techniques used for solving the four challenges is given in Figure 1.3. Every reduction technique from the figure is based on the technique from the preceding point in the diagram.

The core technique is called PARA^2. The acronym stands for *PARAmeterized PAth Reduction with Acceleration*. We analyze a finite path (i.e., an execution) of a system, and

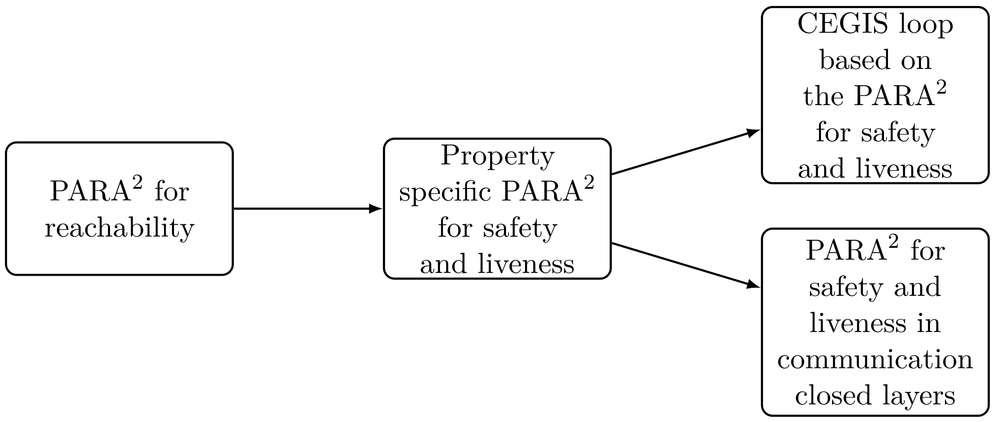

PARA^2 for reachability
Property specific PARA^2 for safety and liveness
CEGIS loop based on the PARA^2 for safety and liveness
PARA^2 for safety and liveness in communication closed layers

Figure 1.3: A schematic representation of the reduction techniques used for solving the challenges from Section 1.2, following the diagram from Figure 1.2.

use a specific type of reduction and acceleration in order to form a short path with the same characteristics. For example, we analyze the relation between statements from Figure 1.1 like "`send ECHO`" and "`received ECHO from at least t+1 distinct processes`" in order to determine which statements are movable. Moving actions in a specific fashion allows us to generate other atomic steps. For instance, if m processes perform event A one after the other, that is, the execution is $A, A, \dots, A$, this can be represented by a single (accelerated) transition A^m. As a result, moving the As together and accelerating them, leads to "shorter" executions. The goal of every PARA^2 technique is to obtain the so-called *short counterexample property*, stating that if there is a counterexample to a formula in a system, then there exists also a counterexample of a bounded length. This allows us to consider only paths of the obtained length. Moreover, the way we use acceleration makes the length of a path independent of the number of processes in the system. Therefore, the obtained bound can be used efficiently for any system size.

In the following, we briefly explain each of the PARA^2 techniques from Figure 1.3, used in this thesis.

PARA^2 for reachability. Inspired by Lipton's method, we first introduce a reduction technique PARA^2 for reachability properties. This means that, when analyzing a path, after applying this reduction technique, the goal is to obtain a short path that satisfies the same reachability properties. In other words, we want to obtain a short path that reaches the same global state as the original path, when the initial state is the same. We exploit the properties of threshold guards to show which transitions can be reordered and accelerated. In Chapter 3 we prove that this type of PARA^2 preserves reachability properties.

Property specific PARA^2 for safety and liveness. In contrast to the goal of only reaching one state, more sophisticated properties are, for instance, expected to hold in every state along a path. Our first PARA^2 technique only ensures that the original path and its short version have the same final state, but it does not guarantee anything about the states visited by the two paths. Therefore, in Chapter 4 we introduce a sophisticated PARA^2 technique that depends both on the properties of threshold guards, as well as on the formula we want to preserve. The bounds on the length of the path obtained by this technique are two times, or in rare cased three times larger than the one from the previous method, but the acceleration still makes this bound independent of the number of processes.

CEGIS loop based on the PARA^2 for safety and liveness. For synthesizing algorithms correct by construction in Chapter 5, we use the CEGIS technique [ABJ$^+$13] that, very roughly explained, gives a guess of a solution and then it checks if the guess is indeed a solution, and continues by repeating the procedure. The checking part is based on the verification technique provided by the PARA^2 technique for safety and liveness.

PARA^2 for safety and liveness in communication closed layers. In randomized distributed algorithms processes repeat the execution of their process code unboundedly many times, in their own speed. When analyzing asynchronous composition of unbounded number of such rounds, we need to raise the PARA^2 technique to the next level. Namely, we advocate for a classic reduction by Elrad and Francez [EF82], which they originally formulated in the context of CSP [Hoa78] (Communicating Sequential Processes). Consider a parallel composition of processes P_i, for $1 \leq i \leq n$, that is, $S = P_1 \parallel P_2 \parallel \cdots \parallel P_n$. Further assume that each process P_i is a sequential composition of layers $L_i^1; L_i^2; \ldots; L_i^k$. Then they assume the following property: if for two processes i and j, the layers L_i^a and L_j^b communicate (have a synchronized event in CSP), then $a = b$. In other words, layers communicate only with layers of the same number, that is, we say they are *communication-closed*. If we consider the parallel compositions of layers $L^k = L_1^k \parallel L_2^k \parallel \cdots \parallel L_n^k$, then the central result — proved with a reduction argument — is that instead of analyzing S, it is sufficient to analyze the sequential composition of layers $S^\ell = L^1; L^2; \ldots L^k$. Observe that S^ℓ has considerably fewer interleavings than S. For instance, in S events of layer 2 at process p, L_p^2, might occur before events of layer 1 at process q (that is, L_q^1), while in S^ℓ this cannot be the case.

Inspired by the work of Elrad and Francez, and in combination with our existing PARA^2 techniques, in Chapter 6 we introduce the PARA^2 technique for safety and liveness in communication closed layers. It allows us to order the layers, and then to use the PARA^2 technique within each layer. Note that this technique does not yield short executions, as we have unboundedly many "short" layers, but we further prove that we can single out one representative layer.

1.5 Structure of the Thesis

In Chapter 2 we introduce the framework for modeling threshold-based fault-tolerant distributed algorithms. That includes threshold automata that are used to model distributed algorithms, counter systems as their semantics, and the fragment $\mathsf{ELTL}_{\mathsf{FT}}$ of temporal logic for specifying temporal properties of distributed systems. The definition of $\mathsf{ELTL}_{\mathsf{FT}}$ is our contribution, whereas counter systems and threshold automata were introduced earlier in [KVW14].

Each of the following Chapters 3–6 is devoted to one of the four challenges from Section 1.2, and thus also to the four papers [KLVW17a, KLVW17b, LKWB17, BKLW18]. This follows the diagram from Figure 1.2. If the reader is interested in only one of the chapters, we suggest also reading the preceding chapters from the diagram.

Where it makes sense for demonstrating our ideas, we start with a simple illustration of the technique, followed by the definitions of the notions required for that chapter only. The techniques are precisely explained and accompanied by examples. We often leave the technical proofs for the second to last sections of chapters. Moreover, every technique is supported by experimental evaluation.

Chapter 7 summarizes these results and offers possible successive research directions.

CHAPTER 2

Threshold Automata

We focus on one particular class of distributed algorithms, namely, threshold-based algorithms. This means that for performing an action, a process has to receive enough messages from its peers. In other words, in our setting processes communicate by message passing, and each process keeps track of the number of messages it has received so far, and compares it with a given threshold. Once the number of received messages is at least as large as the threshold, the process is allowed to perform each action associated to this threshold.

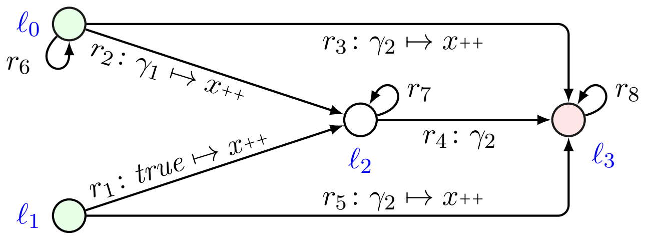
Thresholds are given in the form of a linear combination of parameters. For example, if a system consists of n processes, and if a process needs to wait for messages from a majority of its peers, the corresponding threshold is $n/2$.

Using the properties of these expressions, we present techniques for verification of threshold-based distributed algorithms in Chapter 3, Chapter 4, and Chapter 6, and for synthesizing the expressions themselves in order to obtain correct algorithms in Chapter 5.

In this chapter we introduce the most important notions, necessary and common for all the following chapters. This includes threshold automata in Section 2.1, used for formalizing threshold-based distributed algorithms, as well as counter systems in Section 2.2, used to model semantics of threshold automata. Definitions specific for a single chapter, like a sketch threshold automaton in Chapter 5, or a probabilistic threshold automaton in Chapter 6, are introduced in the respective chapters.

2.1 Modeling Threshold-based Distributed Algorithms

The notion of a *threshold automaton* (TA) is introduced in [KVW14], and it is used to describe a local control flow of a single process in a concurrent system. The TA that corresponds to the pseudo code from Figure 1.1 is given in Figure 2.1. The threshold



ℓ_0
r_6
$r_2\colon \gamma_1 \mapsto x{+}{+}$
$r_3\colon \gamma_2 \mapsto x{+}{+}$
r_7
r_8
$r_4\colon \gamma_2$
ℓ_2
ℓ_3
$r_1\colon true \mapsto x{+}{+}$
$r_5\colon \gamma_2 \mapsto x{+}{+}$
ℓ_1

Figure 2.1: The threshold automaton corresponding to the algorithm from Figure 1.1 with $\gamma_1\colon x \geq (t+1)-f$ and $\gamma_2\colon x \geq (n-t)-f$ over parameters n, t, and f, representing the number of processes, the upper bound on the faulty processes (used in the code), and the actual number of faulty processes. The negative number $-f$ in the threshold is used to model the environment, and captures that at most f of the received messages may have been sent by faulty processes.

automaton represents the local control flow of a single process, where edges represent local transitions that are labeled with $\varphi \mapsto \mathsf{act}$: Expression φ is a threshold guard and the action act may increment a shared variable.

Example 2.1. Here we explain the relation between the TA from Figure 2.1 and the pseudo code from Figure 1.1: if for a process i we have $myval_i = 1$, this corresponds to the initial local state ℓ_1 of the process, while otherwise the process starts in ℓ_0. The local state ℓ_2 in Figure 2.1 captures that the process has sent ECHO and $accept_i$ evaluates to false, while ℓ_3 captures that the process has sent ECHO and $accept_i$ evaluates to true. The syntax of Figure 1.1, although checking how many messages of some type are received, hides bookkeeping details and the environment, e.g., message buffers. For our verification technique, we need to make such issues explicit: The shared variable x stores the number of correct processes that have sent ECHO. Initially no process has sent any messages, and thus $x = 0$. Incrementing x models that ECHO is sent when the transition is taken. Then, execution of Line 4 corresponds to the transition r_1. Executing Line 9 is captured by r_2: the check whether $t+1$ messages are received is captured by the fact that r_2 has the guard γ_1, that is, $x \geq (t+1) - f$. Intuitively, this guard checks whether sufficiently many processes have sent ECHO (i.e., increased x), and takes into account that at most f messages may have been sent by faulty processes. Namely, if we observe the guard in the equivalent form $x+f \geq t+1$, then we notice that it evaluates to true when the total number of received ECHO messages from correct processes (x) and potentially received messages from faulty processes (at most f), is at least $t+1$, which corresponds to the guard of Line 9. Transition r_4 corresponds to Line 13, r_3 captures that Line 4 and Line 13 are performed in one protocol step, and r_5 captures Line 9 and Line 13. $\triangleleft$

While the example shows that the pseudo code and a TA are quite close, it should be noted that in reality, things are slightly more involved. For instance, in the pseudo code threshold guards are evaluated locally, which requires local counters in the implementation, that are abstracted in the TA. Discussions on data abstraction and automated generation of TAs from code in parametric Promela can be found in [KVW16].

We use threshold automata in the theoretical work of this thesis, as well as for an internal representation in the ByMC tool following this work. Let us now formally define them.

Threshold Automaton. A threshold automaton is a tuple $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$, where the components represent sets of local states, variables, rules defining the state changes, and resilience conditions, respectively. Below we define each of the sets.

States. The set of all *local states* (or, *locations*) is denoted by $\mathcal{L}$. This is a nonempty finite set, and for simplicity we use the convention that $\mathcal{L} = \{1, \ldots, |\mathcal{L}|\}$. It contains the set of distinct local states, called *initial states* (or, *initial locations*). The set of all initial states is denoted by $\mathcal{I} \subseteq \mathcal{L}$.

Variables and resilience conditions. The set of all variables $\mathcal{V}$ is partitioned in two sets, denoted by Γ and Π.

The set Γ is the finite nonempty set of *shared variables* over $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$. The finite nonempty set of all *parameter variables*, ranging over $\mathbb{N}_0$, is denoted by Π. The *resilience condition RC* is a predicate over $\mathbb{N}_0^{|\Pi|}$, often given as a formula in linear integer arithmetic, e.g., $n > 3t$. The set of all *admissible parameters* is denoted by $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}_0^{|\Pi|} : RC(\mathbf{p})\}$.

Example 2.2. The admissible parameters and resilience conditions are motivated by fault-tolerant distributed algorithms: Let n be the number of processes, t be the assumed number of faulty processes, and in a run, f be the actual number of faults. For these parameters, the famous result by Pease, Shostak and Lamport [PSL80] states that agreement can be solved iff the resilience condition $n > 3t \wedge t \geq f \geq 0$ is satisfied. Given such constraints, the set $\mathbf{P}_{RC}$ is infinite, and in Section 2.2 we will see that this results in an infinite state system. $\triangleleft$

Threshold guards. The key ingredient of threshold automata are threshold guards. We have already mentioned their role in distributed algorithms and here we precisely define them.

Definition 2.1. *A* threshold guard *(or just a* guard*) is an inequality of one of the following two forms:*

(R) $x \geq a_0 + a_1 \cdot p_1 + \ldots + a_{|\Pi|} \cdot p_{|\Pi|}$*, or*

(F) $x < a_0 + a_1 \cdot p_1 + \ldots + a_{|\Pi|} \cdot p_{|\Pi|}$*,*

where x is a shared variable from Γ, $p_1, \ldots, p_{|\Pi|} \in \Pi$ are parameter variables, and $a_0, a_1, \ldots, a_{|\Pi|} \in \mathbb{Q}$ are rational coefficients. The set of all threshold guards of type (R) is denoted by Φ^{rise}, and the set of all type (F) by Φ^{fall}.

Every guard compares a shared variable with an expression that is called *threshold*. Sometimes we call guards of type (R) *rissing guards*, and those of type (F) we call *falling guards*. In Section 2.2 we discusse the motivation for these names.

Note that guards are not necessarily given in linear integer arithmetic because of the rational coefficients, but since there are only finitely many parameters, every guard can easily be transformed into an inequality in linear integer arithmetic.

Rules. A rule is a conditional transition between two local states that may update shared variables. It is formally defined as a tuple $(\mathit{from}, \mathit{to}, \varphi^{\text{rise}}, \varphi^{\text{fall}}, \mathbf{u})$, where *from* and *to* are local states from $\mathcal{L}$, formulas φ^{rise} and φ^{fall} are conjunctions of the guards from the sets Φ^{rise} and Φ^{fall}, respectively, and $\mathbf{u} \in \mathbb{N}_0^{|\Gamma|}$ is an update vector. We explain the semantics of a rule in the following section, but intuitively one can understand an execution of a rule as a transition of a process from the local state *from* to *to*, under the condition that both φ^{rise} and φ^{fall} evaluate to true, and the shared variables are updated by adding the vector $\mathbf{u}$ to the vector of all shared variables. The set of all rules is denoted by $\mathcal{R}$.

Example 2.3. An example of a threshold automaton is depicted in Figure 2.1. There are four local states, $\mathcal{L} = \{\ell_0, \ell_1, \ell_2, \ell_3\}$, among which ℓ_0 and ℓ_1 are initial states. There is one shared variable x and eight rules. For example, as $\gamma_2 \in \Phi^{\text{rise}}$, rule r_3 is formally defined as $(\ell_0, \ell_3, \gamma_2, \top, 1)$. Note that the first two components encode the edge, and the last three encode the edge label (two different types of guards, and the update vector). Thus, a rule corresponds to a (guarded) statement from Figure 1.1 (or combined statements as discussed in Example 2.1) $\triangleleft$

Threshold automata for fault-tolerant distributed algorithms. The above definition of TAs is quite general. It allows loops, increase of shared variables in loops, etc. As has been observed in [KVW17], if one does not restrict increases on shared variables, the resulting systems may produce runs that visit infinitely many states, and there is little hope for a complete verification method. Hence, we analyzed the TAs of the benchmarks [CT96, ST87b, BT85, MMPR03, Ray97, Gue02, DS06, BGMR01, SvR08]. We observed that some states have self-loops (corresponding to busy-waiting for messages to arrive) and in the case of failure detector based algorithms [Ray97] there are loops that consist of at most two rules. None of the rules in loops increase shared variables. In our theory, we allow more general TAs than actually found in the benchmarks. In more detail, we make the following assumption:

As in [KVW17], we assume that if a rule r is in a loop, then $r.\mathbf{u} = \mathbf{0}$. Automata with this property are called *canonical* in [KVW17]. Different types of canonical TAs have been discussed in [KKW18], where the focus was on exploring the existence of a bounded diameter. In fact, it is proven in [KKW18] that TAs that are not canonical do not have bounded diameter.

In addition, in Chapter 4 we use the restriction that all the cycles of a TA are simple, i.e., between any two locations in a cycle there exists exactly one node-disjoint directed path (nodes in cycles may have self-loops). The technique from Chapter 3 for verifying reachability properties does not rely on this assumption, and thus it can be applied to threshold automata with more complex cycles (but still without updates of shared variables). Still, this restriction is crucial for checking safety and liveness properties from Chapter 4, and therefore also for the techniques from Chapter 5 and Chapter 6 that extend safety and liveness verification.

A possibility to relax this restriction for verification of safety and liveness properties, similarly as for reachability properties, remains a conjecture.

Example 2.4. In the TA from Figure 2.1 we use the shared variable x as the number of correct processes that have sent a message. One easily observes that the rules that update x do not belong to loops. Indeed, all the benchmarks [CT96, ST87b, BT85, MMPR03, Ray97, Gue02, DS06, BGMR01, SvR08] share this structure. This is because at the algorithmic level, all these algorithms are based on the reliable communication assumption (no message loss and no spurious message generation/duplication), and not much is gained by resending the same message. In these algorithms a process checks whether sufficiently many processes (e.g., a majority) have sent a message to signal that they are in some specific local state. Consequently, a receiver would ignore duplicate messages from the same sender. In our analysis we exploit this characteristic of distributed algorithms with threshold guards, and make the corresponding assumption that processes do not send (i.e., increase x) from within a loop. Similarly, as a process cannot make the sending of a message undone, we assume that shared variables are never decreased. So, while we need these assumptions to derive our results, they are justified by our application domain. $\triangleleft$

2.2 Parameterized Counter Systems

A threshold automaton models a single process. Now the question arises how we define the composition of multiple processes that will result in a distributed system. It is well-known that the system state of a specific distributed or concurrent system can be represented as a counter system [Lub84, PXZ02, AGP16, KVW17]: instead of recording for some local state ℓ, which processes are in ℓ, we are only interested in *how many processes are in* ℓ. In this way, we can efficiently encode transition systems in SMT with linear integer arithmetics. Therefore, we formalize the semantics of the threshold automata by counter systems.

Fix a threshold automaton TA, admissible parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, and a function (expressible as linear combination of parameters) $N \colon \mathbf{P}_{RC} \to \mathbb{N}_0$ that determines the number of modeled processes. For example, if we want to tolerate Byzantine faults, we model only $N(n,t,f) = n - f$ (correct) processes; if we have crash faults, we model all $N(n,t,f) = n$ processes.

A counter system $\mathsf{Sys}(\mathsf{TA})$ is defined as a transition system (Σ, I, R), with configurations Σ and I and transition relation R defined below.

Configurations. Every configuration keeps track of how many processes are in each local state, as well as of values of shared and parameter variables.

Definition 2.2. *A configuration* $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$ *consists of a vector of* counter values $\sigma.\boldsymbol{\kappa} \in \mathbb{N}_0^{|\mathcal{L}|}$*, a vector of* shared variable values $\sigma.\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$*, and a vector of* parameter values $\sigma.\mathbf{p} = \mathbf{p}$*. The set Σ contains all configurations. The initial configurations are in set I, and each initial configuration σ satisfies $\sigma.\mathbf{g} = \mathbf{0}$, $\sum_{i \in \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = N(\mathbf{p})$, and $\sum_{i \notin \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = 0$.*

Example 2.5. The safety property from Example 2.2, refers to an initial configuration that satisfies resilience condition $n > 3t \wedge t \geq f \geq 0$, e.g., $\sigma.\mathbf{p} = (4, 1, 0)$, with resilience condition $4 > 3 \cdot 1 \wedge 1 \geq 0 \geq 0$. In our encodings we typically have that N is the function $(n, t, f) \mapsto n - f$. Further, $\sigma.\boldsymbol{\kappa}[\ell_0] = N(\mathbf{p}) = n - f = 4$ and $\sigma.\boldsymbol{\kappa}[\ell_i] = 0$, for $\ell_i \in \mathcal{L} \setminus \{\ell_0\}$, and the shared variable $\sigma.\mathbf{g} = 0$. ◁

Transitions and transition relation. A *transition* is a pair $t = (\mathit{rule}, \mathit{factor})$ of a rule and a non-negative integer called the *acceleration factor*. For $t = (\mathit{rule}, \mathit{factor})$ we write $t.\mathbf{u}$ for $\mathit{rule}.\mathbf{u}$, etc. A transition t is *unlocked* in σ if

$$\forall k \in \{0, \dots, t.\mathit{factor} - 1\}.\ (\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + k \cdot t.\mathbf{u}, \sigma.\mathbf{p}) \models t.\varphi^{\text{rise}} \wedge t.\varphi^{\text{fall}}.$$

A transition t is *applicable (or enabled)* in σ, if it is unlocked, and $\sigma.\boldsymbol{\kappa}[t.\mathit{from}] \geq t.\mathit{factor}$, or $t.\mathit{factor} = 0$.

To simplify notation, sometimes we write $\mathit{rule}^{\mathit{factor}}$ instead of $(\mathit{rule}, \mathit{factor})$.

Example 2.6. This notion of applicability contains acceleration and is central for our approach. Intuitively, the value of the factor corresponds to how many times the rule is executed by different processes. In this way, we can subsume steps by an arbitrary number of processes into one transition. Consider Figure 2.1. If for some k, k processes are in location ℓ_1, then in classic modeling it takes k transitions to move these processes one-by-one to ℓ_2. With acceleration, however, these k processes can be moved to ℓ_2 in one step, independently of k. In this way, the bounds we compute will be independent of the parameter values. However, assuming x to be a shared variable and f being a parameter that captures the number of faults, our (crash-tolerant) benchmarks include rules like "$x < f \mapsto x{+}{+}$" for local transition to a special "crashed" state. The above definition ensures that at most $f - x$ of these transitions are accelerated into one transition (whose factor thus is at most $f - x$). This precise treatment of threshold guards is crucial for fault-tolerant distributed algorithms. For instance, the central contribution of Chapter 4 is to show how acceleration can be used to shorten schedules while maintaining specific temporal logic properties. ◁

Definition 2.3. *The configuration σ' is the result of applying the applicable transition t to σ, if*

1. $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + t.\mathit{factor} \cdot t.\mathbf{u}$
2. $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$
3. *if $t.\mathit{from} \neq t.\mathit{to}$ then*
   - $\sigma'.\boldsymbol{\kappa}[t.\mathit{from}] = \sigma.\boldsymbol{\kappa}[t.\mathit{from}] - t.\mathit{factor}$,
   - $\sigma'.\boldsymbol{\kappa}[t.\mathit{to}] = \sigma.\boldsymbol{\kappa}[t.\mathit{to}] + t.\mathit{factor}$, *and*
   - $\forall \ell \in \mathcal{L} \setminus \{t.\mathit{from}, t.\mathit{to}\}.\ \sigma'.\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell]$.
4. *if $t.\mathit{from} = t.\mathit{to}$ then $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa}$.*

In this case we use the notation $\sigma' = t(\sigma)$.

The transition relation R is defined as follows: Transition (σ, σ') belongs to R iff there is a rule $r \in \mathcal{R}$ and a factor $k \in \mathbb{N}_0$ such that $\sigma' = t(\sigma)$ for $t = (r, k)$.

Since every threshold is a linear combination of parameters, its value depends only on parameter values, and thus by Definition 2.3 it does not change in a run. Therefore, the evaluation of a guard in a run depends only on shared variable values. As updates can only increase the values of shared variables, the shape of threshold guards suggests that once a rising guard becomes true in a run, it remains true, and similarly, once a falling guard becomes false in a run, it remains false. This monotonic behavior is a crucial property and it has been a motivation for the names of rising and falling guards. We recall this property formally in the following proposition from [KVW17, Proposition 7]:

Proposition 2.1 (Monotonicity of guards)**.** *For all configurations σ, all rules r, and all transitions t applicable to σ, the following holds:*

1. *If $\sigma \models r.\varphi^{\mathrm{rise}}$ then $t(\sigma) \models r.\varphi^{\mathrm{rise}}$*
2. *If $t(\sigma) \not\models r.\varphi^{\mathrm{rise}}$ then $\sigma \not\models r.\varphi^{\mathrm{rise}}$*
3. *If $\sigma \not\models r.\varphi^{\mathrm{fall}}$ then $t(\sigma) \not\models r.\varphi^{\mathrm{fall}}$*
4. *If $t(\sigma) \models r.\varphi^{\mathrm{fall}}$ then $\sigma \models r.\varphi^{\mathrm{fall}}$*

Example 2.7. Let us again consider Figure 2.1 with $n = 4$, $t = 1$, and $f = 1$. We consider the initial configuration σ_0 such that $\sigma_0.\boldsymbol{\kappa}[\ell_1] = n - f = 3$ and $\sigma_0.\boldsymbol{\kappa}[\ell_i] = 0$, for $\ell_i \in \mathcal{L} \setminus \{\ell_0\}$. The guard of rule r_5, $\gamma_2\colon x \geq (n-t) - f = 2$, initially evaluates to false because $x = 0$. The guard of rule r_1 is true, and thus any transition (r_1, factor) is unlocked, but as $\sigma_0.\boldsymbol{\kappa}[\ell_1] = 3$, only for $0 \leq \mathit{factor} \leq 3$, transitions (r_1, factor) are applicable to σ_0. If the transition $(r_1, 2)$ is applied to the initial configuration, we obtain a configuration σ_1 where $x = 2$, and therefore, γ_2 evaluates to true in σ_1. Then r_5 is unlocked and the transitions $(r_5, 1)$ and $(r_5, 0)$ are applicable to σ_1 as $\sigma_1.\boldsymbol{\kappa}[\ell_1] = 1$. Since $\gamma_2 \in \Phi^{\mathrm{rise}}$, once it becomes true, it remains true. $\triangleleft$

Schedules and paths. A *schedule* is a (finite or infinite) sequence of transitions. For a schedule τ and an index $i : 1 \le i \le |\tau|$, by $\tau[i]$ we denote the ith transition of τ, and by τ^i we denote the prefix $\tau[1], \dots, \tau[i]$ of τ. A schedule $\tau = t_1, \dots, t_m$ is *applicable* to configuration σ_0, if there is a sequence of configurations $\sigma_1, \dots, \sigma_m$ with $\sigma_i = t_i(\sigma_{i-1})$ for $1 \le i \le m$. A schedule $t_1, \dots, t_m$ where $t_i.\mathit{factor} = 1$ for $0 < i \le m$ is called *conventional.* If there is a $t_i.\mathit{factor} > 1$, then a schedule is *accelerated.* By $\tau \cdot \tau'$ we denote the concatenation of two schedules τ and τ'. Similarly we define applicability of infinite schedules, and conventional and accelerated infinite schedules.

To reason about temporal logic properties, we need to reason about the configurations that are "visited" by a schedule. For that we introduce paths.

A finite or infinite sequence $\sigma_0, t_1, \sigma_1, \dots, \sigma_{k-1}, t_k, \sigma_k, \dots$ of alternating configurations and transitions is called a *path*, if for every transition t_i, $i \in \mathbb{N}$, in the sequence, holds that t_i is enabled in σ_{i-1}, and $\sigma_i = t_i(\sigma_{i-1})$. For a configuration σ_0 and a finite schedule τ applicable to σ_0, by $\mathsf{path}(\sigma_0, \tau)$ we denote $\sigma_0, t_1, \sigma_1, \dots, t_{|\tau|}, \sigma_{|\tau|}$ with $\sigma_i = t_i(\sigma_{i-1})$, for $1 \le i \le |\tau|$. Similarly, if τ is an infinite schedule applicable to σ_0, then $\mathsf{path}(\sigma_0, \tau)$ represents an infinite sequence $\sigma_0, t_1, \sigma_1, \dots, t_k, \sigma_k, \dots$ where $\sigma_i = t_i(\sigma_{i-1})$, for all $i > 0$. Given a path $\mathsf{path}(\sigma, \tau)$, the set of all configurations in the path is denoted by $\mathsf{Cfgs}(\sigma, \tau)$.

Due to the resilience conditions and admissible parameters, our counter systems are in general infinite state. The following proposition establishes an important property for verification.

Proposition 2.2. *Every (finite or infinite) path visits finitely many configurations.*

Proof. By Definition 2.3(3), if a transition t is applied to a configuration σ, then the sum of the counters remains unchanged, that is, $\sum_{\ell \in \mathcal{L}} \sigma.\boldsymbol{\kappa}[\ell] = \sum_{\ell \in \mathcal{L}} t(\sigma).\boldsymbol{\kappa}[\ell]$. By repeating this argument, the sum of the counters remains stable in a path. By Definition 2.3(2) the parameter values also remain stable in a path.

By Definition 2.3(1), it remains to show that in each path eventually the shared variable $\mathbf{g}$ stop increasing. Let us fix a rule $r = (\mathit{from}, \mathit{to}, \varphi^{\mathrm{rise}}, \varphi^{\mathrm{fall}}, \mathbf{u})$ that increases $\mathbf{g}$. By the definition of a transition, applying some transition (r, factor) decreases $\boldsymbol{\kappa}[r.\mathit{from}]$ by factor. As by assumption on TAs, r is not in a cycle, $\boldsymbol{\kappa}[r.\mathit{from}]$ is increased only finitely often, namely, at most $N(\mathbf{p})$ times. As there are only finitely many rules in a TA, the proposition follows. □

2.3 Fault-Tolerant Temporal Logic

In order to formalize a reachability property, we only need to describe a global state, that can easily be expressed as a Boolean formula over atomic propositions[1]

$$AP = \{\boldsymbol{\kappa}[\ell] = 0 \mid \ell \in \mathcal{L}\}.$$

[1] As a counter of a location represents a number of processes in that location, and thus is a non-negative natural number, the negation of $\boldsymbol{\kappa}[\ell] = 0$ becomes $\boldsymbol{\kappa}[\ell] > 0$.

Formalizing safety and liveness specifications requires reasoning about possibly infinite paths, and therefore also more involved temporal formulas. Moreover, as the techniques introduced in Chapters 3 and 4 check if a system can violate specifications, we focus on negations of specifications. We observe that such formulas use a simple subset of linear temporal logic that contains only the temporal operators $\mathbf{F}$ and $\mathbf{G}$.

Example 2.8. Consider the liveness property called Correctness from Section 1.1 formulated in the spirit of the threshold automaton from Figure 2.1:

$$\mathbf{G}\,\mathbf{F}\,\psi_{\text{fair}} \rightarrow (\boldsymbol{\kappa}[\ell_0] = 0 \rightarrow \mathbf{F}\,\boldsymbol{\kappa}[\ell_3] \neq 0). \tag{2.1}$$

Formula ψ_{fair} expresses the reliable communication assumption of distributed algorithms [FLP85]. In this example, this is formally expressed as follows:

$$\psi_{\text{fair}} \equiv \boldsymbol{\kappa}[\ell_1] = 0 \wedge (x \geq t+1 \rightarrow \boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_1] = 0) \wedge (x \geq n-t \rightarrow \boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_2] = 0).$$

Intuitively, $\mathbf{G}\,\mathbf{F}\,\psi_{\text{fair}}$ means that all processes in ℓ_1 should eventually leave this state, and if sufficiently many messages of type x are sent (γ_1 or γ_2 holds true), then all processes eventually receive them. If they do so, they have to eventually fire rules r_1, r_2, r_3, or r_4 and thus leave locations ℓ_0, ℓ_1, and ℓ_2.

Our approach is based on possible shapes of *counterexamples*. Therefore, we consider the negation of the specification (2.1), that is, $\mathbf{G}\,\mathbf{F}\,\psi_{\text{fair}} \wedge \boldsymbol{\kappa}[\ell_0] = 0 \wedge \mathbf{G}\,\boldsymbol{\kappa}[\ell_3] = 0$. In the following we define the logic that can express such counterexamples. $\triangleleft$

The fragment of LTL limited to $\mathbf{F}$ and $\mathbf{G}$ was studied in [EVW02, KOS+11]. We further restrict it to the logic that we call *Fault-Tolerant Temporal Logic* ($\mathsf{ELTL_{FT}}$), whose syntax is shown in Table 2.1. The formulas derived from *cform* — called *counter formulas* — restrict counters, while the formulas derived from *gform* — called *guard formulas* — restrict shared variables. The formulas derived from *pform* are *propositional formulas.* The temporal operators $\mathbf{F}$ and $\mathbf{G}$ follow the standard semantics [CGP99, BK08], that is, for a configuration σ and an infinite schedule τ, it holds that $\mathsf{path}(\sigma, \tau) \models \varphi$, if:

1. $\sigma \models \varphi$, when φ is a propositional formula,
2. $\exists \tau', \tau'' : \tau = \tau' \cdot \tau''.\ \mathsf{path}(\tau'(\sigma), \tau'') \models \psi$, when $\varphi = \mathbf{F}\,\psi$,
3. $\forall \tau', \tau'' : \tau = \tau' \cdot \tau''.\ \mathsf{path}(\tau'(\sigma), \tau'') \models \psi$, when $\varphi = \mathbf{G}\,\psi$.

To stress that the formula should be satisfied by *at least one path*, we prepend $\mathsf{ELTL_{FT}}$-formulas with the existential path quantifier $\mathbf{E}$. We use the shorthand notation *true* for a valid propositional formula, e.g., $\bigwedge_{i\in\emptyset} \boldsymbol{\kappa}[i] = 0$. We also denote with $\mathsf{ELTL_{FT}}$ the set of all formulas that can be written using the logic $\mathsf{ELTL_{FT}}$.

Given a configuration σ, a finite schedule τ applicable to σ, and a propositional formula ψ, by $\mathsf{Cfgs}(\sigma, \tau) \models \psi$ we denote that ψ holds in every configuration σ' visited by the path $\mathsf{path}(\sigma, \tau)$. In other words, for every prefix τ' of τ, we have that $\tau'(\sigma) \models \psi$.

$$\begin{aligned}
\psi &::= \mathit{pform} \mid \mathsf{G}\,\psi \mid \mathsf{F}\,\psi \mid \psi \wedge \psi \\
\mathit{pform} &::= \mathit{cform} \mid \mathit{gform} \vee \mathit{cform} \\
\mathit{cform} &::= \bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0 \mid \bigwedge_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] = 0 \mid \mathit{cform} \wedge \mathit{cform} \\
\mathit{gform} &::= \mathit{guard} \mid \neg \mathit{gform} \mid \mathit{gform} \wedge \mathit{gform}
\end{aligned}$$

Table 2.1: The syntax of $\mathsf{ELTL_{FT}}$-formulas: *pform* defines propositional formulas, and ψ defines temporal formulas. We assume that $\mathit{Locs} \subseteq \mathcal{L}$ and $\mathit{guard} \in \Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}}$.

2.4 Benchmarks

We list a number of examples of fault-tolerant distributed algorithms found in the literature, which we also used to conduct the experiments in Chapter 3 and Chapter 4. We illustrate their threshold automata and formalize their specifications. As our method requires negations of original specifications, we give only the negated formulas in $\mathsf{ELTL_{FT}}$.

Consistent broadcast (STRB) [ST87b]. This is the example from Figure 1.1, whose threshold automaton is given in Figure 2.1. In Section 1.1 we have presented its safety specification unforgeability, here denoted by $\mathsf{S1}$, and two liveness specifications, namely correctness and relay, here respectively denoted by $\mathsf{L1}$ and $\mathsf{L2}$. The negated safety specification $\mathsf{S1}$ is as follows:

S1: $\mathsf{E}\,((\bigvee_{\ell=\ell_1} \boldsymbol{\kappa}[\ell] \neq 0) \wedge \mathsf{F} \bigvee_{\ell=\ell_3} \kappa[\ell] \neq 0)$.

For liveness specifications, all our benchmarks have similar fairness constraints, that is the property of reliable communication that requires the processes to eventually receive the messages from all other correct processes. The fairness constraint that encodes reliable communication for STRB is as follows:

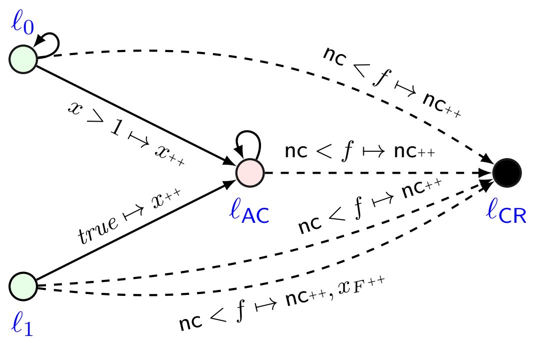
$$\varphi_{rc} \equiv (x < t + 1 \vee \bigwedge_{\ell=\ell_0} \boldsymbol{\kappa}[\ell] = 0) \wedge (x < n - t \vee \bigwedge_{\ell \in \{\ell_0, \ell_1, \ell_2\}} \boldsymbol{\kappa}[\ell] = 0),$$

where locations are chosen because they have an outgoing edge with the corresponding threshold guard.

Using φ_{rc}, we write the liveness properties $\mathsf{L1}$ and $\mathsf{L2}$ as:

L1: $\mathsf{E}\,(\mathsf{G}\,\mathsf{F}\,\varphi_{rc} \wedge \bigwedge_{\ell=\ell_0} \kappa[\ell] = 0 \wedge \mathsf{G} \bigwedge_{\ell=\ell_3} \kappa[\ell] = 0)$,

L2: $\mathsf{E}\,(\mathsf{G}\,\mathsf{F}\,\varphi_{rc} \wedge \mathsf{F}\,(\bigvee_{\ell=\ell_3} \kappa[\ell] \neq 0 \wedge \mathsf{G}\,(\bigvee_{\ell\neq\ell_3} \kappa[\ell] \neq 0))$.



ℓ_0
ℓ_1
ℓ_{AC}
ℓ_{CR}
$x > 1 \mapsto x_{++}$
$true \mapsto x^{++}$
$\mathsf{nc} < f \mapsto \mathsf{nc}_{++}$
$\mathsf{nc} < f \mapsto \mathsf{nc}_{++}$
$\mathsf{nc} < f \mapsto \mathsf{nc}^{++}$
$\mathsf{nc} < f \mapsto \mathsf{nc}^{++}, x_F{}^{++}$

Figure 2.2: The threshold automaton corresponding to the Folklore Reliable Broadcast (FRB). Shared variables x and x_F represents numbers of messages sent by correct and faulty processes, respectively, and nc is the number of crashed processes.

Folklore Reliable Broadcast (FRB). [CT96] This algorithms tolerates crash faults, and its threshold automaton is given in Figure 2.2. FRB has exactly the same specifications S1, L1, and L2 as STRB, but fewer local states. Note that the accepting local state is here denoted by ℓ_{AC} instead of ℓ_3. We also have a special state ℓ_{CR}, such that a process is in this state if and only if it has crashed. As a process can crash at any time, there is a rule from every location to the crash state, but clearly, there is no outgoing rule from ℓ_{CR}.

Asynchronous Byzantine agreement (ABA). [BT85] ABA has exactly the same specifications S1, L1, and L2 as STRB, but more local states and guards. As we will use it in the following chapter, its threshold automaton is depicted in Figure 3.1, and the accepting location is denoted by ℓ_5. In addition to φ_{rc}, ABA has four fairness constraints that enforce local progress of enabled process transitions, e.g., $\mathbf{G\,F}\,(x < 2t+1 \vee \bigwedge_{\ell=\ell_4} \boldsymbol{\kappa}[\ell] = 0)$.

Condition-based consensus (CBC). [MMPR03] CBC has two unique initial local states, where the processes are initialized with values 0 and 1 respectively, and two accepting states AC0 and AC1. In total, our threshold automaton contains 7 locations and 14 rules, which is why we omit illustrating it here. The negation of *termination* is defined as follows:

L1: $\mathbf{E}\,(\mathbf{G\,F}\,\varphi_{rc} \wedge |\kappa[\ell_0] - \kappa[\ell_1]| > t \wedge \mathbf{G} \bigvee_{\ell \notin \{\mathrm{AC,CR}\}} \kappa[\ell] \neq 0)$

The negations of *validity* and *agreement* are as follows:

S1: $\mathbf{E}\,((\bigvee_{\ell=\mathrm{V1}} \boldsymbol{\kappa}[\ell] \neq 0) \wedge \mathbf{F} \bigvee_{\ell=\mathrm{AC0}} \kappa[\ell] \neq 0)$

S2: $\mathbf{E}\,(|\kappa[\ell_0] - \kappa[\ell_1]| > t \wedge \mathbf{F}\,(\bigvee_{\ell \in \{\mathrm{AC0,AC1}\}} \kappa[\ell] \neq 0))$

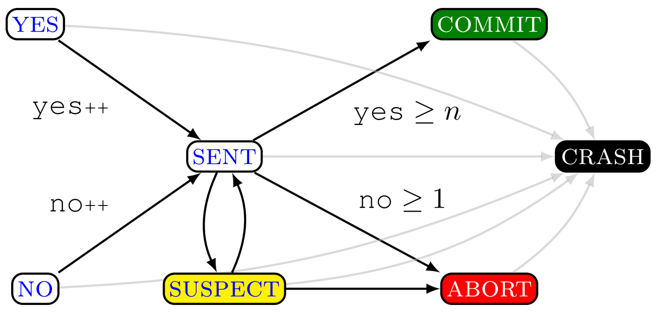


Figure 2.3: Threshold automaton for the non-blocking atomic commit algorithms.

Non-blocking atomic commit. [Ray97, Gue02] Threshold automaton of these algorithms is provided in Figure 2.3. There are two shared variables `yes` and `no`, and two simple threshold guards, namely one universal ($\texttt{yes} \geq n$) and one existential ($\texttt{no} \geq 1$).

In addition to the fairness constraint φ_{rc}, the algorithms NBAC, NBACC, NBACG use a fairness constraint φ_{fd} on a failure detector defined as: $\bigwedge_{\ell=\text{SUSPECT}} \kappa[\ell] = 0 \wedge \mathbf{G}\,(\bigwedge_{\ell=\text{CRASH}} \kappa[\ell] = 0)$.

The negation of *termination* is defined as follows:

L1: $\mathbf{E}\,(\mathbf{G}\,\mathbf{F}\,\varphi_{rc} \wedge \varphi_{fd} \wedge \mathbf{G} \bigvee_{\ell \notin \{\text{COMMIT,ABORT,CRASH}\}} \kappa[\ell] \neq 0)$

The negations of *abort-validity* and *agreement* are as follows:

S1: $\mathbf{E}\,((\bigvee_{\ell=\text{NO}} \boldsymbol{\kappa}[\ell] \neq 0) \wedge \mathbf{F} \bigvee_{\ell=\text{COMMIT}} \kappa[\ell] \neq 0)$.

S2: $\mathbf{E}\,(\mathbf{F}\,(\bigvee_{\ell=\text{ABORT}} \kappa[\ell] \neq 0 \wedge \bigvee_{\ell=\text{COMMIT}} \kappa[\ell] \neq 0))$.

CFCS and C1CS. [DS06, BGMR01] Threshold automata corresponding to these algorithms contain 9 local states and 34 rules. The negation of *fast termination* for value 0 is:

L1: $\mathbf{E}\,(\mathbf{G}\,\mathbf{F}\,\varphi_{rc} \wedge \bigwedge_{\ell \neq \text{V0}} \kappa[\ell] = 0 \wedge \mathbf{G} \bigvee_{\ell \notin \{\text{D0,CR}\}} \kappa[\ell] \neq 0)$

The negation of *one-step* for value 0 is:

S1: $\mathbf{E}\,(\bigwedge_{\ell \neq \text{V0}} \kappa[\ell] = 0 \wedge \mathbf{F} \bigvee_{\ell \in \{\text{D1,U0,U1}\}} \kappa[\ell] \neq 0)$

BOSCO. [SvR08] One-step Byzantine asynchronous consensus is our most challenging benchmark. Its threshold automaton contains 8 locations, 20 transitions, three non-trivial threshold guards, and different properties are expected to hold under three different resilience conditions. We illustrate it in Figure 5.9 in Chapter 6, with threshold guards $\tau_{\mathsf{A}} = n - t$, $\tau_{\mathsf{D0}} = \tau_{\mathsf{D1}} = \frac{n+3t+1}{2}$, and $\tau_{\mathsf{U0}} = \tau_{\mathsf{U1}} = \frac{n-t}{2}$.

The negation of *fast termination* for value 0 is:

L1: $\mathbf{E}\,(\mathbf{G}\,\mathbf{F}\,\varphi_{rc} \wedge \bigwedge_{\ell \neq \mathrm{V0}} \kappa[\ell] = 0 \wedge \mathbf{G} \bigvee_{\ell \notin \{\mathrm{D0,CR}\}} \kappa[\ell] \neq 0)$

The negations of *Lemma 3* and *Lemma 4* of [SvR08] are:

S1: $\mathbf{E}\,(\mathbf{F}\,(\bigvee_{\ell=\mathrm{D0}} \kappa[\ell] \neq 0 \wedge \bigvee_{\ell=\mathrm{D1}} \kappa[\ell] \neq 0))$

S2: $\mathbf{E}\,(\mathbf{F}\,(\bigvee_{\ell=\mathrm{D0}} \kappa[\ell] \neq 0 \wedge \bigvee_{\ell=\mathrm{U1}} \kappa[\ell] \neq 0))$

2.5 Verification Problems

In this section we discuss the verification problems for fault-tolerant distributed algorithms. In Section 1.2 we have roughly stated four challenges $C1$—$C4$, and here we formulate them more precisely.

In each of the challenges, it is important to handle the resilience conditions precisely.

Example 2.9. Unforgeability, the safety property from Section 1.1, expressed in terms of Figure 2.1 means that no process should ever enter ℓ_3 if initially all processes are in ℓ_0, given that $n > 3t \wedge t \geq f \geq 0$. We can express this in the counter system: under the resilience condition $n > 3t \wedge t \geq f \geq 0$, given an initial configuration σ, with $\sigma.\boldsymbol{\kappa}[\ell_0] = n - f$, to verify safety, we have to establish the absence of a schedule τ that satisfies $\sigma' = \tau(\sigma)$ and $\sigma'.\boldsymbol{\kappa}[\ell_3] > 0$.

In order to be able to answer this question, we have to deal with resilience conditions precisely: Observe that ℓ_3 is unreachable, as all outgoing transitions from ℓ_0 contain guards that evaluate to false initially, and since all processes are in ℓ_0 no process ever increases x. A slight modification of $t \geq f$ to $t + 1 \geq f$ in the resilience condition changes the result, i.e., one fault too many breaks the system. For example, if $n = 4$, $t = 1$, and $f = 2$, then the new resilience condition holds, but as the guard $\gamma_1 : x \geq (t+1) - f$ is now initially true, then one correct process can fire the rule r_2 and increase x. Now when $x = 1$, the guard $\gamma_2 : x \geq (n - t) - f$ becomes true, so that the process can fire the rule r_4 and reach the state ℓ_3. This tells us that unforgeability is not satisfied in the system where the resilience condition is $n > 3t \wedge t + 1 \geq f \geq 0$. ◁

Analysis of reachability properties is addressed in the verification challenge $C1$, and solved in Chapter 3. We formalize it as follows:

Challenge 2.1 (Parameterized reachability)**.** *Given a threshold automaton TA and a Boolean formula B over $\{\boldsymbol{\kappa}[\ell] = 0 \mid \ell \in \mathcal{L}\}$, check whether there are parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, an initial configuration $\sigma_0 \in I$ with $\sigma_0.\mathbf{p} = \mathbf{p}$ and a finite schedule τ applicable to σ_0 such that $\tau(\sigma_0) \models B$.*

In Chapter 3 we show that, if such a schedule exists, then there is also a schedule of bounded length. In Chapter 4, we address the challenge $C2$, that is, we extend the reachability question to specifications of *counterexamples to safety and liveness* of FTDAs from the literature.

Challenge 2.2 (Parameterized unsafety & non-liveness)**.** *Given a threshold automaton TA and an $\mathsf{ELTL_{FT}}$ formula ψ, check whether there are parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, an initial configuration $\sigma_0 \in I$ with $\sigma_0.\mathbf{p} = \mathbf{p}$, and an infinite schedule τ of $\mathsf{Sys}(\mathsf{TA})$ applicable to σ_0 such that $\mathsf{path}(\sigma_0, \tau) \models \psi$.*

Recall that the $\mathsf{ELTL_{FT}}$ logic formalizes negations of specifications. Checking if there are parameter values and a path in this concrete system (with these parameters) that satisfies a formula from $\mathsf{ELTL_{FT}}$, is equivalent to checking if the system satisfies the specifications for all values of parameters. If we find parameter values and a path that satisfies ψ, this is a witness that the specification $\neg\psi$ is violated in the system with those parameter values. If there are no such parameter values and no path satisfying ψ, we know that the specification $\neg\psi$ cannot be violated, that is, the system is correct for any values of parameters.

Parameterized unsafety & non-liveness is the core of this thesis and it is addressed in Chapter 4. The solution is based on the solution of the Parameterized reachability, which is the topic of Chapter 3. In this work we also extend these questions in two different research directions.

Namely, in Chapter 5 we focus on the $C3$ challenge, that will be formally introduced as Challenge 5.1. We define a skeleton of a threshold automaton, that can intuitively be understood as a threshold automaton with undefined threshold guards. Then $C3$ becomes a problem of automatically finding threshold guards that together with the given skeleton form a threshold automaton, whose counter system satisfies the given specifications (whose negations are expressed in $\mathsf{ELTL_{FT}}$).

In order to address the $C4$ challenge, in Chapter 6 we define a probabilistic threshold automaton (PTA) that formalizes randomized distributed algorithms, and to capture their specifications, we introduce an extension of the $\mathsf{ELTL_{FT}}$ logic, called multi-round $\mathsf{ELTL_{FT}}$. We formalize Challenge 6.1, similarly as Challenge 2.2: given a PTA and a multi-round $\mathsf{ELTL_{FT}}$ formula ψ, find (if there exist) parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, an initial configuration $\sigma_0 \in I$ with $\sigma_0.\mathbf{p} = \mathbf{p}$, and an infinite schedule τ of $\mathsf{Sys}(\mathsf{PTA})$ applicable to σ_0 such that $\mathsf{path}(\sigma_0, \tau) \models \psi$. Sometimes we require that the probability of finding such a path, that violates the specification, is 0.

2.6 Discussion

Classically, modeling a distributed system is done by parallel composition and interleaving semantics: A state of a distributed system that consists of n processes is modeled as n-dimensional vector of local states. The transition to a successor state is then defined by non-deterministically picking a process, say i, and changing the ith component of the n-dimensional vector according to the local transition relation of the process. This encoding is not convenient for parameterized verification, as we allow n to grow unboundedly, which would require vectors of unbounded length. However, for our domain of threshold-guarded algorithms, we do not take into account the precise n-dimensional vector so that we use a more efficient encoding, namely counter systems [Lub84, PXZ02, AGP16, KVW17].

There are two reasons for our restrictions in the temporal logic: On one hand, in our benchmarks, there is no need to find counterexamples that contain a configuration that satisfies $\boldsymbol{\kappa}[\ell] = 0 \vee \boldsymbol{\kappa}[\ell'] = 0$ for some $\ell, \ell' \in \mathcal{L}$. One would only need such a formula to specify requirement that at least one process is at location ℓ and at least one process is at location ℓ' (the disjunction would be negated in the specification), which is unnatural for fault-tolerant distributed algorithms. On the other hand, enriching our logic with $\bigvee_{i \in Locs} \boldsymbol{\kappa}[i] = 0$ allows one to express tests for zero in the counter system, which leads to undecidability [BJK^{+}15]. For the same reason, we avoid disjunction, as it would allow one to indirectly express test for zero: $\boldsymbol{\kappa}[\ell] = 0 \vee \boldsymbol{\kappa}[\ell'] = 0$.

An existing tool for reachability analysis of counter systems is FAST [BFLP08]. We compare our approach in reachability analysis with FAST in Section 3.10. This tool uses acceleration in order to compute the set of reachable states. The method is complete for the class of flattable systems [BFLS05]. The relation between threshold automata and flattable systems is discussed in [KKW18]. In this work it is proven that the restrictions we impose on threshold automata imply that our systems are flattable. Moreover, multiple relaxations of our restrictions have been studied in [KKW18]. For instance, they introduce extensions of threshold automata that use the following properties and their combinations: (i) non-linear (but piecewise monotone) functions in threshold guards, (ii) guards that compare the difference between two shared variables against a threshold, (iii) decrementing shared variables, and (iv) incrementing variables inside self-loops. Furthermore, for each of the classes they investigate the existence of a bounded diameter, and check decidability for reachability properties.

Recently, in [SKWZ18], threshold automata have been used for modeling synchronous threshold-based FTDAs, for example synchronous byzantine resilient reliable broadcast [ST87a]. Characteristics of threshold guards are the key insight for (i) investigating whether an algorithm induces a bounded diameter despite the parameterization, and (ii) for computing the diameter when it exists. This allows parameterized bounded model checking of reachability properties in synchronous FTDAs that have bounded diameter. It also yields that in general this problem is undecidable for synchronous FTDAs.

CHAPTER 3

Parameterized Reachability

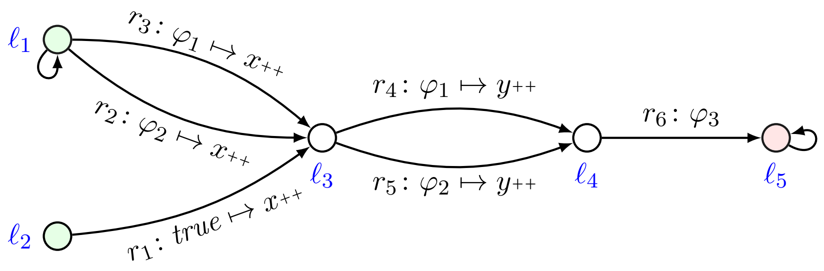
This chapter addresses Challenge 2.1 introduced in Section 2.5, that we recall here:

Challenge 2.1 (Parameterized reachability). *Given a threshold automaton* TA *and a Boolean formula* B *over* $\{\boldsymbol{\kappa}[\ell] = 0 \mid \ell \in \mathcal{L}\}$*, check whether there are parameter values* $\mathbf{p} \in \mathbf{P}_{RC}$*, an initial configuration* $\sigma_0 \in I$ *with* $\sigma_0.\mathbf{p} = \mathbf{p}$ *and a finite schedule* τ *applicable to* σ_0 *such that* $\tau(\sigma_0) \models B$*.*

This question has also been addressed in [KVW17]. The technique there first applies data and counter abstraction, and then runs bounded model checking (BMC). Given an FTDA, it computes an upper bound on the diameter of the system. This makes BMC complete for reachability properties: it always finds a counterexample, if there is an actual error. Nonetheless, to verify state-of-the-art FTDAs, we need to enumerate all possible traces of bounded length, which is often inefficient. Therefore, further improvement is needed.

In contrast to the SAT-based method in [KVW17], we suggest a new method: encoding bounded executions over integer counters in SMT. In addition, we introduce a new form of reduction that exploits acceleration and the structure of the FTDAs, called parameterized path reduction with acceleration (PARA^2). Namely, we introduce so called schemas, that generate families of traces, and prove that for checking reachability, it is enough to analyze schemas. This aggressively prunes the execution space to be explored by the solver. In this way, we verify safety of seven FTDAs that were out of reach before.

An intuitive explanation of our method is given in Section 3.1, for a threshold automaton from Figure 3.1 that is motivated by the distributed asynchronous broadcast protocol from [BT85]. For the formal explanation, we present the necessary definitions in Section 3.2, with the exception of the definition of a schema that is introduced in Section 3.3 together with the main result, stating the following: There is a set of schemas, whose cardinality depends only on the number of guards in the algorithm, such that the set of states reachable using schemas matches the set of all reachable states. Our construction



ℓ_1
$r_3\colon \varphi_1 \mapsto x{++}$
$r_2\colon \varphi_2 \mapsto x{++}$
ℓ_2
$r_1\colon true \mapsto x{++}$
ℓ_3
$r_4\colon \varphi_1 \mapsto y{++}$
$r_5\colon \varphi_2 \mapsto y{++}$
ℓ_4
$r_6\colon \varphi_3$
ℓ_5

Figure 3.1: A threshold automaton with threshold guards "$\varphi_1\colon x \geq \lceil (n+t)/2 \rceil - f$", "$\varphi_2\colon y \geq (t+1) - f$", and "$\varphi_3\colon y \geq (2t+1) - f$", that corresponds to the ABA algorithm [BT85].

of schemas is done in layers, and the first step is described in Section 3.4, where we construct schemas for finite paths that consist of only one loop or one transition, and where the evaluation of the guards does not change. Technical proofs of this section can be found in Section 3.8. The second layer combines the results of the first one, where we allow multiple loops or transitions, but still no change of guards, which is demonstrated in Section 3.5. Again, we leave the technical proofs for Section 3.9. The final step for obtaining schemas combines the previous steps, by cutting an arbitrary path to segments where guards do not change their evaluations, which is illustrated in Section 3.6. In Section 3.7.1 we see how this construction can be optimized, and in Section 3.7.2 we present our encoding of schemas in SMT. We recall the experimental evaluation from [KLVW17a] in Section 3.7.3.

3.1 Our approach at a glance

We use Figure 3.1 to describe our contributions in this section. The figure presents a threshold automaton TA over two shared variables x and y and parameters n, t, and f, which is inspired by the distributed asynchronous broadcast protocol from [BT85]. There, $n - f$ correct processes concurrently follow the control flow of TA, and f processes are Byzantine faulty. As is typical for fault-tolerant distributed algorithms, the parameters must satisfy a resilience condition, e.g., $n > 3t \wedge t \geq f \geq 0$, that is, less than a third of the processes are faulty.

In order to intuitively explain results of this chapter, we address an instance of the parameterized reachability problem, e.g., can at least one correct process reach the local state ℓ_5, when $n - f$ correct processes start in the local state ℓ_1? Or, in terms of counter systems, is a configuration with $\boldsymbol{\kappa}[\ell_5] \neq 0$ reachable from an initial configuration with $\boldsymbol{\kappa}[\ell_1] = n - f \wedge \boldsymbol{\kappa}[\ell_2] = 0$? As discussed in [KVW17], acceleration does not affect reachability, and precise treatment of the resilience condition and threshold guards is crucial for solving this problem.

3.1.1 Schemas

As initially x and y are zero, threshold guards φ_1, φ_2, and φ_3 evaluate to false. As rules may increase variables, these guards may eventually become true. (In this example we do not consider guards like $x < t$ that are initially true and become false, although we treat them later.) In fact, initially only r_1 is unlocked. Because r_1 increases x, it may unlock φ_1. Thus r_4 becomes unlocked. Rule r_4 increases y and thus repeated application of r_4 (by different processes) first unlocks φ_2 and then φ_3. We introduce a notion of a *context* that is the set of threshold guards that evaluate to true in a configuration. For our example we observe that each path goes through the following sequence of contexts $\{\}$, $\{\varphi_1\}$, $\{\varphi_1, \varphi_2\}$, and $\{\varphi_1, \varphi_2, \varphi_3\}$. In fact, the sequence of contexts in a path is always monotonic, as the shared variables can only be increased.

The conjunction of the guards in the context $\{\varphi_1, \varphi_2\}$ implies the guards of the rules r_1, r_2, r_3, r_4, r_5; we call these rules unlocked in the context. This motivates our definition of a *schema*: a sequence of contexts and rules. We give an example of a schema below, where inside the curly brackets we give the contexts, and fixed sequences of rules in between. (We discuss the underlined rules below.)

$$\begin{aligned} S = \{\}\ &\underline{r_1}, \underline{r_1}\ \{\varphi_1\}\ \underline{r_1}, \underline{r_3}, r_4, \underline{r_4}\ \{\varphi_1, \varphi_2\} \\ &r_1, r_2, r_3, r_4, r_5, r_4, \underline{r_5}\ \{\varphi_1, \varphi_2, \varphi_3\}\ r_1, r_2, r_3, r_4, \underline{r_5}, \underline{r_6}\ \{\varphi_1, \varphi_2, \varphi_3\} \end{aligned} \tag{3.1}$$

Next we introduce a notion of a schedule generated by a schema. To this end, we analyze the following schedule, where, e.g., r_1^1 is an abbreviation for the transition $(r_1, 1)$:

$$\tau' = \underbracket{r_1^1}_{\tau'_1}, \underbrace{r_1^1}_{t_1}, \underbracket{r_1^1, r_3^1}_{\tau'_2}, \underbrace{r_4^1}_{t_2}, \underbracket{\ }_{\tau'_3}, \underbrace{r_5^1}_{t_3}, \underbracket{r_5^2, r_6^4}_{\tau'_4} \tag{3.2}$$

We say that τ' is generated by schema S, because the sequence of the underlined rules in S matches the sequence of rules appearing in τ'. In this chapter, we show that the schedules generated by a few schemas — one per each monotonic sequence of contexts — span the set of all reachable configurations. To this end, we apply the PARA^2 technique to relate arbitrary schedules to their representatives, which are generated by schemas.

3.1.2 PARAmeterized PAth Reduction with Acceleration - PARA^2

Consider, e.g., the initial state σ_0 with $n = 5$, $t = f = 1$, $\boldsymbol{\kappa}[\ell_1] = 1$, and $\boldsymbol{\kappa}[\ell_2] = 3$. We are interested in whether there is a schedule that reaches a configuration, where all processes are in state ℓ_5. Consider the following schedule:

$$\tau \ = \ \underbracket{r_1^1}_{\tau_1}, \underbrace{r_1^1}_{t_1}, \underbracket{r_3^1, r_1^1}_{\tau_2}, \underbrace{r_4^1}_{t_2}, \underbracket{\ }_{\tau_3}, \underbrace{r_5^1}_{t_3}, \underbracket{r_6^1, r_5^1, r_5^1, r_6^1, r_6^1, r_6^1}_{\tau_4}$$

Observe that after r_1^1, r_1^1, variable $x = 2$, and thus φ_1 is true. Hence transition t_1 changes the context from $\{\}$ to $\{\varphi_1\}$. Similarly t_2 and t_3 change the context. Context changing transitions are marked with curly brackets. Between them we have the subschedules $\tau_1, \dots, \tau_4$ (τ_3 is empty) marked with square brackets.

To show that this schedule is captured by the schema, we apply our reduction arguments inspired by the mover analysis [Lip75, EF82], regarding distributed computations: As the guards φ_2 and φ_3 evaluate to true in τ_4, and r_5 precedes r_6 in the control flow of the TA, all transitions r_5^1 can be moved to the left in τ_4. Similarly, r_1^1 can be moved to the left in τ_2. The resulting schedule is applicable and leads to the same configuration as the original one. Further, we can accelerate the adjacent transitions with the same rule, e.g., the sequence r_5^1, r_5^1 can be transformed into r_5^2. Thus, we transform subschedules τ_i into τ_i', and arrive at the schedule τ' from (3.2), which we call the representative schedule of τ. Importantly for reachability checking, if τ and τ' are applied to the same configuration, they end in the same configuration. These arguments are formalized in Sections 3.4–3.6.

3.1.3 Encoding a Schema in SMT

One of the key insights in this work is that reachability checking via schemas can be encoded efficiently as SMT queries in linear integer arithmetic. In more detail, finite paths of counter systems can be expressed with inequalities over counters such as $\boldsymbol{\kappa}[\ell_2]$ and $\boldsymbol{\kappa}[\ell_3]$, shared variables such as x and y, parameters such as n, t, and f, and acceleration factors. In particular, threshold guards and resilience conditions are expressions in linear integer arithmetic.

We give an example of reachability checking with SMT using the following simple schema:

$$\{\}\ r_1, r_1\ \{\varphi_1\}$$

that is a part of the schema S in (3.1). To obtain a complete encoding for S, one can encode similarly the other simple schemas and combine them.

To this end, we have to express constraints on three configurations σ_0, σ_1, and σ_2. For the configuration σ_0, we introduce integer variables: $\boldsymbol{\kappa}_1^0, \dots, \boldsymbol{\kappa}_5^0$ for local state counters, x^0 and y^0 for shared variables, and n, t, and f for parameters. As it is written in Equations (3.3), the configuration σ_0 should satisfy the initial constraints, and its context should be empty:

$$\begin{aligned} &\boldsymbol{\kappa}_1^0 + \boldsymbol{\kappa}_2^0 = n - f \wedge \boldsymbol{\kappa}_3^0 = \boldsymbol{\kappa}_4^0 = \boldsymbol{\kappa}_5^0 = 0 \wedge x^0 = y^0 = 0 \\ &\wedge n \geq 3t \wedge t \geq f \geq 0 \wedge (\neg\varphi_1 \wedge \neg\varphi_2 \wedge \neg\varphi_3)[x^0/x, y^0/y] \end{aligned} \tag{3.3}$$

The configuration σ_1 is reached from σ_0 by applying a transition with the rule r_1 and an acceleration factor δ_1, and the configuration σ_2 is reached from σ_1 by applying a transition with the rule r_1 and an acceleration factor δ_2. Applying transition with the rule r_1 to σ_0 just means to increase both $\boldsymbol{\kappa}[\ell_3]$ and x by δ_1 and decrease $\boldsymbol{\kappa}[\ell_2]$ by δ_1. Hence, we introduce four fresh variables per transition and write down the arithmetic operations. According to the schema, configuration σ_2 has the context $\{\varphi_2\}$. Equations (3.4) and (3.5) express these constraints:

$$\boldsymbol{\kappa}_3^1 = \boldsymbol{\kappa}_3^0 + \delta^1 \wedge \boldsymbol{\kappa}_2^1 = \boldsymbol{\kappa}_2^0 - \delta^1 \wedge x^1 = x^0 + \delta^1 \wedge x^1 \geq 0 \tag{3.4}$$

$$\boldsymbol{\kappa}_3^2 = \boldsymbol{\kappa}_3^1 + \delta^2 \wedge \boldsymbol{\kappa}_2^2 = \boldsymbol{\kappa}_2^1 - \delta^2 \wedge x^2 = x^1 + \delta^2 \wedge x^2 \geq 0$$
$$\wedge (\varphi_1 \wedge \neg\varphi_2 \wedge \neg\varphi_3)[x^2/x, y^0/y] \tag{3.5}$$

Finally, we have to express the reachability question for all paths generated by the simple schema $\{\}\ r_1, r_1\ \{\varphi_1\}$. Recall, that we want to check, whether there is a configuration with $\boldsymbol{\kappa}[\ell_5] \neq 0$ reachable from an initial configuration with $\boldsymbol{\kappa}[\ell_1] = n - f$ and $\boldsymbol{\kappa}[\ell_2] = 0$. This is formally written as:

$$\boldsymbol{\kappa}_1^0 = n - f \wedge \boldsymbol{\kappa}_2^0 = 0 \wedge \boldsymbol{\kappa}_5^0 \neq 0 \tag{3.6}$$

Note that we only check $\boldsymbol{\kappa}_5^0$ against zero, as the local state ℓ_5 is never updated by the rule r_1. It is easy to see that Equations (3.3)–(3.6) do not have a solution, and thus all paths generated by the schema $\{\}\ r_1, r_1\ \{\varphi_1\}$ are safe. By writing down constraints for the other three simple schemas in Equation (3.1), we can ensure that the paths generated by the whole schema are safe as well. As discussed in Section 3.1.2, our results also imply safety of the paths whose representatives are generated by the schema. Details on the SMT encoding can be found in Section 3.7.2.

3.2 Preliminaries

Before we start explaining in details the technique intuitively described in Section 3.1, we formally introduce necessary definitions.

3.2.1 Looplets

Definition 3.1. *Given a threshold automaton* $(\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$*, we define the* precedence relation $\prec_P$*: for a pair of rules* $r_1, r_2 \in \mathcal{R}$*, it holds that* $r_1 \prec_P r_2$ *if and only if* $r_1.to = r_2.from$*. We denote by* $\prec_P^+$ *the transitive closure of* $\prec_P$*. Further, we say that* $r_1 \sim_P r_2$*, if* $r_1 \prec_P^+ r_2 \wedge r_2 \prec_P^+ r_1$*, or* $r_1 = r_2$*.*

Example 3.1. Recall from Section 2.1 that we limit ourselves to canonical threshold automata, i.e., those where $r.\mathbf{u} = \mathbf{0}$ for all rules $r \in \mathcal{R}$ that satisfy $r \prec_P^+ r$. In the threshold automaton from Figure 3.2 we have that $r_2 \prec_P r_3 \prec_P r_4 \prec_P r_5 \prec_P r_6 \prec_P r_8 \prec_P r_2$. Thus, we have that $r_2 \prec_P^+ r_2$. In our case this implies that $r_2.\mathbf{u} = \mathbf{0}$ by definition. Similarly we can conclude that $r_3.\mathbf{u} = r_4.\mathbf{u} = r_5.\mathbf{u} = r_6.\mathbf{u} = r_7.\mathbf{u} = r_8.\mathbf{u} = \mathbf{0}$. ◁

The relation $\sim_P$ defines equivalence classes of rules. An equivalence class corresponds to a loop or to a single rule that is not part of a loop. Hence, we use the term looplet for one such equivalence class. For a given set of rules $\mathcal{R}$ let $\mathcal{R}/\sim$ be the set of equivalence classes

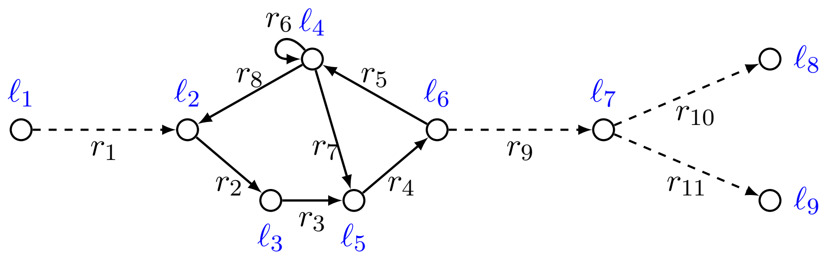


Figure 3.2: A threshold automaton TA with local states $\mathcal{L} = \{\ell_i : 1 \le i \le 9\}$ and rules $\mathcal{R} = \{r_i : 1 \le i \le 11\}$. The rules drawn with solid arrows $\{r_2, \dots, r_8\}$ constitute a single equivalence class, while all other rules are singleton equivalence classes.

defined by $\sim_P$. We denote by $[r]$ the equivalence class of rule r. For two classes c_1 and c_2 from $\mathcal{R}/\!\sim$ we write $c_1 \prec_C c_2$ iff there are two rules r_1 and r_2 in $\mathcal{R}$ satisfying $[r_1] = c_1$ and $[r_2] = c_2$ and $r_1 \prec_P^+ r_2$ and $r_1 \not\sim_P r_2$. As the relation $\prec_C$ is a strict partial order, there are linear extensions of $\prec_C$. Below, we fix an *arbitrary* of these linear extensions to sort transitions in a schedule: We denote by $\prec_C^{lin}$ a linear extension of $\prec_C$.

Example 3.2. Consider Figure 3.2. Here, $r_9 \prec_P r_{10}$, since $r_9.to = \ell_7 = r_{10}.from$. By transitivity we have $r_4 \prec_P^+ r_{10}$. Further, inside a looplet we have, e.g., $r_4 \prec_P^+ r_2$, because $r_4 \prec_P r_5 \prec_P r_8 \prec_P r_2$. The threshold automaton has five looplets: $c_1 = \{r_1\}$, $c_2 = \{r_2, \dots, r_8\}$, $c_3 = \{r_9\}$, $c_4 = \{r_{10}\}$, and $c_5 = \{r_{11}\}$. From $r_9 \prec_P r_{10}$, it follows that $c_3 \prec_C c_4$, and from $r_4 \prec_P^+ r_{10}$, it follows that $c_2 \prec_C c_4$. We can pick two linear extensions of $\prec_C$, denoted by $\prec_1$ and $\prec_2$. We have $c_1 \prec_1 \cdots \prec_1 c_5$, and $c_1 \prec_2 c_2 \prec_2 c_3 \prec_2 c_5 \prec_2 c_4$. In this work we always fix one linear extension. $\triangleleft$

Remark 3.1. It may seem natural to collapse such loops into singleton local states. In our case studies, e.g, [Gue02], non-trivial loops are used to express non-deterministic choice due to failure detectors [CT96], as shown in Figure 3.3. Importantly, some local states inside the loops appear in the specifications. Thus, one would have to use arguments from distributed computing to characterize when collapsing states is sound. In this work, we present a technique that deals with the loops without need for additional modeling arguments.

3.2.2 Contexts and Slices

The evaluation of the guards in the sets Φ^{rise} and Φ^{fall} in a configuration solely defines whether certain transitions are unlocked (but not necessarily enabled). From Proposition 2.1 (Monotonicity of guards), one can see that after a transition has been applied, more guards from Φ^{rise} may get unlocked and more guards from Φ^{fall} may get locked. In other words, more guards from Φ^{rise} may evaluate to true and more guards from Φ^{fall} may evaluate to false. To capture this intuition, we define:

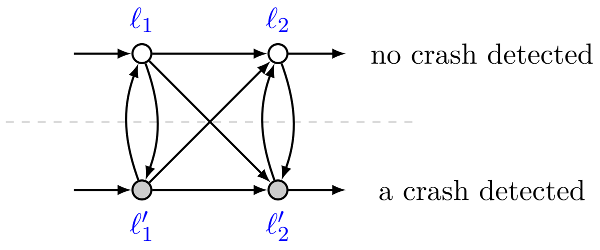


Figure 3.3: A typical structure found in threshold automata that model fault-tolerant algorithms with a failure detector [CT96]. The gray circles depict those local states, where the failure detector reports a crash. The local states ℓ_i and ℓ'_i differ only in the output of the failure detector. As the failure detector output changes non-deterministically, the threshold automaton contains loops of size two.

Definition 3.2. *A* context *is a pair* $(\Omega^{\text{rise}}, \Omega^{\text{fall}})$ *with* $\Omega^{\text{rise}} \subseteq \Phi^{\text{rise}}$ *and* $\Omega^{\text{fall}} \subseteq \Phi^{\text{fall}}$*. We denote by* Ω *the pair* $(\Omega^{\text{rise}}, \Omega^{\text{fall}})$*, and by* $|\Omega| = |\Omega^{\text{rise}}| + |\Omega^{\text{fall}}|$*.*

For two contexts $(\Omega_1^{\text{rise}}, \Omega_1^{\text{fall}})$ and $(\Omega_2^{\text{rise}}, \Omega_2^{\text{fall}})$, we define that $(\Omega_1^{\text{rise}}, \Omega_1^{\text{fall}}) \sqsubset (\Omega_2^{\text{rise}}, \Omega_2^{\text{fall}})$ if and only if $\Omega_1^{\text{rise}} \cup \Omega_1^{\text{fall}} \subset \Omega_2^{\text{rise}} \cup \Omega_2^{\text{fall}}$. Then, a sequence of contexts $\Omega_1, \dots, \Omega_m$ is *monotonically increasing*, if $\Omega_i \sqsubset \Omega_{i+1}$, for $1 \le i < m$. Further, a monotonically increasing sequence of contexts $\Omega_1, \dots, \Omega_m$ is *maximal*, if $\Omega_1 = (\emptyset, \emptyset)$ and $\Omega_m = (\Phi^{\text{rise}}, \Phi^{\text{fall}})$ and $|\Omega_{i+1}| = |\Omega_i| + 1$, for $1 \le i < m$. We obtain:

Proposition 3.1. *Every maximal monotonically increasing sequence of contexts is of length* $|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}| + 1$*. There are at most* $(|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}|)!$ *such sequences.*

Example 3.3. For the example in Figure 3.1, we have $\Phi^{\text{rise}} = \{\varphi_1, \varphi_2, \varphi_3\}$, and $\Phi^{\text{fall}} = \emptyset$. Thus, there are $(|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}|)! = 6$ maximal monotonically increasing sequences of contexts. Two of them are $(\emptyset, \emptyset) \sqsubset (\{\varphi_1\}, \emptyset) \sqsubset (\{\varphi_1, \varphi_2\}, \emptyset) \sqsubset (\{\varphi_1, \varphi_2, \varphi_3\}, \emptyset)$ and $(\emptyset, \emptyset) \sqsubset (\{\varphi_3\}, \emptyset) \sqsubset (\{\varphi_1, \varphi_3\}, \emptyset) \sqsubset (\{\varphi_1, \varphi_2, \varphi_3\}, \emptyset)$. All of them have length $|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}| + 1 = 4$. ◁

To every configuration σ, we attach the context consisting of all guards in Φ^{rise} that evaluate to true in σ, and all guards in Φ^{fall} that evaluate to false in σ:

Definition 3.3. *Given a threshold automaton, we define its* configuration context *as a function* $\omega : \Sigma \to 2^{\Phi^{\text{rise}}} \times 2^{\Phi^{\text{fall}}}$ *that for each configuration* $\sigma \in \Sigma$ *gives a context* $(\Omega^{\text{rise}}, \Omega^{\text{fall}})$ *with* $\Omega^{\text{rise}} = \{\varphi \in \Phi^{\text{rise}} : \sigma \models \varphi\}$ *and* $\Omega^{\text{fall}} = \{\varphi \in \Phi^{\text{fall}} : \sigma \not\models \varphi\}$*.*

The following monotonicity result is a direct consequence of Proposition 2.1.

Proposition 3.2. *If a transition* t *is enabled in a configuration* σ*, then either* $\omega(\sigma) \sqsubset \omega(t(\sigma))$*, or* $\omega(\sigma) = \omega(t(\sigma))$*.*

Definition 3.4. *A schedule τ is* steady *for a configuration σ, if for every prefix τ' of τ, the context does not change, i.e., $\omega(\tau'(\sigma)) = \omega(\sigma)$.*

Proposition 3.3. *A schedule τ is steady for a configuration σ if and only if $\omega(\sigma) = \omega(\tau(\sigma))$.*

In the following definition, we associate a sequence of contexts with a path:

Definition 3.5. *Given a configuration σ and a schedule τ applicable to σ, we say that $\mathsf{path}(\sigma, \tau)$* is consistent with *a sequence of contexts $\Omega_1, \dots, \Omega_m$, if there exist indices $n_0, \dots, n_m$, with $0 = n_0 \le n_1 \le \dots \le n_m = |\tau| + 1$, such that for every k, $1 \le k \le m$, and every i with $n_{k-1} \le i < n_k$, it holds that $\omega(\tau^i(\sigma)) = \Omega_k$.*

In fact, every path is consistent with a uniquely defined maximal monotonically increasing sequence of contexts. (Some of the indices $n_0, \dots, n_m$ in Definition 3.5 may be equal.) In Section 3.3, we use this sequence of contexts to construct a schema recognizing many paths that are consistent with the same sequence of contexts.

A context defines which rules of the TA are unlocked. A schedule that is steady for a configuration visits only one context, and thus we can statically remove TA's rules that are locked in the context:

Definition 3.6. *Given a threshold automaton $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$ and a context Ω, we define the* slice *of TA with context $\Omega = (\Omega^{\mathrm{rise}}, \Omega^{\mathrm{fall}})$ as a threshold automaton $\mathsf{TA}|_{\Omega} = (\mathcal{L}, \mathcal{V}, \mathcal{R}|_{\Omega}, RC)$, where a rule $r \in \mathcal{R}$ belongs to $\mathcal{R}|_{\Omega}$ if and only if $\left(\bigwedge_{\varphi \in \Omega^{\mathrm{rise}}} \varphi\right) \rightarrow r.\varphi^{\mathrm{rise}}$ and $\left(\bigwedge_{\psi \in \Phi^{\mathrm{fall}} \setminus \Omega^{\mathrm{fall}}} \psi\right) \rightarrow r.\varphi^{\mathrm{fall}}$.*

In other words, $\mathcal{R}|_{\Omega}$ contains those and only those rules r that are using guards that evaluate to true in all configurations σ with $\omega(\sigma) = \Omega$. These are exactly the guards from $\Omega^{\mathrm{rise}} \cup (\Phi^{\mathrm{fall}} \setminus \Omega^{\mathrm{fall}})$. When $\omega(\sigma) = \Omega$, then all guards from Ω^{rise} evaluate to true, and then $r.\varphi^{\mathrm{rise}}$ must also be true. As Ω^{fall} contains those guards from Φ^{fall} that evaluate to false in σ, then all other guards from Φ^{fall} must evaluate to true, and then $r.\varphi^{\mathrm{fall}}$ must be true as well.

3.3 Main Result: A Complete Set of Schemas

To address parameterized reachability, we introduce schemas: alternating sequences of contexts and rule sequences. A schema serves as a pattern for infinitely many paths, and it is used to efficiently encode parameterized reachability in SMT. As parameters give rise to infinitely many initial states, a schema captures an *infinite* set of paths. We show how to construct a *finite* set of schemas $\mathcal{S}$ with the following property: for each configuration σ, and each schedule τ applicable to σ, there is a representative schedule $s(\tau)$ such that: (1) applying $s(\tau)$ to σ results in $\tau(\sigma)$, and (2) $\mathsf{path}(\sigma, s(\tau))$ is generated by a schema from $\mathcal{S}$.

Definition 3.7. *A* schema *is a sequence $\Omega_0, \rho_1, \Omega_1, \dots, \rho_m, \Omega_m$ of alternating contexts and rule sequences. We often write $\{\Omega_0\}\rho_1\{\Omega_1\} \dots \{\Omega_{m-1}\}\rho_m\{\Omega_m\}$ for a schema. A schema with two contexts is called* simple*.*

Given two schemas $S_1 = \Omega_0, \rho_1, \dots, \rho_k, \Omega_k$ and $S_2 = \Omega'_0, \rho'_1, \dots, \rho'_m, \Omega'_m$ with $\Omega_k = \Omega'_0$, we define their *composition* $S_1 \circ S_2$ to be the schema that is obtained by concatenation of the two sequences: $\Omega_0, \rho_1, \dots, \rho_k, \Omega'_0, \rho'_1, \dots, \rho'_m, \Omega'_m$.

Definition 3.8. *Given a configuration σ and a schedule τ applicable to σ, we say that $\mathsf{path}(\sigma, \tau)$ is* generated by a simple schema $\{\Omega\}\,\rho\,\{\Omega'\}$*, if the following hold:*

- *If $\rho = r_1, \dots, r_k$, then there exists a monotonically increasing sequence of indices $i(1), \dots, i(m)$, i.e., $1 \le i(1) < \dots < i(m) \le k$, and there are factors $f_1, \dots, f_m \ge 0$, so that schedule $(r_{i(1)}, f_1), \dots, (r_{i(m)}, f_m) = \tau$.*
- *The first and the last states match the contexts: $\omega(\sigma) = \Omega$ and $\omega(\tau(\sigma)) = \Omega'$.*

In general, we say that $\mathsf{path}(\sigma, \tau)$ is generated by a schema S*, if $S = S_1 \circ \dots \circ S_k$ for simple schemas $S_1, \dots, S_k$ and $\tau = \tau_1 \cdots \tau_k$ such that each $\mathsf{path}(\pi_i(\sigma), \tau_i)$ is generated by the simple schema S_i, for $\pi_i = \tau_1 \cdots \tau_{i-1}$ and $1 \le i \le k$.*

Remark 3.2. Definition 3.8 allows schemas to generate paths that have transitions with zero acceleration factors. Applying a transition with a zero factor to a configuration σ results in the same configuration σ, which corresponds to a stuttering step. This does not affect reachability. In the following, we apply Definition 3.8 to representative paths that may have transitions with zero factors.

Example 3.4. Let us go back to the example of a schema S and a schedule τ' introduced in Equations (3.1) and (3.2) in Section 3.1.2. It is easy to see that schema S can be decomposed into four simple schemas $S_1 \circ \dots \circ S_4$, e.g., $S_1 = \{\}\ r_1, r_1\ \{\varphi_1\}$ and $S_2 = \{\varphi_1\}\ r_1, r_3, r_4, r_4\ \{\varphi_1, \varphi_2\}$. Consider an initial state σ_0 with $n = 5$, $t = f = 1$, $x = y = 0$, $\boldsymbol{\kappa}[\ell_1] = 1$, $\boldsymbol{\kappa}[\ell_2] = 3$, and $\boldsymbol{\kappa}[\ell_i] = 0$ for $i \in \{3, 4, 5\}$. To ensure that $\mathsf{path}(\sigma_0, \tau')$ is generated by schema S, one has to check Definition 3.8 for schemas $S_1, \dots, S_4$ and schedules $(\tau'_1 \cdot t_1)$, $(\tau'_2 \cdot t_2)$, $(\tau'_3 \cdot t_3)$, and τ'_4, respectively.

For instance, $\mathsf{path}(\sigma_0, \tau'_1 \cdot t_1)$ is generated by S_1. Indeed, take the sequence of indices 1 and 2 and the sequence of acceleration factors 1 and 1. The path $\mathsf{path}(\sigma_0, \tau'_1 \cdot t_1)$ ends in the configuration σ_1 that differs from σ_0 in that $\boldsymbol{\kappa}[\ell_2] = 1$, $\boldsymbol{\kappa}[\ell_3] = 2$, and $x = 2$. The contexts $\omega(\sigma_0) = (\{\}, \{\})$ and $\omega(\sigma_1) = (\{\varphi_1\}, \{\})$ match the contexts of schema S_1, as required by Definition 3.8.

Similarly, $\mathsf{path}(\sigma_1, \tau'_2 \cdot t_2)$ is generated by schema S_2. To see that, compare the contexts and use the index sequence 1, 2, 4, and unary acceleration factors. $\triangleleft$

The *language of a schema S* — denoted with $\mathcal{L}(S)$ — is the set of all paths generated by S. For a set of configurations $C \subseteq \Sigma$ and a set of schemas $\mathcal{S}$, we define the set $\mathsf{Reach}(C, \mathcal{S})$

to contain all configurations reachable from C via the paths generated by the schemas from $\mathcal{S}$, i.e.,

$$\mathsf{Reach}(C, \mathcal{S}) = \{\tau(\sigma) \mid \sigma \in C,\ \exists S \in \mathcal{S}.\ \mathsf{path}(\sigma, \tau) \in \mathcal{L}(S)\}.$$

We say that a set $\mathcal{S}$ of schemas is *complete*, if for every set of configurations $C \subseteq \Sigma$ it is the case that the set of all states reachable from C via the paths generated by the schemas from $\mathcal{S}$, is exactly the set of all possible states reachable from C. Formally,

$$\forall C \subseteq \Sigma.\ \{\tau(\sigma) \mid \sigma \in C,\ \tau \text{ is applicable to } \sigma\} = \mathsf{Reach}(C, \mathcal{S}).$$

In [KVW17], a quantity $\mathcal{C}$ has been introduced that depends on the number of conditions in a TA. It has been shown that for every configuration σ and every schedule τ applicable to σ, there is a schedule τ' of length at most $d = |\mathcal{R}| \cdot (\mathcal{C} + 1) + \mathcal{C}$ that is also applicable to σ and results in $\tau(\sigma)$ [KVW17, Theorem. 8]. Hence, by enumerating all sequences of rules of length up to d, one can construct a complete set of schemas:

Theorem 3.4. *For a threshold automaton, there is a complete schema set $\mathcal{S}_d$ of cardinality $|\mathcal{R}|^{|\mathcal{R}| \cdot (\mathcal{C}+1)+\mathcal{C}}$.*

Although the set $\mathcal{S}_d$ is finite, enumerating all its elements is impractical. We show that there is a complete set of schemas whose cardinality solely depends on the number of guards that syntactically occur in the TA. These numbers $|\Phi^{\mathrm{rise}}|$ and $|\Phi^{\mathrm{fall}}|$ are in practice much smaller than the number of rules $|\mathcal{R}|$:

Theorem 3.5. *For all threshold automata, there exists a complete schema set of cardinality at most $(|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|)!$. In this set, the length of each schema does not exceed $(3 \cdot |\Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}}| + 2) \cdot |\mathcal{R}|$.*

Proof Sketch: In the following sections we prove the ingredients of the following argument for the theorem: Construct the set Z of all maximal monotonically increasing sequences of contexts. From Proposition 3.1, we know that there are at most $(|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|)!$ maximal monotonically increasing sequences of contexts. Therefore, $|Z| \leq (|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|)!$. Then, for each sequence $z \in Z$, we do the following:

(1) We show that for each configuration σ and each schedule τ applicable to σ and consistent with the sequence z, there is a schedule $s(\tau)$ that has a specific structure, and is also applicable to σ. We call $s(\tau)$ the representative of τ. We introduce and formally define this specific structure of representative schedules in Sections 3.4–3.6. We prove existence and properties of the representative schedule in Theorem 3.14 (Section 3.6). Before that we consider special cases: when all rules of a schedule belong to the same looplet (Theorem 3.9, Section 3.4), and when a schedule is steady (Theorem 3.11, Section 3.5).

(2) Next we construct a schema (for the sequence z) and show that it generates all paths of all schedules $s(\tau)$ found in (1). The length of the schema is at most $(3 \cdot (|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}|) + 2) \cdot |\mathcal{R}|$. This is shown in Theorem 3.15 (Section 3.6).

Theorem 3.5 follows from the above mentioned theorems, which we prove in the following sections. Auxiliary lemmas and technical proofs of Section 3.4 and Section 3.5 are either only sketched or omitted, but the detailed proofs can be found in Section 3.8 and Section 3.9. □

Remark 3.3. Let us stress the difference between [KVW17] and this work. From the work of [KVW17], it follows that in order to check correctness of a TA it is sufficient to check only the schedules of bounded length $d(\mathsf{TA})$. The bound $d(\mathsf{TA})$ does not depend on the parameters, and can be computed for each TA. The proofs in [KVW17] demonstrate that every schedule longer than $d(\mathsf{TA})$ can be transformed into an "equivalent" representative schedule, whose length is bounded by $d(\mathsf{TA})$. Consequently, one can treat every schedule of length up to $d(\mathsf{TA})$ as its own representative schedule. Similar reasoning does not apply to the schemas we construct here: (i) We construct a complete set of schemas, whose cardinality is substantially smaller than $|\mathcal{S}_d|$, and (ii) the schemas constructed in this chapter can be twice as long as the schemas in $\mathcal{S}_d$.

As discussed in Remark 3.1, the looplets in our case studies are typically either singleton looplets or looplets of size two. In fact, most of our benchmarks have singleton looplets only, and thus their threshold automata can be reduced to directed acyclic graphs. The theoretical constructs of Section 3.4.2 are presented for the more general case of looplets of any size. For most of the benchmarks — the ones not using failure detectors — we need only the simple construction laid out in Section 3.4.1.

3.4 Case I: One Context and One Looplet

We show that for each schedule that uses only the rules from a fixed looplet and does not change its context, there exists a representative schedule of bounded length that reaches the same final state. The goal is to construct a single schema per looplet. The technical challenge is that this *single* schema must generate representative schedules for *all* possible schedules, where, intuitively, processes may move arbitrarily between all local states in the looplet. As a consequence, the rules that appear in the representative schedules can differ from the rules that appear in the arbitrary schedules visiting a looplet.

We fix a threshold automaton, a context Ω, a configuration σ with $\omega(\sigma) = \Omega$, a looplet c, and a schedule τ applicable to σ and using only rules from c. We then construct the representative schedule $\mathsf{crep}_c[\sigma, \tau]$ and the schema $\mathsf{cschema}_c$.

The construction of $\mathsf{crep}_c[\sigma, \tau]$ for the case when $|c| = 1$ is given in Section 3.4.1, and for the case when $|c| > 1$, in Section 3.4.2.

We show in Section 3.4.3 that these constructions give us a schedule that has the desired properties: it reaches the same final state as the given schedule τ, and its length does not exceed $2 \cdot |c|$. Technical details can be found in Section 3.8.

Note that in [KVW17], the length of the representative schedule was bounded by $|c|$. However, all representative schedules of a looplet in this section can be generated by a single looplet schema.

3.4.1 Singleton Looplet

Let us consider the case of the looplet $c = \{r\}$ containing only one rule, that is, $|c| = 1$. There is a trivial representative schedule of a single transition:

Lemma 3.6. *Given a threshold automaton, a configuration σ, and a schedule $\tau = (r, f_1)$, $\dots$, (r, f_m) applicable to σ, one of the two schedules is also applicable to σ and results in $\tau(\sigma)$: either schedule $(r, f_1 + \ldots + f_m)$, or schedule $(r, 0)$.*

A complete proof of Lemma 3.6 can be found in Section 3.8.

Consequently, when c has a single rule r, for configuration σ and a schedule $\tau = (r, f_1), \ldots, (r, f_m)$, Lemma 3.6 allows us to take the singleton schedule (r, f) as $\mathsf{crep}_c[\sigma, \tau]$ and to take the singleton schema $\{\Omega\}\ r\ \{\Omega\}$ as $\mathsf{cschema}_c$. The factor f is either $f_1 + \ldots + f_m$ or zero.

3.4.2 Non-singleton Looplet

Next we focus on non-singleton looplets. Thus, we assume that $|c| > 1$. Our construction is based on two directed trees, whose undirected versions are spanning trees, sharing the same root. In order to find a representative of a steady schedule τ which leads from σ to $\tau(\sigma)$, we determine for each local state how many processes have to move in or out of the state, and then we move them along the edges of the trees. First, we give the definitions of such trees, and then we show how to use them to construct the representative schedules and the schema.

Spanning out-trees and in-trees. We construct the *underlying graph of looplet c*, that is, a directed graph G_c, whose vertices consist of local states that appear as components *from* or *to* of the rules from c, and the edges are the rules of c. More precisely, we construct a directed graph $G_c = (V_c, E_c, L_c)$, whose edges from E_c are labeled by function $L_c : E_c \to c$ with the rules of c, as follows:

$$\begin{aligned} V_c &= \{\ell \mid \exists r \in c,\ r.\mathit{to} = \ell \vee r.\mathit{from} = \ell\}, \\ E_c &= \{(\ell, \ell') \mid \exists r \in c,\ r.\mathit{from} = \ell,\ r.\mathit{to} = \ell'\}, \\ L_c((\ell, \ell')) &= r, \text{ if } r.\mathit{from} = \ell,\ r.\mathit{to} = \ell' \text{ for } (\ell, \ell') \in E_c \text{ and } r \in c. \end{aligned}$$

Lemma 3.7. *Given a threshold automaton and a non-singleton looplet $c \in \mathcal{R}/{\sim}$, graph G_c is non-empty and strongly connected.*

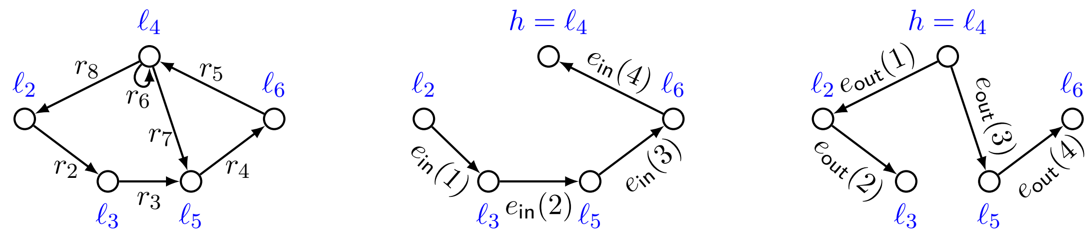


Figure 3.4: The underlying graph of the looplet c_2 of the threshold automaton from Example 3.2 and Figure 3.2 (left), together with trees T_{in} (middle) and T_{out} (right).

Proof. As, $|c| > 1$ and thus $E_c \geq 2$, graph G_c is non-empty. To prove that G_c is strongly connected, we consider a pair of rules $r_1, r_2 \in c$. By the definition of a looplet, it holds that $r_1 \prec_P^+ r_2$ and $r_2 \prec_P^+ r_1$. Thus, there is a path in G_c from $r_1.\mathit{to}$ to $r_2.\mathit{from}$, and there is a path in G_c from $r_2.\mathit{to}$ to $r_1.\mathit{from}$. As r_1 and r_2 correspond to some edges in G_c, there is a cycle that contains the vertices $r_1.\mathit{from}$, $r_1.\mathit{to}$, $r_2.\mathit{from}$, and $r_2.\mathit{to}$. Thus, graph G_c is strongly connected. □

As G_c is non-empty and strongly connected, we can fix an arbitrary node $h \in V_c$ — called a *hub* — and construct two directed trees with the roots at h, whose undirected versions are spanning trees of the undirected version of G_c. These are two subgraphs of G_c: a directed tree $T_{\mathsf{out}} = (V_c, E_{\mathsf{out}})$, whose edges $E_{\mathsf{out}} \subseteq E_c$ are *pointing away from* h (out-tree); a directed tree $T_{\mathsf{in}} = (V_c, E_{\mathsf{in}})$, whose edges $E_{\mathsf{in}} \subseteq E_c$ are *pointing to* h (in-tree). For every node $v \in V_c \setminus \{h\}$, it holds that $|\{u\colon (u, v) \in E_{\mathsf{out}}\}| = 1$ and $|\{w\colon (v, w) \in E_{\mathsf{in}}\}| = 1$.

Further, we fix a topological order $\preceq_{\mathsf{in}}$ on the edges of tree T_{in}. More precisely, $\preceq_{\mathsf{in}}$ is such a partial order on E_{in} that for each pair of adjacent edges $(\ell, \ell'), (\ell', \ell'') \in E_{\mathsf{in}}$, it holds that $(\ell, \ell') \preceq_{\mathsf{in}} (\ell', \ell'')$. In the same way, we fix a topological order $\preceq_{\mathsf{out}}$ on the edges of tree T_{out}.

Example 3.5. Consider again the threshold automaton from Example 3.2 and Figure 3.2. We construct trees T_{in} and T_{out} for looplet c_2, shown in Figure 3.4.

The underlying graph has the set of vertices $V_c = \{\ell_2, \ell_3, \ell_4, \ell_5, \ell_6\}$, and the set of edges is $E_c = \{(\ell_2, \ell_3), (\ell_3, \ell_5), (\ell_5, \ell_6), (\ell_6, \ell_4), (\ell_4, \ell_4), (\ell_4, \ell_5), (\ell_4, \ell_2)\}$. Fix ℓ_4 as a hub. We can fix a linear order $\preceq_{\mathsf{in}}$ such that $(\ell_2, \ell_3) \preceq_{\mathsf{in}} (\ell_3, \ell_5) \preceq_{\mathsf{in}} (\ell_5, \ell_6) \preceq_{\mathsf{in}} (\ell_6, \ell_4)$, and a linear order $\preceq_{\mathsf{out}}$ such that $(\ell_4, \ell_2) \preceq_{\mathsf{out}} (\ell_2, \ell_3) \preceq_{\mathsf{out}} (\ell_4, \ell_5) \preceq_{\mathsf{out}} (\ell_5, \ell_6)$.

Note that for the chosen hub ℓ_4 and this specific example, T_{in} and $\preceq_{\mathsf{in}}$ are uniquely defined, while an out-tree can be different from T_{out} from our Figure 3.4 (the rules r_8, r_2, r_3, r_4 constitute a different tree from the same hub). Because out-tree T_{out} is not a chain, several linear orders different from $\preceq_{\mathsf{out}}$ can be chosen, e.g., $(\ell_4, \ell_2) \preceq_{\mathsf{out}} (\ell_4, \ell_5) \preceq_{\mathsf{out}} (\ell_2, \ell_3) \preceq_{\mathsf{out}} (\ell_5, \ell_6)$. ◁

Representatives of non-singleton looplets. Using these trees, we show how to construct a representative $\mathsf{crep}_c[\sigma, \tau]$ of a schedule τ applicable to σ with $\sigma' = \tau(\sigma)$. For a configuration σ and a schedule τ applicable to σ, consider the trees T_{in} and T_{out}. We construct two sequences: the sequence $e_{\mathsf{in}}(1), \dots, e_{\mathsf{in}}(|E_{\mathsf{in}}|)$ of all edges of T_{in} following the order $\preceq_{\mathsf{in}}$, i.e., if $e_{\mathsf{in}}(i) \preceq_{\mathsf{in}} e_{\mathsf{in}}(j)$, then $i \le j$; the sequence $e_{\mathsf{out}}(1), \dots, e_{\mathsf{out}}(|E_{\mathsf{out}}|)$ of all edges of T_{out} following the order $\preceq_{\mathsf{out}}$. Further, we define the sequence of rules $r_{\mathsf{in}}(1), \dots, r_{\mathsf{in}}(|E_{\mathsf{in}}|)$ with $r_{\mathsf{in}}(i) = L_c(e_{\mathsf{in}}(i))$ for $1 \le i \le |E_{\mathsf{in}}|$, and the sequence of rules $r_{\mathsf{out}}(1), \dots, r_{\mathsf{out}}(|E_{\mathsf{out}}|)$ with $r_{\mathsf{out}}(i) = L_c(e_{\mathsf{out}}(i))$ for $1 \le i \le |E_{\mathsf{out}}|$. Using σ and $\sigma' = \tau(\sigma)$, we define:

$$\delta_{\mathsf{in}}(i) = \sigma.\boldsymbol{\kappa}[f] - \sigma'.\boldsymbol{\kappa}[f], \text{ for } f = r_{\mathsf{in}}(i).\mathit{from} \text{ and } 1 \le i \le |E_{\mathsf{in}}|,$$
$$\delta_{\mathsf{out}}(j) = \sigma'.\boldsymbol{\kappa}[t] - \sigma.\boldsymbol{\kappa}[t], \text{ for } t = r_{\mathsf{out}}(j).\mathit{to} \text{ and } 1 \le j \le |E_{\mathsf{out}}|.$$

If $\delta_{\mathsf{in}}(i) \ge 0$, then $\delta_{\mathsf{in}}(i)$ processes should leave the local state $r_{\mathsf{in}}(i).\mathit{from}$ towards the hub, and they do it exclusively using the edge $e_{\mathsf{in}}(i)$. If $\delta_{\mathsf{out}}(j) \ge 0$, then $\delta_{\mathsf{out}}(j)$ processes should reach the state $r_{\mathsf{out}}(j).\mathit{to}$ from the hub, and they do it exclusively using the edge $e_{\mathsf{out}}(j)$. The negative values of $\delta_{\mathsf{in}}(i)$ and $\delta_{\mathsf{out}}(j)$ do not play any role in our construction, and thus, we use $\max(\delta_{\mathsf{in}}(i), 0)$ and $\max(\delta_{\mathsf{out}}(j), 0)$.

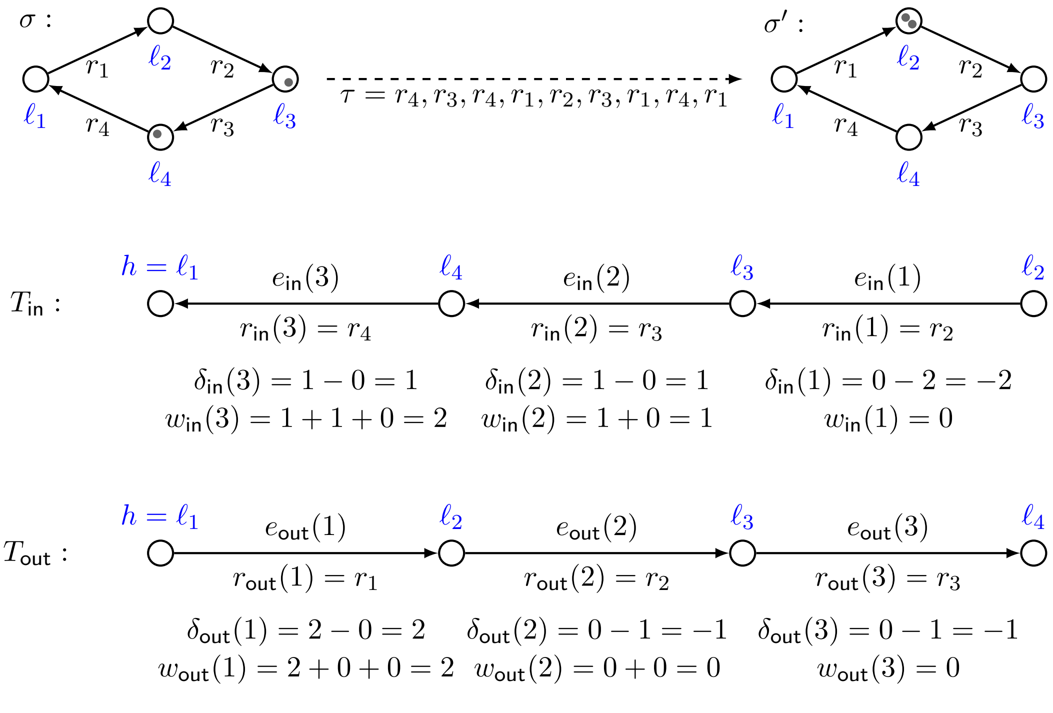
The main idea of the representative construction is as follows: First, we fire the sequence of rules $r_{\mathsf{in}}(1), \dots, r_{\mathsf{in}}(k)$ to collect sufficiently many processes in the hub. Then, we fire the sequence of rules $r_{\mathsf{out}}(1), \dots, r_{\mathsf{out}}(k)$ to distribute the required number of processes from the hub. As a result, for each location ℓ in the graph, if $\sigma[\ell] > \sigma'[\ell]$, then the processes are transferred from ℓ to the other locations, and if $\sigma[\ell] < \sigma'[\ell]$, additional processes arrive at ℓ. Using $\delta_{\mathsf{in}}(i)$ and $\delta_{\mathsf{out}}(i)$, we define the acceleration factors for each rule as follows:

$$w_{\mathsf{in}}(i) = \sum_{j\,:\; e_{\mathsf{in}}(j) \preceq_{\mathsf{in}} e_{\mathsf{in}}(i)} \max(\delta_{\mathsf{in}}(j), 0) \text{ and}$$
$$w_{\mathsf{out}}(i) = \sum_{j\,:\; e_{\mathsf{out}}(i) \preceq_{\mathsf{out}} e_{\mathsf{out}}(j)} \max(\delta_{\mathsf{out}}(j), 0).$$

Finally, we construct the schedule $\mathsf{crep}_c[\sigma, \tau]$ as follows:

$$\mathsf{crep}_c[\sigma, \tau] = (r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \dots, (r_{\mathsf{in}}(|E_{\mathsf{in}}|), w_{\mathsf{in}}(|E_{\mathsf{in}}|)),$$
$$(r_{\mathsf{out}}(1), w_{\mathsf{out}}(1)), \dots, (r_{\mathsf{out}}(|E_{\mathsf{out}}|), w_{\mathsf{out}}(|E_{\mathsf{out}}|)). \quad (3.7)$$

Example 3.6. Consider the TA shown in Figure 3.5, with four local states and four rules. Let c be the four-element looplet that contains the rules r_1, r_2, r_3, and r_4, and let τ be the schedule $\tau = (r_4, 1), (r_3, 1), (r_4, 1), (r_1, 1), (r_2, 1), (r_3, 1), (r_1, 1), (r_4, 1), (r_1, 1)$, which uses the rules of the looplet c. Consider a configuration σ with $\sigma.\boldsymbol{\kappa}[\ell_3] = \sigma.\boldsymbol{\kappa}[\ell_4] = 1$, and $\sigma.\boldsymbol{\kappa}[\ell_1] = \sigma.\boldsymbol{\kappa}[\ell_2] = 0$. The final configuration $\sigma' = \tau(\sigma)$ has the following properties: $\sigma'.\boldsymbol{\kappa}[\ell_2] = 2$ and $\sigma'.\boldsymbol{\kappa}[\ell_1] = \sigma'.\boldsymbol{\kappa}[\ell_3] = \sigma'.\boldsymbol{\kappa}[\ell_4] = 0$. By comparing σ and σ', we notice



σ :
ℓ_1
ℓ_2
ℓ_3
ℓ_4
r_1
r_2
r_3
r_4
$\tau = r_4, r_3, r_4, r_1, r_2, r_3, r_1, r_4, r_1$
σ' :
T_{in} :
$h = \ell_1$
$e_{\mathsf{in}}(3)$
$e_{\mathsf{in}}(2)$
$e_{\mathsf{in}}(1)$
$r_{\mathsf{in}}(3) = r_4$
$r_{\mathsf{in}}(2) = r_3$
$r_{\mathsf{in}}(1) = r_2$
$\delta_{\mathsf{in}}(3) = 1 - 0 = 1$
$w_{\mathsf{in}}(3) = 1 + 1 + 0 = 2$
$\delta_{\mathsf{in}}(2) = 1 - 0 = 1$
$w_{\mathsf{in}}(2) = 1 + 0 = 1$
$\delta_{\mathsf{in}}(1) = 0 - 2 = -2$
$w_{\mathsf{in}}(1) = 0$
T_{out} :
$e_{\mathsf{out}}(1)$
$e_{\mathsf{out}}(2)$
$e_{\mathsf{out}}(3)$
$r_{\mathsf{out}}(1) = r_1$
$r_{\mathsf{out}}(2) = r_2$
$r_{\mathsf{out}}(3) = r_3$
$\delta_{\mathsf{out}}(1) = 2 - 0 = 2$
$w_{\mathsf{out}}(1) = 2 + 0 + 0 = 2$
$\delta_{\mathsf{out}}(2) = 0 - 1 = -1$
$w_{\mathsf{out}}(2) = 0 + 0 = 0$
$\delta_{\mathsf{out}}(3) = 0 - 1 = -1$
$w_{\mathsf{out}}(3) = 0$

Figure 3.5: Construction of T_{in} and T_{out} for the four-element cycle, following Example 3.6.

that one process should move from ℓ_3 to ℓ_2, and one from ℓ_4 to ℓ_2. We will now show how this is achieved by our construction.

For constructing the representative schedule $\mathsf{crep}_c[\sigma, \tau]$, we first define trees T_{in} and T_{out}. If we choose ℓ_1 to be the hub, then by following the construction we obtain that $E_{\mathsf{in}} = \{(\ell_4, \ell_1), (\ell_3, \ell_4), (\ell_2, \ell_3)\}$, and thus the order is $(\ell_2, \ell_3) \preceq_{\mathsf{in}} (\ell_3, \ell_4) \preceq_{\mathsf{in}} (\ell_4, \ell_1)$. This explains why $e_{\mathsf{in}}(1) = (\ell_2, \ell_3)$, $e_{\mathsf{in}}(2) = (\ell_3, \ell_4)$, $e_{\mathsf{in}}(3) = (\ell_4, \ell_1)$. By calculating $\delta_{\mathsf{in}}(i)$ for every $i \in \{1, 2, 3\}$, we see that $\delta_{\mathsf{in}}(2) = 1$ and $\delta_{\mathsf{in}}(3) = 1$ are positive. Consequently, two processes go to the hub: one from $r_{\mathsf{in}}(2).\mathit{from} = \ell_3$ and one from $r_{\mathsf{in}}(3).\mathit{from} = \ell_4$. The coefficients w_{in} give us acceleration factors for all rules.

Similarly, we obtain $E_{\mathsf{out}} = \{(\ell_1, \ell_2), (\ell_2, \ell_3), (\ell_3, \ell_4)\}$, and the order must be $(\ell_1, \ell_2) \preceq_{\mathsf{out}} (\ell_2, \ell_3) \preceq_{\mathsf{out}} (\ell_3, \ell_4)$. Thus, we have $e_{\mathsf{out}}(1) = (\ell_1, \ell_2)$, $e_{\mathsf{in}}(2) = (\ell_2, \ell_3)$, and $e_{\mathsf{out}}(3) = (\ell_3, \ell_4)$. Here only $\delta_{\mathsf{out}}(1) = 2$ has a positive value, and hence, two processes should move from hub to the local state $r_{\mathsf{out}}(1).\mathit{to} = \ell_2$. To achieve this, the acceleration factor of every rule $r_{\mathsf{out}}(i)$, $1 \leq i \leq 3$, must be $w_{\mathsf{out}}(i)$.

Therefore, by Equation (3.7), the representative schedule is

$$\mathsf{crep}_c[\sigma, \tau] = (r_2, 0), (r_3, 1), (r_4, 2), (r_1, 2), (r_2, 0), (r_3, 0).$$

Choosing another hub gives us another representative. For each hub, the representative is not longer than $2|c| = 8$, and leads to σ' when applied to σ. $\triangleleft$

This construction is defined in such a way that the following lemma holds. As its proof is rather technical, we present it in Section 3.8.

Lemma 3.8. *Fix a threshold automaton* TA*, a context* Ω*, and a non-singleton looplet* c *of the slice* $\mathsf{TA}|_{\Omega}$*. Let* σ *be a configuration of* TA *and let* τ *be a schedule that is contained in* c *and it is applicable to* σ*. Then the schedule* $\mathsf{crep}_c[\sigma,\tau]$ *has the following properties:*

a) $\mathsf{crep}_c[\sigma,\tau]$ *is applicable to* σ*, and*

b) $\mathsf{crep}_c[\sigma,\tau]$ *results in* $\tau(\sigma)$ *when applied to* σ*.*

3.4.3 Representatives for One Context and One Looplet

We now summarize results from Sections 3.4.1 and 3.4.2, giving the representative of a schedule τ in the case when τ uses only the rules from one looplet, and does not change its context. If the given looplet consists of a single rule, the construction is given in Section 3.4.1, and otherwise in Section 3.4.2. We show that these constructions indeed give us a schedule of bounded length, that reaches the same state as τ.

In the following, given a threshold automaton TA and a looplet c, we will say that a *schedule* $\tau = t_1, \ldots, t_n$ *is contained in* c, if $[t_i.rule] = c$ for $1 \le i \le n$.

Theorem 3.9. *Fix a threshold automaton, a context* Ω*, and a looplet* c *in the slice* $\mathsf{TA}|_{\Omega}$*. Let* σ *be a configuration and let* τ *be a steady schedule contained in* c *and applicable to* σ*. There exists a representative schedule* $\mathsf{crep}_c[\sigma,\tau]$ *with the following properties:*

a) schedule $\mathsf{crep}_c[\sigma,\tau]$ *is applicable to* σ*, and* $\mathsf{crep}_c[\sigma,\tau](\sigma) = \tau(\sigma)$*,*

b) the rule of each transition t *in* $\mathsf{crep}_c[\sigma,\tau]$ *belongs to* c*, that is,* $[t.rule] = c$*,*

c) schedule $\mathsf{crep}_c[\sigma,\tau]$ *is not longer than* $2 \cdot |c|$*.*

Proof. If $|c| = 1$, then we use a single accelerated transition or the empty schedule as representative, as described in Lemma 3.6.

If $|c| > 1$, we construct the representative as in Section 3.4.2 and Equation (3.7), so that by Lemma 3.8 property a) follows. For every edge $e \in E_c$, the rule $L_c(e)$ belongs to c, and thus $\mathsf{crep}_c[\sigma,\tau]$ satisfies property b). As $|E_{\mathsf{in}}| \le |c|$ and $|E_{\mathsf{out}}| \le |c|$, we conclude that $|\mathsf{crep}_c[\sigma,\tau]| \le 2 \cdot |c|$, and thus property c) is also satisfied. From this and Lemma 3.8, we conclude that $\mathsf{crep}_c[\sigma,\tau]$ is the required representative schedule. □

Theorem 3.9 gives us a way to construct schemas that generate all representatives of the schedules contained in a looplet:

Theorem 3.10. *Fix a threshold automaton* TA*, a context* Ω*, and a looplet* c *in the slice* $\mathsf{TA}|_{\Omega}$*. There exists a schema* $\mathsf{cschema}_c$ *with the following properties:*

Fix an arbitrary configuration σ *and a steady schedule* τ *applicable to* σ *that is contained in* c*. Let* $\tau' = \mathsf{crep}_c[\sigma, \tau]$ *be the representative schedule of* τ*, from Theorem 3.9. Then,* $\mathsf{path}(\sigma, \tau')$ *is generated by* $\mathsf{cschema}_c$*. Moreover, the length of* $\mathsf{cschema}_c$ *is at most* $2 \cdot |c|$*.*

Proof. Note that $\tau' = \mathsf{crep}_c[\sigma, \tau]$ can be constructed in two different ways depending on the looplet c.

If $|c| = 1$, then by Lemma 3.6 we have that $\tau' = (r, f)$ for a rule $r \in c$ and a factor $f \in \mathbb{N}_0$. In this case we construct $\mathsf{cschema}_c$ to be

$$\mathsf{cschema}_c = \{\Omega\}\, r\, \{\Omega\}.$$

It is easy to see that $\mathsf{path}(\sigma, \tau')$ is generated by $\mathsf{cschema}_c$, as well as that the length of $\mathsf{cschema}_c$ is exactly 1, that is less than $2 \cdot |c|$.

If $|c| > 1$, then we use the trees T_{in} and T_{out} to construct the schema $\mathsf{cschema}_c$ as follows:

$$\mathsf{cschema}_c = \{\Omega\}\, r_{\mathsf{in}}(1) \cdots r_{\mathsf{in}}(|E_{\mathsf{in}}|) \cdot r_{\mathsf{out}}(1) \cdots r_{\mathsf{out}}(|E_{\mathsf{out}}|)\, \{\Omega\}. \tag{3.8}$$

Since for an arbitrary configuration σ and a schedule τ, we use the same sequence of edges in Equations (3.7) and (3.8) to construct $\mathsf{crep}_c[\sigma, \tau]$ and $\mathsf{cschema}_c$, the schema $\mathsf{cschema}_c$ generates all paths of the representative schedules, and its length is at most $2 \cdot |c|$. $\square$

3.5 Case II: One Context and Multiple Looplets

In this section, we show that for each steady schedule, there exists a representative steady schedule of bounded length that reaches the same final state.

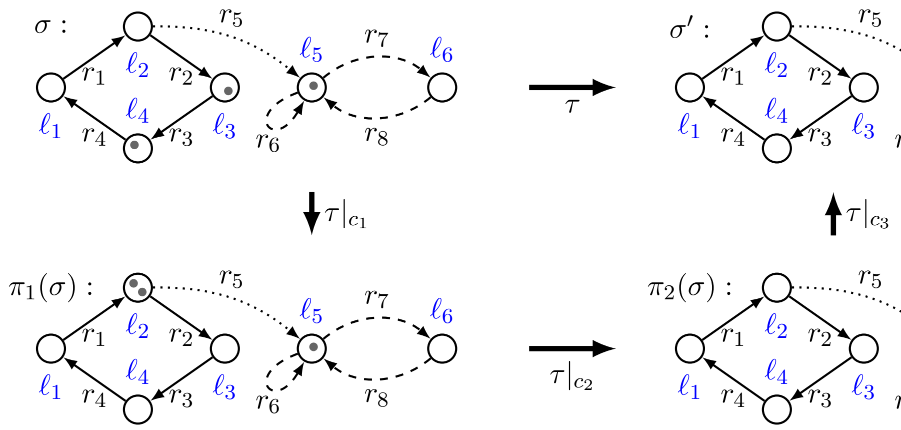
Theorem 3.11. *Fix a threshold automaton and a context* Ω*. For every configuration* σ *with* $\omega(\sigma) = \Omega$ *and every steady schedule* τ *applicable to* σ*, there exists a steady schedule* $\mathsf{srep}[\sigma, \tau]$ *with the following properties:*

a) $\mathsf{srep}[\sigma, \tau]$ *is applicable to* σ*, and* $\mathsf{srep}[\sigma, \tau](\sigma) = \tau(\sigma)$*,*

b) $|\mathsf{srep}[\sigma, \tau]| \leq 2 \cdot |(\mathcal{R}|_{\Omega})|$

Let us first explain the construction of the representative $\mathsf{srep}[\sigma, \tau]$, and leave the proof of this theorem for later in this section.

To construct a representative schedule, we fix a context Ω of a TA, a configuration σ with $\omega(\sigma) = \Omega$, and a steady schedule τ applicable to σ. The key notion in our construction is a projection of a schedule on a set of looplets:



σ :
σ' :
$\pi_1(\sigma)$:
$\pi_2(\sigma)$:
τ
$\tau|_{c_1}$
$\tau|_{c_2}$
$\tau|_{c_3}$
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8
ℓ_1 ℓ_2 ℓ_3 ℓ_4 ℓ_5 ℓ_6

Figure 3.6: Threshold automaton and its configurations used in Example 3.7. The number of dots inside a local state ℓ represents the value of its counter $\boldsymbol{\kappa}[\ell]$.

Definition 3.9. *Let $\tau = t_1, \ldots, t_k$, for $k > 0$, be a schedule, and let C be a set of looplets. Given an increasing sequence of indices $i(1), \ldots, i(m) \in \{1, \ldots, k\}$, where $m \leq k$, i.e., $i(j) < i(j+1)$, for $1 \leq j < m$, a schedule $t_{i(1)} \ldots t_{i(m)}$ is a* projection of τ on C, *if each index $j \in \{1, \ldots, k\}$ belongs to $\{i(1), \ldots, i(m)\}$ if and only if $[t_j.rule] \in C$.*

In fact, each schedule τ has a unique projection on a set C. In the following, we write $\tau|_{c_1,\ldots,c_m}$ to denote the projection of τ on a set $\{c_1, \ldots, c_m\}$.

Provided that $c_1, \ldots, c_m$ are all looplets of the slice $\mathcal{R}|_\Omega$ ordered with respect to $\prec_C^{lin}$, we construct the following sequences of projections on each looplet (note that π_0 is the empty schedule): $\pi_i = \tau|_{c_1} \cdot \ldots \cdot \tau|_{c_i}$, for $0 \leq i \leq m$.

Having defined $\{\pi_i\}_{0 \leq i \leq m}$, we construct the representative $\mathsf{srep}[\sigma, \tau]$ simply as a concatenation of the representatives of each looplet:

$$\mathsf{srep}[\sigma,\tau] = \mathsf{crep}_{c_1}[\pi_0(\sigma), \tau|_{c_1}] \cdot \mathsf{crep}_{c_2}[\pi_1(\sigma), \tau|_{c_2}] \cdot \ldots \cdot \mathsf{crep}_{c_m}[\pi_{m-1}(\sigma), \tau|_{c_m}]$$

Example 3.7. Consider the TA shown in Figure 3.6. It has three looplets, namely $c_1 = \{r_1, r_2, r_3, r_4\}$, $c_2 = \{r_5\}$, $c_3 = \{r_6, r_7, r_8\}$, and the rules are depicted as solid, dotted, and dashed, respectively. These looplets are ordered such that $c_1 \prec_C^{lin} c_2 \prec_C^{lin} c_3$.

Let σ be the upper left configuration in Figure 3.6, with $\boldsymbol{\kappa}[\ell_3] = \boldsymbol{\kappa}[\ell_4] = \boldsymbol{\kappa}[\ell_5] = 1$ and $\boldsymbol{\kappa}[\ell_1] = \boldsymbol{\kappa}[\ell_2] = \boldsymbol{\kappa}[\ell_6] = 0$. Let τ be the schedule $(r_4, 1), (r_6, 1), (r_3, 1), (r_4, 1), (r_1, 1), (r_2, 1), (r_7, 1), (r_3, 1), (r_1, 1), (r_5, 1), (r_7, 1), (r_4, 1), (r_8, 1), (r_1, 1), (r_6, 1), (r_7, 1), (r_5, 1), (r_8, 1), (r_7, 1)$. Note that τ is applicable to σ and that $\tau(\sigma)$ is the configuration σ' from Figure 3.6 top right, i.e. $\boldsymbol{\kappa}[\ell_5] = 1$, $\boldsymbol{\kappa}[\ell_6] = 2$ and $\boldsymbol{\kappa}[\ell_1] = \boldsymbol{\kappa}[\ell_2] = \boldsymbol{\kappa}[\ell_3] = \boldsymbol{\kappa}[\ell_4] = 0$. We construct the representative schedule $\mathsf{srep}[\sigma, \tau]$.

Projection of τ on the looplets c_1, c_2, and c_3, gives us the following schedules:

$$\begin{aligned}
\tau|_{c_1} &= (r_4,1),(r_3,1),(r_4,1),(r_1,1),(r_2,1),(r_3,1),(r_1,1),(r_4,1),(r_1,1),\\
\tau|_{c_2} &= (r_5,1),(r_5,1),\\
\tau|_{c_3} &= (r_6,1),(r_7,1),(r_7,1),(r_8,1),(r_6,1),(r_7,1),(r_8,1),(r_7,1).
\end{aligned}$$

Recall that

$$\mathsf{srep}[\sigma,\tau] = \mathsf{crep}_{c_1}[\pi_0(\sigma),\tau|_{c_1}] \cdot \mathsf{crep}_{c_2}[\pi_1(\sigma),\tau|_{c_2}] \cdot \mathsf{crep}_{c_3}[\pi_2(\sigma),\tau|_{c_3}].$$

In order to construct this schedule, we firstly construct the required configurations. Note that $\pi_0(\sigma) = \sigma$. Then $\pi_1(\sigma) = \tau|_{c_1}(\sigma)$, and this is the configuration from Figure 3.6 lower left, i.e. $\boldsymbol{\kappa}[\ell_2] = 2$, $\boldsymbol{\kappa}[\ell_5] = 1$ and $\boldsymbol{\kappa}[\ell_1] = \boldsymbol{\kappa}[\ell_3] = \boldsymbol{\kappa}[\ell_4] = \boldsymbol{\kappa}[\ell_6] = 0$. Configuration $\pi_2(\sigma) = \tau|_{c_1} \cdot \tau|_{c_2}(\sigma) = \tau|_{c_2}(\pi_1(\sigma))$ is represented on Figure 3.6 lower right, i.e. $\boldsymbol{\kappa}[\ell_5] = 3$ and all other counters are zero.

Section 3.4 deals with the construction of representatives of schedules that contain rules from only one looplet. Recall that construction of $\mathsf{crep}_{c_1}[\pi_0(\sigma),\tau|_{c_1}]$ corresponds to the one from Example 3.6. Thus, we know that

$$\mathsf{crep}_{c_1}[\pi_0(\sigma),\tau|_{c_1}] = (r_2,0),(r_3,1),(r_4,2),(r_1,2),(r_2,0),(r_3,0).$$

As c_2 is a singleton looplet, we use the result of Section 3.4.1. Thus,

$$\mathsf{crep}_{c_2}[\pi_1(\sigma),\tau|_{c_2}] = (r_5,2).$$

Using the result from Section 3.4.2 we obtain that

$$\mathsf{crep}_{c_3}[\pi_2(\sigma),\tau|_{c_3}] = (r_8,0),(r_7,2),$$

and finaly we have the representative for τ that is

$$\mathsf{srep}[\sigma,\tau] = (r_2,0),(r_3,1),(r_4,2),(r_1,2),(r_2,0),(r_3,0),(r_5,2),(r_8,0),(r_7,2),$$

obtained by concatenating the previous three looplet representatives. $\triangleleft$

The following lemma serves for reordering transitions inside a schedule, in a way that respects the order of looplets. Its proof is in Section 3.9.

Lemma 3.12 (Looplet sorting)**.** *Given a threshold automaton, a context Ω, a configuration σ, a steady schedule τ applicable to σ, and a sequence $c_1,\dots,c_m$ of all looplets in the slice $\mathcal{R}|_\Omega$ with the property $c_i \prec_C^{lin} c_j$ for $1 \le i < j \le m$, the following holds:*

a) Schedule $\tau|_{c_1}$ is applicable to the configuration σ.

b) Schedule $\tau|_{c_2,\dots,c_m}$ is applicable to the configuration $\tau|_{c_1}(\sigma)$.

c) Schedule $\tau|_{c_1} \cdot \tau|_{c_2,\dots,c_m}$, when applied to σ, results in configuration $\tau(\sigma)$.

Now we are ready to prove Theorem 3.11 from the beginning of this section.

Proof of Theorem 3.11 By iteratively applying Lemma 3.12, we prove by induction that schedule $\tau|_{c_1} \cdot \ldots \cdot \tau|_{c_m}$ is applicable to σ and results in $\tau(\sigma)$. From Theorem 3.9, we conclude that each schedule $\tau|_{c_i}$ can be replaced by its representative $\mathsf{crep}_{c_i}[\pi_{i-1}(\sigma), \tau|_{c_i}]$. Thus, $\mathsf{srep}[\sigma, \tau]$, which is the concatenation of these representatives, is applicable to σ and results in $\tau(\sigma)$. By Proposition 3.3, schedule $\mathsf{srep}[\sigma, \tau]$ is steady, since $\omega(\sigma) = \omega(\tau(\sigma))$. □

Finally, we show that for a given context, there is a schema that generates all paths of such representative schedules.

Theorem 3.13. *Fix a threshold automaton and a context Ω. Let $c_1, \dots, c_m$ be the sorted sequence of all looplets of the slice $\mathcal{R}|_{\Omega}$, i.e., $c_1 \prec_C^{lin} \dots \prec_C^{lin} c_m$. Schema* $\mathsf{sschema} = \mathsf{cschema}_{c_1} \circ \cdots \circ \mathsf{cschema}_{c_m}$ *has two properties:*

a) For a configuration σ with $\omega(\sigma) = \Omega$ and a steady schedule τ applicable to σ, $\mathsf{path}(\sigma, \tau')$ of the representative $\tau' = \mathsf{srep}[\sigma, \tau]$ is generated by $\mathsf{sschema}$*; and*

b) the length of $\mathsf{sschema}$ *is at most $2 \cdot |(\mathcal{R}|_{\Omega})|$.*

Proof. Fix a configuration σ with $\omega(\sigma) = \Omega$ and a steady schedule τ applicable to σ. As $\mathsf{srep}[\sigma, \tau]$ is a sorted sequence of the looplet representatives, all paths of $\mathsf{srep}[\sigma, \tau]$ are generated by $\mathsf{sschema}$, which is not longer than $2 \cdot |(\mathcal{R}|_{\Omega})|$. □

3.6 Proving the Main Result

Using the results from Sections 3.4 and 3.5, for each configuration and each schedule (without restrictions) we construct a representative schedule.

Theorem 3.14. *Given a threshold automaton, a configuration σ, and a schedule τ applicable to σ, there exists a schedule* $\mathsf{rep}[\sigma, \tau]$ *with the following properties:*

a) $\mathsf{rep}[\sigma, \tau]$ *is applicable to σ, and* $\mathsf{rep}[\sigma, \tau](\sigma) = \tau(\sigma)$*,*

b) $|\mathsf{rep}[\sigma, \tau]| \leq 2 \cdot |\mathcal{R}| \cdot (|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}| + 1) + |\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|$.

Proof. Given a threshold automaton, fix a configuration σ and a schedule τ applicable to σ. Let $\Omega_1, \dots, \Omega_{K+1}$ be the maximal monotonically increasing sequence of contexts such that $\mathsf{path}(\sigma, \tau)$ is consistent with the sequence by Definition 3.5. From Proposition 3.1, the length of the sequence is $K + 1 = |\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}| + 1$. Thus, there are at most K

transitions $t_1^\star, \ldots, t_K^\star$ in τ that change their context, i.e., for $i \in \{1, \ldots, K\}$, it holds $\omega(\sigma_i) \sqsubset \omega(t_i^\star(\sigma_i))$ for $t_i^\star$'s respective state σ_i in τ. Therefore, we can divide τ into $K+1$ steady schedules separated by the transitions $t_1^\star, \ldots, t_K^\star$:

$$\tau = \nu_1 \cdot t_1^\star \cdot \nu_2 \cdots \nu_K \cdot t_K^\star \cdot \nu_{K+1}.$$

Now, the main idea is to replace the steady schedules with their representatives from Theorem 3.11. That is, using $t_1^\star, \ldots, t_K^\star$ and $\nu_1, \ldots, \nu_{K+1}$, we construct the schedules $\rho_1, \ldots, \rho_K$ (by convention, ρ_0 is the empty schedule):

$$\rho_i = \rho_{i-1} \cdot \nu_i \cdot t_i^\star \quad \text{for } 1 \le i \le K.$$

Finally, the representative schedule $\mathsf{rep}[\tau, \sigma]$ is constructed as follows:

$$\mathsf{rep}[\sigma, \nu_1] \quad \cdot \quad t_1^\star \quad \cdot \quad \mathsf{rep}[\rho_1(\sigma), \nu_2] \cdots \mathsf{rep}[\rho_{K-1}(\sigma), \nu_K] \quad \cdot \quad t_K^\star \quad \cdot \quad \mathsf{rep}[\rho_K(\sigma), \nu_{K+1}]$$

From Theorem 3.11, it follows that $\mathsf{rep}[\tau, \sigma]$ is applicable to σ and it results in $\tau(\sigma)$. Moreover, the representative of a steady schedule is not longer than $2|\mathcal{R}|$, which together with K transitions gives us the bound $2|\mathcal{R}|(K+1)+K$. As we have that $K = |\Phi^{\mathrm{rise}}|+|\Phi^{\mathrm{fall}}|$, this gives us the required bound. □

Further, given a maximal monotonically increasing sequence z of contexts, we construct a schema that generates all paths of the schedules consistent with z:

Theorem 3.15. *For a threshold automaton and a monotonically increasing sequence z of contexts, there exists a schema* $\mathsf{schema}(z)$ *that generates all paths of the representative schedules that are consistent with z, and the length of* $\mathsf{schema}(z)$ *does not exceed* $3 \cdot |\mathcal{R}| \cdot (|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|) + 2 \cdot |\mathcal{R}|$.

Proof. Given a threshold automaton, let ρ_{all} be the sequence $r_1, \ldots, r_{|\mathcal{R}|}$ of all rules from $\mathcal{R}$, and let $z = \Omega_0, \ldots, \Omega_m$ be a monotonically increasing sequence of contexts. By the construction in Theorem 3.14, each representative schedule $\mathsf{rep}[\sigma, \tau]$ consists of the representatives of steady schedules terminated with transitions that change the context. Then, for each context Ω_i, for $0 \le i < m$, we compose $\mathsf{sschema}$ and $\{\Omega_i\}\, \rho_{\mathsf{all}}\, \{\Omega_{i+1}\}$. This composition generates the representative of a steady schedule and the transition changing the context from Ω_i to Ω_{i+1}. Consequently, we construct the $\mathsf{schema}(z)$ as follows:

$$(\mathsf{sschema}_{\Omega_0} \circ \{\Omega_0\}\, \rho_{\mathsf{all}}\, \{\Omega_1\}) \circ \ldots \circ (\mathsf{sschema}_{\Omega_{m-1}} \circ \{\Omega_{m-1}\}\, \rho_{\mathsf{all}}\, \{\Omega_m\}) \circ \mathsf{sschema}_{\Omega_m}$$

By inductively applying Theorem 3.13, we prove that $\mathsf{schema}(z)$ generates all paths of schedules $\mathsf{rep}[\sigma, \tau]$ that are consistent with the sequence z. We get the needed bound on the length of $\mathsf{schema}(z)$ by using an argument similar to Theorem 3.14 and by noting that for every context, instead of one rule that is changing it, we add $|\mathcal{R}|$ extra rules. □

3.7 Application of the PARA^2 technique for reachability

3.7.1 Complete Set of Schemas and Optimizations

Our proofs show that the set of schemas is easily computed from the TA: The threshold guards are syntactic parts of the TA, and enable us to directly construct increasing sequences of contexts. To find a slice of the TA for a given context, we filter the rules with unlocked guards, i.e., check whether the context contains the guard. To produce the simple schema of a looplet, we compute a spanning tree over the slice. To construct simple schemas, we do a topological sort over the looplets. For example, it takes just 30 seconds to compute the schemas in our longest experiment that runs for 4 hours. In our tool, the following optimizations have been implemented, that lead to simpler and fewer SMT queries.

Entailment optimization. We say that a guard $\varphi_1 \in \Phi^{\text{rise}}$ *entails* a guard $\varphi_2 \in \Phi^{\text{rise}}$, if for all combinations of parameters $\mathbf{p} \in \mathbf{P}_{RC}$ and shared variables $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$, it holds that $(\mathbf{g}, \mathbf{p}) \models \varphi_1 \rightarrow \varphi_2$. For instance, in our example, $\varphi_3\colon y \geq (2t+1) - f$ entails $\varphi_2\colon y \geq (t+1) - f$. If φ_1 entails φ_2, then we can omit all monotonically increasing sequences that contain a context $(\Omega^{\text{rise}}, \Omega^{\text{fall}})$ with $\varphi_1 \in \Omega^{\text{rise}}$ and $\varphi_2 \notin \Omega^{\text{rise}}$. If the number of schemas before applying this optimization is $m!$ and there are k entailments, then the number of schemas reduces from $m!$ to $(m-k)!$. A similar optimization is introduced for the guards from Φ^{fall}.

Control flow optimization. Based on the proof of Lemma 3.12, we introduce the following optimization for TAs that are directed acyclic graphs (possibly with self loops). We say that a rule $r \in \mathcal{R}$ *may unlock* a guard $\varphi \in \Phi^{\text{rise}}$, if there is a $\mathbf{p} \in \mathbf{P}_{RC}$ and $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$ satisfying: $(\mathbf{g}, \mathbf{p}) \models r.\varphi^{\text{rise}} \wedge r.\varphi^{\text{fall}}$ (the rule is unlocked); $(\mathbf{g}, \mathbf{p}) \not\models \varphi$ (the guard is locked); $(\mathbf{g} + r.\mathbf{u}, \mathbf{p}) \models \varphi$ (the guard is now unlocked).

In our example from Figure 3.1, the rule $r_1\colon \mathit{true} \mapsto x{+}{+}$ may unlock the guard $\varphi_1\colon x \geq \lceil (n+t)/2 \rceil - f$.

Let $\varphi \in \Phi^{\text{rise}}$ be a guard, $r'_1, \ldots, r'_m$ be the rules that use φ, and $r_1, \ldots, r_k$ be the rules that may unlock φ. If $r_i \prec_C^{\mathit{lin}} r'_j$, for $1 \leq i \leq k$ and $1 \leq j \leq m$, then we exclude some sequences of contexts as follows (we call φ *forward-unlockable*). Let $\psi_1, \ldots, \psi_n \in \Phi^{\text{rise}}$ be the guards of $r_1, \ldots, r_k$. Guard φ cannot be unlocked before $\psi_1, \ldots, \psi_n$, and thus we can omit all sequences of contexts, where φ appears in the contexts before $\psi_1, \ldots, \psi_n$. Moreover, as $\psi_1, \ldots, \psi_n$ are the only guards of the rules unlocking φ, we omit the sequences with different combinations of contexts involving φ and the guards from $\Phi^{\text{rise}} \setminus \{\varphi, \psi_1, \ldots, \psi_n\}$. Finally, as the rules $r'_1, \ldots, r'_m$ appear after the rules $r_1, \ldots, r_k$ in the order $\prec_C^{\mathit{lin}}$, the rules $r'_1, \ldots, r'_m$ appear after the rules $r_1, \ldots, r_k$ in a rule sequence of every schema. Thus, we omit the combinations of the contexts involving φ and $\psi_1, \ldots, \psi_n$.

Hence, we add all forward-unlockable guards to the initial context (we still check the guards of the rules in the SMT encoding in Section 3.7.2). If the number of schemas before applying this optimization is $m!$ and there are k forward-unlocking guards, then the number of schemas reduces from $m!$ to $(m-k)!$. A similar optimization is introduced for the guards from Φ^{fall}.

3.7.2 Checking a Schema with SMT

We decompose a schema into a sequence of simple schemas, and encode the simple schemas. Given a simple schema $S = \{\Omega_1\}\, r_1, \dots, r_m\, \{\Omega_2\}$, which contains m rules, we construct an SMT formula such that every model of the formula represents a path from $\mathcal{L}(S)$ — the language of paths generated by schema S — and for every path in $\mathcal{L}(S)$ there is a corresponding model of the formula. Thus, we need to model a path of $m+1$ configurations and m transitions (whose acceleration factors may be 0).

To represent a configuration σ_i, for $0 \le i \le m$, we introduce two vectors of SMT variables: Given the set of local states L and the set of shared variables Γ, a vector $\mathbf{k}^i = (k_1^i, \dots, k_{|L|}^i)$ to represent the process counters, a vector $\mathbf{x}^i = (x_1^i, \dots, x_{|\Gamma|}^i)$ to represent the shared variables. We call the pair $(\mathbf{k}^i, \mathbf{x}^i)$ the *layer* i, for $1 \le i \le m$.

Based on this we encode schemas, for which the sequence of rules $r_1, \dots, r_m$ is fixed. We exploit this in two ways: First, we encode for each layer i the constraints of rule r_i. Second, as this constraint may update only two counters — the processes move from and move to according to the rule — we do not need $|L|$ counter variables per layer, but only encode the two counters per layer that have actually changed. As is a common technique in bounded model checking, the counters that are not changed are "reused" from previous layers in our encoding. By doing so, we encode the schema rules with $|L| + |\Gamma| + m \cdot (2 + |\Gamma|)$ integer variables, $2m$ equations, and inequalities in linear integer arithmetic that represent threshold guards that evaluate to true (at most the number of threshold guards times m of these inequalities).

In the following, we use the notation $[k : m]$ to denote the set $\{k, \dots, m\}$. In order to reuse the variables from the previous layers, we introduce a function $\upsilon : L \times [0 : m] \to [0 : m]$ that for a layer $i \in [0 : m]$ and a local state $\ell \in L$, gives the largest number $j \le i$ of the layer, where the counter k_ℓ^j is updated:

$$\upsilon(\ell, i) = \begin{cases} i, & \text{if } i = 0 \lor \ell \in \{r_i.\mathit{from}, r_i.\mathit{to}\} \\ \upsilon(\ell, i-1), & \text{otherwise.} \end{cases}$$

Having defined layers, we encode: the effect of rules on counters and shared variables (in formulas M and U below), the effect of rules on the configuration (T), restrictions imposed by contexts (C), and, finally, the reachability question.

To represent m transitions, for each transition $i \in [1 : m]$, we introduce a non-negative variable δ^i for the acceleration factor, and define two formulas: formula $M^\ell(i-1, i)$ to express the update of the counter of local state $\ell \in L$, and formula $U^x(i-1, i)$ to represent the update of the shared variable $x \in \Gamma$:

$$M^{\ell}(i-1,i) \;\equiv\; \begin{cases} k_{\ell}^{i} = k_{\ell}^{v(\ell,i-1)} + \delta^{i}, & \text{for } \ell = r_i.\mathit{to} \text{ and } i \in [1:m] \\ k_{\ell}^{i} = k_{\ell}^{v(\ell,i-1)} - \delta^{i}, & \text{for } \ell = r_i.\mathit{from} \text{ and } i \in [1:m] \\ \mathit{true}, & \text{otherwise} \end{cases}$$

$$U^{x}(i-1,i\,) \equiv \begin{cases} x^{i} = x^{i-1} + \delta^{i} \cdot u, & \text{if } u = r_i.\mathbf{u}[j] > 0, \\ \mathit{true}, & \text{otherwise.} \end{cases}$$

The formula $T(i-1,i)$ collects all constraints by the rule r_i:

$$T(i-1,i) \;\equiv\; \bigwedge_{\ell \in L} M^{\ell}(i-1,i) \wedge \bigwedge_{x \in \Gamma} U^{x}(i-1,i).$$

For a formula φ, we denote by $\varphi[\mathbf{x}^i]$ the formula, where each variable $x \in \Gamma$ is substituted with x^i. Then, given a context $\Omega = (\Omega^{\mathrm{rise}}, \Omega^{\mathrm{fall}})$, a formula $C_{\Omega}(i)$ adds the constraints of the context Ω on the layer i:

$$C_{\Omega}(i) \;\equiv\; \bigwedge_{\varphi \in \Omega^{\mathrm{rise}}} \varphi[\mathbf{x}^i] \wedge \bigwedge_{\varphi \in \Phi^{\mathrm{rise}} \setminus \Omega^{\mathrm{rise}}} \neg\varphi[\mathbf{x}^i] \wedge \bigwedge_{\varphi \in \Omega^{\mathrm{fall}}} \neg\varphi[\mathbf{x}^i] \wedge \bigwedge_{\varphi \in \Phi^{\mathrm{fall}} \setminus \Omega^{\mathrm{fall}}} \varphi[\mathbf{x}^i].$$

Finally, the formula $C_{\Omega_1}(0) \wedge T(0,1) \wedge \dots \wedge T(m-1,m) \wedge C_{\Omega_2}(m)$ captures all the constraints of the schema $S = \{\Omega_1\}\, r_1, \dots, r_m\, \{\Omega_2\}$, and thus, its models correspond to the paths of schedules that are generated by S.

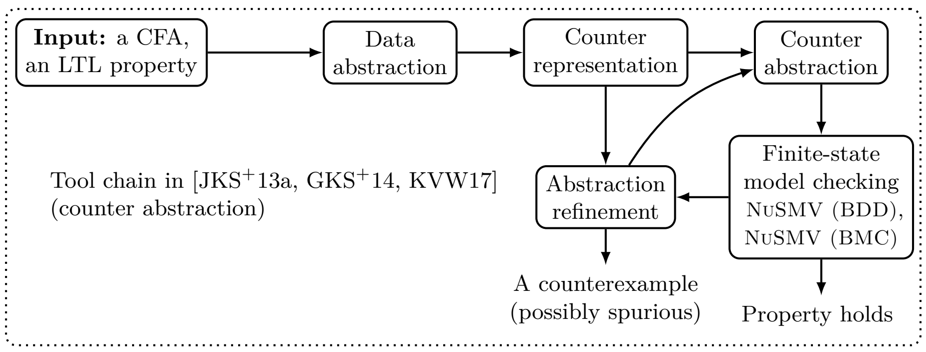
Let $I(0)$ be the formula over the variables of layer i that captures the initial states of the threshold automaton, and $B(i)$ be a state property over the variables of layer i. Then, parameterized reachability for the schema S is encoded with the following formula in linear integer arithmetic:

$$I(0) \wedge C_{\Omega_1}(0) \wedge T(0,1) \wedge \dots \wedge T(m-1,m) \wedge C_{\Omega_2}(m) \wedge \big(B(0) \vee \dots \vee B(m)\big).$$

3.7.3 Experiments

The technique presented in this chapter has been implemented by our team in the tool ByMC (Byzantine Model Checker [KW18]). In this section we recall the experimental evaluation from [KLVW17a]. All of our benchmark algorithms were originally published in pseudo-code, and we model them in a parametric extension of Promela, which was discussed in [JKS+13b, GKS+14].

Comparison to data and counter abstractions. In [KVW17], the following automated method has been introduced, which combines reduction methods with data abstraction [JKS+13a]:


Input: a CFA, an LTL property
Data abstraction
Counter representation
Counter abstraction
Finite-state model checking NuSMV (BDD), NuSMV (BMC)
Abstraction refinement
Tool chain in [JKS+13a, GKS+14, KVW17] (counter abstraction)
A counterexample (possibly spurious)
Property holds

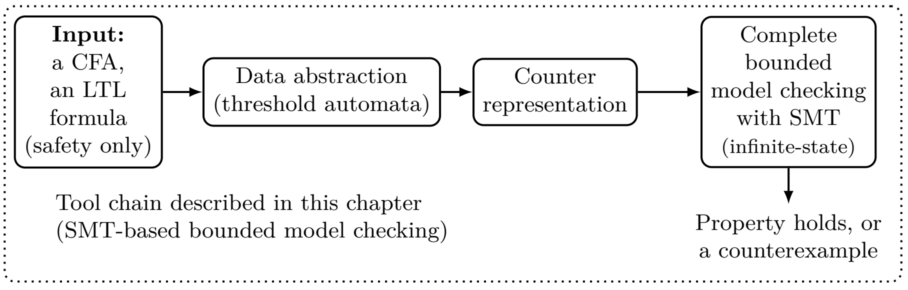

Input: a CFA, an LTL formula (safety only)
Data abstraction (threshold automata)
Counter representation
Complete bounded model checking with SMT (infinite-state)
Tool chain described in this chapter (SMT-based bounded model checking)
Property holds, or a counterexample

Figure 3.7: Tool chain with counter abstraction [JKS$^+$13a, GKS$^+$14, KVW17] on top, and with SMT-based bounded model checking on bottom.

1. Apply a parametric data abstraction to the process code to get a finite state process description, and construct the threshold automaton (TA) [JKS$^+$13a, KP00].
2. Compute the diameter bound, based on the control flow of the TA.
3. Construct a system with abstract counters, i.e., a counter abstraction [PXZ02, JKS$^+$13a].
4. Perform SAT-based bounded model checking [BCCZ99, CKOS04] up to the diameter bound, to check whether bad states are reached in the counter abstraction.
5. If a counterexample is found, check its feasibility and refine, if needed [CGJ$^+$03, JKS$^+$13a].

The top of Figure 3.7 gives a diagram that shows the technique [KVW17] based on counter abstraction.

The message counters are first mapped to parametric intervals, e.g., counters range over the abstract domain $\hat{D} = \{[0,1),\ [1,t+1), [t+1,n-t), [n-t,\infty)\}$. By doing so, we obtain a finite (data) abstraction of each process, and thus it is possible to represent the system as a counter system: We maintain one counter $\kappa[\ell]$ per local state ℓ of a process, as well as the counters for the sent messages. Then, in the counter abstraction step, every process counter $\kappa[\ell]$ is mapped to the set of parametric intervals $\hat{D}$. As the abstractions may produce spurious counterexamples, we run them in an abstraction-refinement loop that incrementally prunes spurious transitions and unfair executions. More details on the data and counter abstractions and refinement can be found in [JKS$^+$13a]. In our experiments based on the tool chain from the top of Figure 3.7, we use two kinds of model checkers as backend:

1. **BDD.** The counter abstraction is checked with nuXmv [CCD$^+$14] using Binary Decision Diagrams (BDDs). For safety properties, the tool executes the command `check_invar`. In the experiments, we used the timeout of three days, as there was at least one benchmark that needed a bit more than a day to complete.
2. **BMC.** The counter abstraction is checked with nuXmv using bounded model checking [BCCZ99]. To ensure completeness (at the level of counter abstraction), we explore the computations of the length up to the diameter bounds that were obtained in [KVW17]. To efficiently eliminate shallow spurious counterexamples, we first run the bounded model checker in the incremental mode up to length of 30. This is done by issuing the `nuXmv` command `check_ltlspec_sbmc_inc`, which uses the built-in SAT solver `MiniSAT`. Then, we run a single-shot SAT problem by issuing the `nuXmv` command `gen_ltlspec_sbmc` and checking the generated formula with the SAT solver `lingeling` [Bie13]. In our experiments, we set the timeout to one day.

Reachability for threshold automata. Based on the results of this chapter, to obtain a threshold automaton, our tool first applies data abstraction over the domain $\hat{D}$ to the Promela code, which abstracts the message counters that keep the number of messages received by every process, while the message counters for the sent messages are kept as integers. More details can be found in [KVW16]. Having constructed a threshold automaton, we compare two verification approaches:

1. **PARA^2.** *Bounded model checking with SMT.* The approach of this chapter. BYMC enumerates the schemas (as explained in Section 3.3), encodes them in SMT (as explained in Section 3.7.2) and checks every schema with the SMT solver `Z3` [MB08].
2. **FAST.** *Acceleration of counter automata.* In this chain, our tool constructs a threshold automaton and checks the reachability properties with the existing tool `FAST` [BFLP08]. For comparison with our tool, we run `FAST` with the `MONA` plugin that produced the best results in our experiments.

The challenge in the verification of FTDAs is the immense non-determinism caused by interleavings, asynchronous message passing, and faults. In our modeling, all these are reflected in non-deterministic choices in the Promela code. To obtain threshold automata, as required for our technique, our tool constructs a parametric interval data abstraction [JKS$^+$13a] that adds to non-determinism.

Table 3.1: The benchmarks used in our experiments. Some benchmarks, e.g., ABA, require us to consider several cases on the parameters, which are mentioned in the column "Case". The meaning of the other columns is as follows: $|\mathcal{L}|$ is the number of local states in TA, $|\mathcal{R}|$ is the number of rules in TA, $|\Phi^{\text{rise}}|$ and $|\Phi^{\text{fall}}|$ is the number of (R)- and (F)-guards respectively. Finally, $|\mathcal{S}|$ is the number of enumerated schemas, and Bound is the theoretical upper bound on $|\mathcal{S}|$, as given in Theorem 3.5.

#	Input	Case	Threshold	Automaton			Schemas	
	FTDA	(if more than one)	$\|\mathcal{L}\|$	$\|\mathcal{R}\|$	$\|\Phi^{\text{rise}}\|$	$\|\Phi^{\text{fall}}\|$	$\|\mathcal{S}\|$	Theor. Bound
1	FRB	—	7	10	1	0	1	1
2	STRB	—	7	15	3	0	4	6
3	NBACC	—	78	1356	0	0	1	1
4	NBAC	—	77	988	6	0	448	720
5	NBACG	—	24	44	4	0	14	24
6	CF1S	$f = 0$	41	266	4	0	14	24
7	CF1S	$f = 1$	41	266	4	1	60	120
8	CF1S	$f > 1$	68	672	6	1	3429	5040
9	C1CS	$f = 0$	101	1254	8	0	70	$4 \cdot 10^4$
10	C1CS	$f = 1$	70	629	6	1	140	5040
11	C1CS	$f > 1$	101	1298	8	1	630	$3.6 \cdot 10^5$
12	BOSCO	$\lfloor\frac{n+3t}{2}\rfloor + 1 = n - t$	28	126	6	0	20	720
13	BOSCO	$\lfloor\frac{n+3t}{2}\rfloor + 1 > n - t$	40	204	8	0	70	$4 \cdot 10^4$
14	BOSCO	$\lfloor\frac{n+3t}{2}\rfloor + 1 < n - t$	32	158	6	0	20	720
15	BOSCO	$n > 5t \wedge f = 0$	82	1292	12	0	924	$4.8 \cdot 10^8$
16	BOSCO	$n > 7t$	90	1656	12	0	924	$4.8 \cdot 10^8$
17	ABA	$\frac{n+t}{2} = 2t + 1$	37	180	6	0	448	720
18	ABA	$\frac{n+t}{2} > 2t + 1$	61	392	8	0	2100	$4 \cdot 10^4$
19	CBC	$\lfloor\frac{n}{2}\rfloor < n - t \wedge f = 0$	164	1996	22	12	2	$2.9 \cdot 10^{38}$
20	CBC	$\lfloor\frac{n}{2}\rfloor = n - t \wedge f = 0$	73	442	17	12	2	$8.8 \cdot 10^{30}$
21	CBC	$\lfloor\frac{n}{2}\rfloor < n - t \wedge f > 0$	304	6799	27	12	5	$2 \cdot 10^{46}$
22	CBC	$\lfloor\frac{n}{2}\rfloor = n - t \wedge f > 0$	161	2040	22	12	5	$2.9 \cdot 10^{38}$

Comparing to [KVW15], the presented implementation uses an optimization to schema checking that dramatically reduced the running times for some of the benchmarks. In this optimization, we group schemas in a prefix tree, whose nodes are contexts and edges are simple schemas. In each node of the prefix tree, our tool checks, whether there are configurations that are reachable from the initial configurations by following the schemas in the prefix. If there are no such reachable configurations, we can safely prune the whole suffix and thus prove many schemas to be unsatisfiable at once.

Evaluation. Table 3.1 summarizes the features of threshold automata that are automatically constructed by ByMC from parametric PROMELA. The number of local states $|\mathcal{L}|$ varies from 7 (FRB and STRB) to hundreds (C1CS and CBC). Our threshold automata

Table 3.2: Summary of our experiments on AMD Opteron®6272, 32 cores, 192 GB. The symbols are: "🕒" for timeout (72 h. for BDD and 24 h. otherwise); "💣" for memory overrun of 32 GB; "⚠" for BDD nodes overrun; "↻" for timeout in the refinement loop (72 h. for BDD and 24 h. otherwise); "☹" for spurious counterexamples due to counter abstraction.

	Input	**Time, seconds**				**Memory, GB**			
#	**FTDA**	**PARA^2**	**FAST**	**BMC**	**BDD**	**PARA^2**	**FAST**	**BMC**	**BDD**
1	**FRB**	1	1	1	1	0.1	0.1	0.1	0.1
2	**STRB**	1	1	3	2	0.1	0.1	0.1	0.1
3	**NBACC**	13	💣	🕒	🕒	0.1	💣	🕒	🕒
4	**NBAC**	88	💣	↻	🕒	0.1	💣	↻	🕒
5	**NBACG**	1	⚠	☹	↻	0.1	⚠	☹	↻
6	**CF1S**	6	2227	723	122	0.1	10.7	1.5	0.2
7	**CF1S**	11	6510	2235	2643	0.1	22.1	2.0	0.4
8	**CF1S**	263	⚠	🕒	40451	0.3	⚠	🕒	1.9
9	**C1CS**	45	💣	💣	10071	0.1	💣	💣	2.5
10	**C1CS**	21	💣	94962	87141	0.1	💣		9.3
11	**C1CS**	171	💣	💣	🕒	0.3	💣	💣	🕒
12	**BOSCO**	3	⚠	17892	294	0.1	⚠	1.4	0.2
13	**BOSCO**	17	⚠	↻	☹	0.1	⚠	↻	☹
14	**BOSCO**	5	⚠	2424	4	0.1	⚠	1.9	0.1
15	**BOSCO**	1013	💣	🕒	405	0.2	💣	🕒	0.7
16	**BOSCO**	1459	⚠	🕒	847	0.4	⚠	🕒	1.3
17	**ABA**	16	767	☹	11	0.1	3.5	☹	0.1
18	**ABA**	294	5757	☹	41	0.3	12.4	☹	0.2
19	**CBC**	128	💣	💣	💣	0.6	💣	💣	💣
20	**CBC**	9	⚠	2671	41873	0.1	⚠	2.8	9.9
21	**CBC**	3351	3304	💣	💣	19.3	0.1	💣	💣
22	**CBC**	215	⚠	💣	💣	4.0	⚠	💣	💣

are obtained by applying interval abstraction to Promela code, which keeps track of the number of messages received by each process. Thus, the number $|\mathcal{L}|$ is proportional to the number of control states and $|\widehat{D}|^k$, where $\widehat{D}$ is the domain of parametric intervals (discussed above) and k is the number of message types. Sometimes, one can manually construct a more efficient threshold automaton that models the same fault-tolerant distributed algorithm and preserves the same safety properties. For instance, Figure 3.1 shows a manual abstraction of ABA that has only 5 local states, in contrast to 61 local states in the automatic abstraction (cf. Table 3.1). We leave open the question of whether one can automatically construct a minimal threshold automaton with respect to given specifications. Experiments on manual encodings are given in [KW18].

Table 3.2 summarizes the experiments conducted with the techniques introduced earlier in this section: BDD, BMC, PARA^2, and FAST. On large problems, our new technique

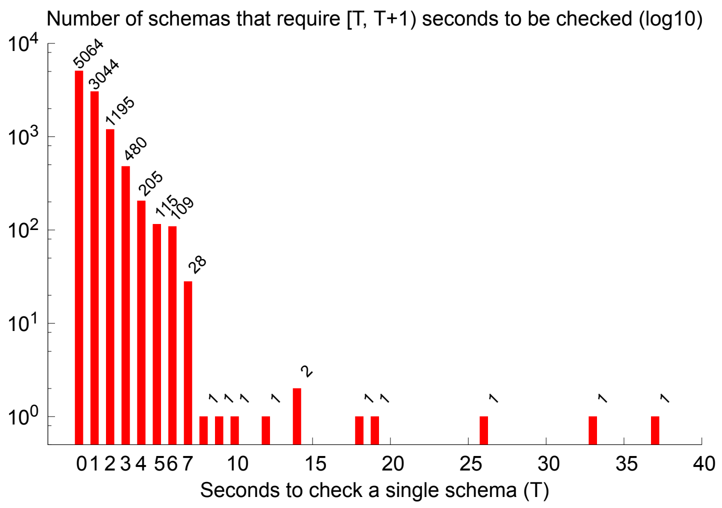


Figure 3.8: The times required to check individual schemas and the distribution of schemas over these times (the value 0 refers to the running times of less than a second). The benchmarks containing the schemas that are verified in (a) $T \geq 8$ sec. and (b) $T \geq 18$ sec. are: (a) C1CS, CBC, CF1S, and (b) CBC and CF1S.

works significantly better than BDD- and SAT-based model checking. BDD-based model checking works very well on top of counter abstraction. Importantly, our new technique does not use abstraction refinement. In comparison to the experiments from [KVW15], we verified safety of a larger set of benchmarks with `nuXmv`. We believe that this is due to the improvments in `nuXmv` and, probably, slight modifications of the benchmarks from [KLVW17b].

NBAC and NBACC are challenging as the model checker produces many spurious counterexamples, which are an artifact of counter abstraction losing or adding processes. When using SAT-based model checking, the individual calls to `nuXmv` are fast, but the abstraction-refinement loop times out, due to a large number of refinements (about 500). BDD-based model checking times out when looking for a counterexample. Our new technique, preserves the number of proceses, and thus, there are no spurious counterexamples of this kind. In comparison to the general-purpose acceleration tool FAST, our tool uses less memory and is faster on the benchmarks where FAST is successful.

Sets of schemas and time to check a single schema. On one hand, Theorem 3.5 gives us a theoretical bound on the number of schemas to be explored. On the other hand, optimizations discussed in Section 3.7.1 introduce many ways of reducing the number of schemas. Two columns in Table 3.1 compare the theoretical bound and the practical number of schemas: the column “Theoretical bound” shows the bound of $(|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|)!$, while the column $|\mathcal{S}|$ shows the actual number of schemas. (For reachability, we are merging the schemas with the prefix tree, and thus the actual number of explored schemas is even smaller.) As one can see, the theoretical bound is quite pessimistic, and is only useful to show completeness of the set of schemas. The much smaller numbers for the fault-tolerant distributed algorithms are due to a natural order on guards, e.g., as $x \geq t+1$ becomes true earlier than $x \geq n-t$ under the resilience condition $n > 3t$. The drastic reduction in the case of CBC is due to the control flow optimization discussed in Section 3.7.1 and the fact that basically all guards are forward-unlocking.

3.8 Detailed Proofs for Section 3.4

In this section we recall claims from Section 3.4, and present their formal proofs.

Lemma 3.6. *Given a threshold automaton, a configuration σ, and a schedule $\tau = (r, f_1)$, $\dots$, (r, f_m) applicable to σ, one of the two schedules is also applicable to σ and results in $\tau(\sigma)$: either schedule $(r, f_1 + \ldots + f_m)$, or schedule $(r, 0)$.*

Proof. We distinguish two cases:

Case $\boldsymbol{r.to = r.from}$. Then, $r.\mathbf{u} = \mathbf{0}$, and $\tau^k(\sigma) = \sigma$ for $0 \leq k \leq |\tau|$. Consequently, the schedule $(r, 0)$ is applicable to σ, and it results in $\tau(\sigma) = \sigma$.

Case $\boldsymbol{r.to \neq r.from}$. We prove by induction on the length $k : 1 \leq k \leq m$ of a prefix of τ, that the following constraints hold for all k:

$$(\tau^k(\sigma)).\boldsymbol{\kappa}[r.\mathit{from}] = \sigma.\boldsymbol{\kappa}[r.\mathit{from}] - (f_1 + \cdots + f_k) \tag{3.9}$$

$$(\tau^k(\sigma)).\mathbf{g} = \sigma.\mathbf{g} + (f_1 + \cdots + f_k) \cdot r.\mathbf{u} \tag{3.10}$$

$$(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + f \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\text{fall}} \wedge r.\varphi^{\text{rise}} \text{ for all } f \in \{0, \ldots, f_1 + \cdots + f_k\} \tag{3.11}$$

Base case: $k = 1$. As schedule τ is applicable to σ, its first transition is enabled in σ. Thus, by the definition of an enabled transition, the rule r is unlocked, i.e., for all $f \in \{0, \ldots, f_1\}$, it holds $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + f_1 \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\text{fall}} \wedge r.\varphi^{\text{rise}}$. By the definition, once the transition $\tau[1]$ is applied, it holds that $\tau^1(\sigma).\boldsymbol{\kappa}[\mathit{from}] = \sigma.\boldsymbol{\kappa}[\mathit{from}] - f_1$ and $(\tau^k(\sigma)).\mathbf{g} = \sigma.\mathbf{g} + f_1 \cdot r.\mathbf{u}$. Thus, constraints (3.9)–(3.11) are satisfied for $k = 1$.

Inductive step: $k > 1$. As schedule τ is applicable to σ, its prefix τ^k is applicable to σ. Hence, transition $\tau[k]$ is applicable to $\tau^{k-1}(\sigma)$.

By the definition of an enabled transition, for all $f \in \{0, \ldots, f_k\}$, it holds

$$((\tau^{k-1}(\sigma)).\boldsymbol{\kappa}, ((\tau^{k-1}(\sigma)).\mathbf{g} + f \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\text{fall}} \wedge r.\varphi^{\text{rise}}.$$

By applying the Equation (3.10) for $k-1$ of the inductive hypothesis, we obtain that for all $f \in \{0, \ldots, f_k\}$, it holds that $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + (f_1 + \cdots + f_{k-1} + f \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\text{fall}} \wedge r.\varphi^{\text{rise}}$. By combining this constraint with the constraint (3.11) for $k-1$, we arrive at the constraint (3.11) for k.

By applying $\tau[k]$, we get that $(\tau^k(\sigma)).\boldsymbol{\kappa}[r.\mathit{from}] = (\tau^{k-1}(\sigma)).\boldsymbol{\kappa}[r.\mathit{from}] - f_k$ and $(\tau^k(\sigma)).\mathbf{g} = (\tau^{k-1}(\sigma)).\mathbf{g} + f_k \cdot r.\mathbf{u}$. By applying (3.9) and (3.10) for $k-1$ to these equations, we arrive at the Equations (3.9) and (3.10) for k.

Based on (3.9) and (3.11) for all values of k, and in particular $k = m$, we can now show applicability. From Equation (3.9), we immediately obtain that $\sigma.\boldsymbol{\kappa}[r.\mathit{from}] \geq f_1 + \cdots + f_m$. From constraint (3.11), we obtain that $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + f \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\mathrm{fall}} \wedge r.\varphi^{\mathrm{rise}}$ for all $f \in \{0, \dots, f_1 + \cdots + f_m\}$. These are the required conditions for the transition $(r, f_1 + \cdots + f_m)$ to be applicable to the configuration σ. □

In the following, we fix a threshold automaton TA, a context Ω, and a non-singleton looplet c of the slice $\mathsf{TA}|_{\Omega}$. We also fix a configuration σ of TA and a schedule τ that is contained in c and it is applicable to σ. Our goal is to prove Lemma 3.8, which states that $\mathsf{crep}_c[\sigma, \tau]$ is indeed applicable to σ and ends in $\tau(\sigma)$. To this end, we first prove auxiliary Lemmas 3.16–3.20.

Lemma 3.16. *For every $i : 1 \leq i \leq |E_{\mathsf{in}}|$, it holds that $\sigma.\boldsymbol{\kappa}[r_i.\mathit{from}] \geq \max(\delta_{\mathsf{in}}(i), 0)$, where $r_i = L_c(e_{\mathsf{in}}(i))$.*

Proof. Recall that by the definition of a configuration, every counter $\sigma.\boldsymbol{\kappa}[\ell]$ is non-negative. If $\delta_{\mathsf{in}}(i) \geq 0$, then $\max(\delta_{\mathsf{in}}(i), 0) = \delta_{\mathsf{in}}(i) = \sigma.\boldsymbol{\kappa}[r_i.\mathit{from}] - \sigma'.\boldsymbol{\kappa}[r_i.\mathit{from}]$, which is bound from above by $\sigma.\boldsymbol{\kappa}[r_i.\mathit{from}]$. Otherwise, $\delta_{\mathsf{in}}(i) \leq 0$, and we trivially have $\max(\delta_{\mathsf{in}}(i), 0) = 0$ and $0 \leq \sigma.\boldsymbol{\kappa}[r_i.\mathit{from}]$. □

Lemma 3.17. *Schedule $\tau_{\mathsf{in}} = (r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \dots, (r_{\mathsf{in}}(|E_{\mathsf{in}}|), w_{\mathsf{in}}(|E_{\mathsf{in}}|))$ is applicable to σ.*

Proof. We denote by α^i the schedule $(r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \dots, (r_{\mathsf{in}}(i), w_{\mathsf{in}}(i))$, for $1 \leq i \leq |E_{\mathsf{in}}|$. Then $\tau_{\mathsf{in}} = \alpha^{|E_{\mathsf{in}}|}$.

All rules $r_{\mathsf{in}}(1), \dots, r_{\mathsf{in}}(|E_{\mathsf{in}}|)$ are from $\mathcal{R}|_{\Omega}$, and thus are unlocked. Hence, it is sufficient to show that the values of the locations from the set V_c are large enough to enable each transition $(r_{\mathsf{in}}(i), w_{\mathsf{in}}(i))$ for $1 \leq i \leq |E_{\mathsf{in}}|$. To this end, we prove by induction that $(\alpha^{i-1}(\sigma)).\boldsymbol{\kappa}[r_i.\mathit{from}] \geq w_{\mathsf{in}}(i)$, for $1 \leq i \leq |E_{\mathsf{in}}|$ and $r_i = L_c(e_{\mathsf{in}}(i))$.

Base case: $i = 1$. For $r_1 = L_c(e_{\mathsf{in}}(1))$, we want to show that $\sigma.\boldsymbol{\kappa}[r_1.\mathit{from}] \geq w_{\mathsf{in}}(1)$. As $e_{\mathsf{in}}(1)$ is the first element of the sequence $e_{\mathsf{in}}(1), \dots, e_{\mathsf{in}}(E_{\mathsf{in}})$, which respects the order $\preceq_{\mathsf{in}}$, we conclude that $w_{\mathsf{in}}(1) = \max(\delta_{\mathsf{in}}(1), 0)$. From Lemma 3.16, it follows that $\sigma.\boldsymbol{\kappa}[r_1.\mathit{from}] \geq \max(\delta_{\mathsf{in}}(1), 0)$.

Inductive step k: assume that for all $i : 1 \leq i \leq k-1 < |E_{\mathsf{in}}|$, schedule α^i is applicable to σ and show that $(\alpha^{k-1}(\sigma)).\boldsymbol{\kappa}[r_k.\mathit{from}] \geq w_{\mathsf{in}}(k)$ with $r_k = L_c(e_{\mathsf{in}}(k))$.

To this end, we construct the set of edges P_k that precede the edge $e_{\mathsf{in}}(k)$ in the topological order $\preceq_{\mathsf{in}}$, that is, $P_k = \{e \mid e \in E_{\mathsf{in}},\ e \preceq_{\mathsf{in}} e_{\mathsf{in}}(k),\ e \neq e_{\mathsf{in}}(k)\}$. We show that the following equation holds:

$$\alpha^{k-1}(\sigma)).\boldsymbol{\kappa}[r_k.\mathit{from}] = \sigma.\boldsymbol{\kappa}[r_k.\mathit{from}] + \sum_{e_{\mathsf{in}}(j) \in P_k} \max(\delta_{\mathsf{in}}(j), 0). \tag{3.12}$$

Indeed, pick an edge $e_{\mathsf{in}}(j) \in P_k$. Edge $e_{\mathsf{in}}(j)$ adds $w_{\mathsf{in}}(j)$ to the counter $\boldsymbol{\kappa}[r_k.\mathit{from}]$. As the sequence $\{e_{\mathsf{in}}(i)\}_{i \le k}$ is topologically sorted, it follows that $j < k$. Moreover, as the tree T_{in} is oriented towards the root, $e_{\mathsf{in}}(k)$ is the only edge leaving the local state $r_k.\mathit{from}$. Thus, no edge $e_{\mathsf{in}}(i)$ with $i < k$ decrements the counter $\sigma.\boldsymbol{\kappa}[r_k.\mathit{from}]$.

From Equation (3.12) and Lemma 3.16, we conclude that $(\alpha^{k-1}(\sigma)).\boldsymbol{\kappa}[r_k.\mathit{from}]$ is not less than $\max(\delta_{\mathsf{in}}(k), 0) + \sum_{e_{\mathsf{in}}(j)\colon e_{\mathsf{in}}(j) \preceq_{\mathsf{in}} e_{\mathsf{in}}(k),\ j \ne k} \max(\delta_{\mathsf{in}}(j), 0)$, which equals to $w_{\mathsf{in}}(k)$. This proves the inductive step.

Therefore, we have shown that $\tau_{\mathsf{in}} = \alpha^{|E_{\mathsf{in}}|}$ is applicable to σ. □

The following lemma is easy to prove by induction on the length of a schedule. The base case for a single transition follows from the definition of a counter system.

Lemma 3.18. *Let σ and σ' be two configurations and τ be a schedule applicable to σ such that $\tau(\sigma) = \sigma'$. Then it holds that $\sum_{\ell \in \mathcal{L}} (\sigma'[\ell] - \sigma[\ell]) = 0$.*

Further, we show that the required number of processes is reaching (or leaving) the hub, when the transitions derived from the trees T_{in} and T_{out} are executed:

Lemma 3.19. *The following equality holds:*

$$\sigma'.\boldsymbol{\kappa}[h] - \sigma.\boldsymbol{\kappa}[h] = \sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0) - \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0).$$

Proof. Recall that T_{in} is a tree directed towards h, and the undirected version of T_{in} is a spanning tree of graph C. Hence, for each local state $\ell \in V_c \setminus \{h\}$, there is exactly one edge $e \in E_{\mathsf{in}}$ with $L_c(e).\mathit{from} = \ell$. Thus, the following equation holds:

$$\sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0) = \sum_{\ell \in V_c \setminus \{h\}} \max(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell], 0). \tag{3.13}$$

Similarly, T_{out} is a tree directed outwards h, and the undirected version of T_{out} is a spanning tree of graph C. Hence, for each local state $\ell \in V_c \setminus \{h\}$, there is exactly one edge $e \in E_{\mathsf{out}}$ with $L_c(e).\mathit{to} = \ell$. Thus, the following equation holds:

$$\sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0) = \sum_{\ell \in V_c \setminus \{h\}} \max(\sigma'.\boldsymbol{\kappa}[\ell] - \sigma.\boldsymbol{\kappa}[\ell], 0). \tag{3.14}$$

By combining (3.13) and (3.14), we obtain the following:

$$\begin{aligned}
&\sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0) - \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0) \\
= &\sum_{\ell \in V_c \setminus \{h\}} \big(\max(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell], 0) - \max(\sigma'.\boldsymbol{\kappa}[\ell] - \sigma.\boldsymbol{\kappa}[\ell], 0)\big) \\
= &\sum_{\ell \in V_c \setminus \{h\}} (\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell]) = \big(\sum_{\ell \in V_c} \sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell]\big) - (\sigma.\boldsymbol{\kappa}[h] - \sigma'.\boldsymbol{\kappa}[h]).
\end{aligned}$$

As the initial schedule τ is applicable to σ, and $\tau(\sigma) = \sigma'$, by Lemma 3.18, $\sum_{\ell \in \mathcal{L}}(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell]) = 0$. As all rules in $\mathsf{crep}_c[\sigma,\tau]$ are from $\mathcal{R}|_\Omega$ and thus change only the counters of local states in V_c, for each local state $\ell \in \mathcal{L} \setminus V_c$, its respective counter does not change, that is, $\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell] = 0$. Hence, $\sum_{\ell \in V_c}(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell]) = 0$. From this and the above equation, we obtain the required. □

Lemma 3.20. *With τ_{in} denote the schedule $(r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \dots, (r_{\mathsf{in}}(|E_{\mathsf{in}}|), w_{\mathsf{in}}(|E_{\mathsf{in}}|))$. The following equation holds:*

$$\tau_{\mathsf{in}}(\sigma).\boldsymbol{\kappa}[\ell] = \begin{cases} \sigma'.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0), & \text{if } \ell = h \\ \min(\sigma.\boldsymbol{\kappa}[\ell], \sigma'.\boldsymbol{\kappa}[\ell]), & \text{if } \ell \in V_c \setminus \{h\}. \end{cases}$$

Proof. We prove the lemma by case distinction:

Case $\ell = h$. We show that $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[h] = \sigma.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0)$. Indeed, let P be the indices of edges coming into h, i.e., $P = \{i \mid 1 \le i \le |E_{\mathsf{in}}|,\ L_c(e_{\mathsf{in}}(i)) = r,\ h = r.to\}$. As all edges in T_{in} are oriented towards h, it holds that $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[h]$ equals to $\sigma.\boldsymbol{\kappa}[h] + \sum_{i \in P} w_{\mathsf{in}}(i)$. By unfolding the definition of w_{in}, we obtain that $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[h] = \sigma.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0)$. It is easy to see that by Lemma 3.19, this sum equals to $\sigma'.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0)$. This proves the first case.

Case $\ell \in V_c \setminus \{h\}$. We show that $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[\ell] = \min(\sigma.\boldsymbol{\kappa}[\ell], \sigma'.\boldsymbol{\kappa}[\ell])$. Indeed, fix a node $\ell \in V_c \setminus \{h\}$ and construct two sets: the set of incoming edges $\mathit{In} = \{e_{\mathsf{in}}(i) \mid \exists \ell' \in V_c.\ e_{\mathsf{in}}(i) = (\ell', \ell)\}$ and the singleton set of outgoing edges $\mathit{Out} = \{e_{\mathsf{in}}(i) \mid \exists \ell' \in V_c.\ e_{\mathsf{in}}(i) = (\ell, \ell')\}$. By summing up the effect of all transitions in τ_{in}, we obtain $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell] + \sum_{e_{\mathsf{in}}(i) \in \mathit{In}} w_{\mathsf{in}}(i) - \sum_{e_{\mathsf{out}}(i) \in \mathit{Out}} w_{\mathsf{out}}(i)$. By unfolding the definition of w_{in}, we obtain $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell] - \sum_{e_{\mathsf{in}}(i) \in \mathit{Out}} \delta_{\mathsf{in}}(i)$, which can be rewritten as $\sigma.\boldsymbol{\kappa}[\ell] - \max(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell], 0)$, which, in turn, equals to $\min(\sigma.\boldsymbol{\kappa}[\ell], \sigma'.\boldsymbol{\kappa}[\ell])$. This proves the second case. □

Now we are in a position to prove that schedule $\mathsf{crep}_c[\sigma,\tau]$ is applicable to configuration σ and results in configuration $\tau(\sigma)$:

Lemma 3.8. *Fix a threshold automaton TA, a context Ω, and a non-singleton looplet c of the slice $\mathsf{TA}|_\Omega$. Let σ be a configuration of TA and let τ be a schedule that is contained in c and it is applicable to σ. Then the schedule $\mathsf{crep}_c[\sigma,\tau]$ has the following properties:*

a) $\mathsf{crep}_c[\sigma,\tau]$ *is applicable to σ, and*

b) $\mathsf{crep}_c[\sigma,\tau]$ *results in $\tau(\sigma)$ when applied to σ.*

Proof. Denote with τ_{in} the prefix $(r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \cdots, (r_{\mathsf{in}}(|E_{\mathsf{in}}|), w_{\mathsf{in}}(|E_{\mathsf{in}}|))$ of the schedule $\mathsf{crep}_c[\sigma,\tau]$. For each $j : 1 \le j \le |E_{\mathsf{out}}|$, denote with β^j the prefix of $\mathsf{crep}_c[\sigma,\tau]$ that has length of $|E_{\mathsf{in}}| + j$. Note that $\beta^{|E_{\mathsf{out}}|} = \mathsf{crep}_c[\sigma,\tau]$.

Proving applicability of $\mathsf{crep}_c[\sigma,\tau]$ *to* σ.. We notice that all rules in $\mathsf{crep}_c[\sigma,\tau]$ are from $\mathcal{R}|_\Omega$ and thus are unlocked, and that τ_{in} is applicable to σ by Lemma 3.17. Hence, we only have to check that the values of counters from V_c are large enough, so that transitions $(r_{\mathsf{out}}(j), w_{\mathsf{out}}(j))$ can fire.

We prove that each schedule β^j is applicable to σ, for $j : 1 \le j \le |E_{\mathsf{out}}|$. We do so by induction on the distance from the root h in the tree T_{out}.

Base case: root node h. Denote with O_h the set $\{(\ell,\ell') \in E_{\mathsf{out}} \mid \ell = h\}$. Let $j_1,\dots,j_m$ be the indices of all edges in O_h, and j_m be the maximum among them.

From Lemma 3.20, $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[h] = \sigma'.\boldsymbol{\kappa}[h] + \sum_{1\le i\le|E_{\mathsf{out}}|}\max(\delta_{\mathsf{out}}(i),0) = \sigma'.\boldsymbol{\kappa}[h] + \sum_{e_{\mathsf{out}}(j)\in O_h} w_{\mathsf{out}}(j)$. Thus, every transition $(e_{\mathsf{out}}(j), w_{\mathsf{out}}(j))$ with $e_{\mathsf{out}}(j) \in O_h$, is applicable to $\beta^{j-1}(\sigma)$. Also, $(\beta^{j_m}(\sigma)).\boldsymbol{\kappa}[h] = \sigma'.\boldsymbol{\kappa}[h]$.

Inductive step: assume that for a node $\ell \in V_c$ and an edge $e_{\mathsf{out}}(k) = (\ell,\ell') \in E_{\mathsf{out}}$ outgoing from node ℓ, schedule β^k is applicable to configuration σ. Show that for each edge $e_{\mathsf{out}}(i)$ outgoing from node ℓ' the following hold: (i) schedule β^i is also applicable to σ; and (ii) $\beta^{|E_{\mathsf{out}}|}(\sigma).\boldsymbol{\kappa}[\ell'] = \sigma'.\boldsymbol{\kappa}[\ell']$.

(i) As the sequence $\{e_{\mathsf{out}}(j)\}_{j\le|E_{\mathsf{out}}|}$ is topologically sorted, for each edge $e_{\mathsf{out}}(i)$ outgoing from node ℓ', it holds that $k < i$.

From Lemma 3.20, we have that $\beta^k(\sigma).\boldsymbol{\kappa}[\ell'] = \min(\sigma.\boldsymbol{\kappa}[\ell'], \sigma'.\boldsymbol{\kappa}[\ell'])$. Because the transition $(e_{\mathsf{out}}(k), w_{\mathsf{out}}(k))$ adds $w_{\mathsf{out}}(k)$ to $\beta^{k-1}(\sigma).\boldsymbol{\kappa}[\ell']$, we have $\beta^k(\sigma).\boldsymbol{\kappa}[\ell'] = \min(\sigma.\boldsymbol{\kappa}[\ell'], \sigma'.\boldsymbol{\kappa}[\ell'])+ w_{\mathsf{out}}(k)$. Let S be the set of all immediate successors of $e_{\mathsf{out}}(k)$, i.e., $S = \{i \mid \exists \ell''.\ (\ell',\ell'') = e_{\mathsf{out}}(i)\}$. From the definition of $w_{\mathsf{out}}(k)$, it follows that $w_{\mathsf{out}}(k) = \max(\delta_{\mathsf{out}}(k),0) + \sum_{s\in S} w_{\mathsf{out}}(s)$. Thus, the transition $(e_{\mathsf{out}}(i), w_{\mathsf{out}}(i))$ for edge $e_{\mathsf{out}}(i)$ outgoing from node ℓ', can be executed.

(ii) Let $j_1,\dots,j_m$ be the indices of all edges outgoing from ℓ', and j_m be the maximum among them. From (i), it follows that $(\beta^{j_m}(\sigma)).\boldsymbol{\kappa}[\ell'] = \min(\sigma.\boldsymbol{\kappa}[\ell'], \sigma'.\boldsymbol{\kappa}[\ell']) + \max(\delta_{\mathsf{out}}(k),0)$, which equals to $\sigma'.\boldsymbol{\kappa}[\ell']$.

This proves that the schedule $\beta^{|E_{\mathsf{out}}|} = \mathsf{crep}_c[\sigma,\tau]$ is applicable to σ.

Proving that $\mathsf{crep}_c[\sigma,\tau]$ *results in* $\tau(\sigma)$. From the induction above, we conclude that for each $\ell \in V_c$, it holds that $(\beta^{|E_{\mathsf{out}}|}(\sigma)).\boldsymbol{\kappa}[\ell] = \sigma'.\boldsymbol{\kappa}[\ell]$. Edges in the trees T_{in} and T_{out} change only local states from V_c. We conclude that for all $\ell \in \mathcal{L}$, it holds that $\mathsf{crep}_c[\sigma,\tau](\sigma).\boldsymbol{\kappa}[\ell] = \sigma'.\boldsymbol{\kappa}[\ell]$. As the rules in non-singleton looplets do not change shared variables, we have $\mathsf{crep}_c[\sigma,\tau](\sigma).\mathbf{g} = \sigma.\mathbf{g} = \sigma'.\mathbf{g}$. Therefore, it holds that $\mathsf{crep}_c[\sigma,\tau](\sigma) = \sigma'$. □

3.9 Detailed Proofs for Section 3.5

Lemma 3.12. *Given a threshold automaton, a context* Ω*, a configuration* σ*, a steady schedule* τ *applicable to* σ*, and a sequence* $c_1,\dots,c_m$ *of all looplets in the slice* $\mathcal{R}|_\Omega$ *with the property* $c_i \prec_C^{lin} c_j$ *for* $1 \le i < j \le m$*, the following holds:*

a) Schedule $\tau|_{c_1}$ *is applicable to the configuration* σ.

b) Schedule $\tau|_{c_2,\dots,c_m}$ *is applicable to the configuration* $\tau|_{c_1}(\sigma)$.

c) Schedule $\tau|_{c_1} \cdot \tau|_{c_2,\dots,c_m}$*, when applied to* σ*, results in configuration* $\tau(\sigma)$.

Proof. In the following, we show Points a)–c) one-by-one.

We need extra notation. For a local state ℓ we denote by $\mathbf{1}_\ell$ the $|L|$-dimensional vector, where the ℓth component is 1, and all the other components are 0. Given a schedule $\rho = t_1 \cdots t_k$, we introduce a vector $\Delta_{\boldsymbol{\kappa}}(\rho) \in \mathbb{Z}^{|L|}$ to keep counter difference and a vector $\Delta_{\mathbf{g}}(\rho) \in \mathbb{N}_0^{|\Gamma|}$ to keep difference on shared variables as follows:

$$\Delta_{\boldsymbol{\kappa}}(\rho) = \sum_{1 \le i \le |\rho|} t_i.\mathit{factor} \cdot (\mathbf{1}_{t_i.\mathit{to}} - \mathbf{1}_{t_i.\mathit{from}}) \quad \text{and} \quad \Delta_{\mathbf{g}}(\rho) = \sum_{1 \le i \le |\rho|} t_i.\mathbf{u}$$

Proof of a). Assume, on contrary, that schedule $\tau|_{c_1}$ is not applicable to configuration σ. Thus, there is a schedule τ' and a transition t^* that constitute a prefix of $\tau|_{c_1}$, with the following property: τ' is applicable to σ, whereas $\tau' \cdot t^*$ is not applicable to σ. Let $\ell = t^*.\mathit{from}$ and $\ell' = t^*.\mathit{to}$.

There are three cases of why t^* may be not applicable to $\tau'(\sigma)$:

(i) There is not enough processes to move: $(\sigma.\boldsymbol{\kappa} + \Delta_{\boldsymbol{\kappa}}(\tau' \cdot t^*))[\ell] < 0$. As τ is applicable to σ, there is a transition t of τ with $[t.\mathit{rule}] \ne c_1$ and $t.\mathit{to} = \ell$ as well as $t.\mathit{factor} > 0$. From this, by definition of $\prec_C^{\mathit{lin}}$, it follows that $[t.\mathit{rule}] \prec_C^{\mathit{lin}} c_1$. This contradicts the lemma's assumption on the order $c_1 \prec_C^{\mathit{lin}} \cdots \prec_C^{\mathit{lin}} c_m$.

(ii) The condition $t^*.\varphi^{\text{rise}}$ is not satisfied, that is, $\tau'(\sigma) \not\models t^*.\varphi^{\text{rise}}$. Then, there is a guard $\varphi \in$ ‘ (’$t^*.\varphi^{\text{rise}}$) with $\tau'(\sigma) \not\models \varphi$.

Since τ is applicable to σ, there is a prefix $\rho \cdot t$ of τ, for a schedule ρ and a transition t that unlocks φ in $\rho(\sigma)$, that is, $\rho(\sigma) \not\models \varphi$ and $t(\rho(\sigma)) \models \varphi$. Thus, transition t changes the context: $\omega(\rho(\sigma)) \ne \omega(t(\rho(\sigma)))$. This contradicts the assumption that schedule τ is steady.

(iii) The condition $t^*.\varphi^{\text{fall}}$ is not satisfied: $\tau'(\sigma) \not\models t^*.\varphi^{\text{fall}}$. Then, there is a guard $\varphi \in$ ‘ (’$t^*.\varphi^{\text{fall}}$) with $\tau'(\sigma) \not\models \varphi$.

Let ρ be the longest prefix of τ satisfying $\rho|_{c_1} = \tau'$. Note that $\rho \cdot t^*$ is also a prefix of τ. As $\rho|_{c_1} = \tau'$ and no transition decrements the shared variables, we conclude that $(\tau'(\sigma)).\mathbf{g} \le (\rho(\sigma)).\mathbf{g}$. From this and from the fact that $\tau'(\sigma) \not\models \varphi$, it follows that $\rho(\sigma) \not\models \varphi$. Thus transition t^* is not applicable to $\rho(\sigma)$. This contradicts the assumption that τ is applicable to σ.

From (i), (ii), and (iii), we conclude that *a)* holds.

Proof of b). We show that $\tau|_{c_2,\dots,c_m}$ is applicable to $\tau|_{c_1}(\sigma)$.

To this end, we fix an arbitrary prefix τ' of τ, a transition t, and a suffix τ'', that constitute τ, that is, $\tau = \tau' \cdot t \cdot \tau''$. We show that if schedule $\tau'|_{c_2,\dots,c_m}$ is applicable to $\tau|_{c_1}(\sigma)$, then so is $(\tau' \cdot t)|_{c_2,\dots,c_m}$.

Let us assume that $\tau'|_{c_2,\dots,c_m}$ is applicable to $\tau|_{c_1}(\sigma)$, and let σ'' denote the resulting state $(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m})(\sigma)$. We consider two cases:

- $[t.\mathit{rule}] = c_1$. This case holds trivially, as $(\tau' \cdot t)|_{c_2,\dots,c_m}$ equals to $\tau'|_{c_2,\dots,c_m}$, which is applicable to $\tau|_{c_1}(\sigma)$ by assumption.
- $[t.\mathit{rule}] \neq c_1$. In order to prove that$(\tau' \cdot t)|_{c_2,\dots,c_m}$ is applicable to $\tau|_{c_1}(\sigma)$, we show that counters $\sigma''.\boldsymbol{\kappa}$ and shared variables $\sigma''.\mathbf{g}$ are large enough, so that transition t is applicable to σ'':

(i) We start by showing that $\sigma''.\boldsymbol{\kappa}[t.\mathit{from}] \geq t.\mathit{factor}$. We distinguish between different cases on source and target states of transition t.

(i.A) If there is a rule $r \in c_1$ with $t.to = r.\mathit{from}$. On one hand, as $[t.\mathit{rule}] \neq c_1$, by definition of $\prec_C^{lin}$, it follows that $[t.\mathit{rule}] \prec_C^{lin} \dots \prec_C^{lin} c_1$. On the other hand, as $[t.\mathit{rule}] \neq c_1$ and $c_1, \dots, c_m$ are all classes of the rules used in τ, it holds that $[t.\mathit{rule}] \in \{c_2, \dots, c_m\}$. By the lemma's assumption, $c_1 \prec_C^{lin} \cdots \prec_C^{lin} c_m$, and thus, $c_1 \prec_C^{lin} \cdots \prec_C^{lin} [t.\mathit{rule}]$. We arrive at a contradiction.

(i.B) If there is a rule $r \in c_1$ with $r.to = t.\mathit{from}$. Assume, on contrary, that t is not applicable to σ'', that is, $\sigma''.\boldsymbol{\kappa}[t.\mathit{from}] < t.\mathit{factor}$. On one hand, transition t is not applicable to $\sigma'' = (\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m})(\sigma)$. Then by the definition of $\Delta_{\boldsymbol{\kappa}}$, it holds that $\sigma[t.\mathit{from}] + (\Delta_{\boldsymbol{\kappa}}(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m}) + \Delta_{\boldsymbol{\kappa}}(t))[t.\mathit{from}] < 0$. By observing that $\tau|_{c_1} = \tau'|_{c_1} + \tau''|_{c_1}$, we derive the following inequality:

$$\sigma[t.\mathit{from}] + (\Delta_{\boldsymbol{\kappa}}(\tau'|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau''|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau'|_{c_2,\dots,c_m}) + \Delta_{\boldsymbol{\kappa}}(t))[t.\mathit{from}] < 0 \quad (3.15)$$

On the other hand, schedule $\tau = \tau' \cdot t \cdot \tau''$ is applicable to configuration σ. Thus, $\sigma[t.\mathit{from}] + (\Delta_{\boldsymbol{\kappa}}(\tau') + \Delta_{\boldsymbol{\kappa}}(t) + \Delta_{\boldsymbol{\kappa}}(\tau''))[t.\mathit{from}] \geq 0$. By observing that $\tau|_{c_1} = \tau'|_{c_1} + \tau''|_{c_1}$ and $\tau|_{c_2,\dots,c_m} = \tau'|_{c_2,\dots,c_m} + \tau''|_{c_2,\dots,c_m}$, we arrive at:

$$\sigma[t.\mathit{from}] + (\Delta_{\boldsymbol{\kappa}}(\tau'|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau'|_{c_2,\dots,c_m}) + \Delta_{\boldsymbol{\kappa}}(t) + \Delta_{\boldsymbol{\kappa}}(\tau''|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau''|_{c_2,\dots,c_m}))[t.\mathit{from}] \geq 0 \quad (3.16)$$

By subtracting (3.16) from (3.15), and by commutativity of vector addition, we arrive at $\Delta_{\boldsymbol{\kappa}}(\tau''|_{c_2,\dots,c_m})[t.\mathit{from}] > 0$. Thus, there is a transition t' in $\tau''|_{c_2,\dots,c_m}$ and a rule $r' \in c_1$ such that $t'.to = r'.\mathit{from}$. We again arrived at the contradictory Case (i.A). Hence, transition t must be applicable to configuration σ''.

(i.C) Otherwise, neither $t.\mathit{from}$ nor $t.\mathit{to}$ belong to the set of local states affected by the rules from c_1, i.e., $\{t.\mathit{from}, t.\mathit{to}\} \cap \{\ell \mid \exists r \in c_1.\ r.\mathit{from} = \ell \vee r.\mathit{to} = \ell\}$ is empty. Then, schedule $\tau|_{c_1}$ does not change the counter $\boldsymbol{\kappa}[t.\mathit{from}]$, and $\Delta_{\boldsymbol{\kappa}}(\tau')[t.\mathit{from}] = \Delta_{\boldsymbol{\kappa}}(\tau'|_{c_2,\dots,c_m})[t.\mathit{from}]$. As t is applicable to $\tau'(\sigma)$, that is, $(\tau'(\sigma)).\boldsymbol{\kappa}[t.\mathit{from}] \geq t.\mathit{factor}$, we conclude that $\sigma''.\boldsymbol{\kappa}[t.\mathit{from}] \geq t.\mathit{factor}$.

(ii) We now show that $\sigma'' \models t.\varphi^{\text{rise}} \wedge t.\varphi^{\text{fall}}$. On contrary, assume that $\sigma'' \not\models t.\varphi^{\text{rise}} \wedge t.\varphi^{\text{fall}}$. There are two cases to consider.

If $\sigma'' \not\models t.\varphi^{\text{rise}}$. By definition, the shared variables are never decremented. As τ' is a prefix of τ, schedule $\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m}$ includes all transitions of τ'. Thus, $\Delta_{\mathbf{g}}(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m}) \geq \Delta_{\mathbf{g}}(\tau')$. From this and $\sigma'' \not\models t.\varphi^{\text{rise}}$, it follows that $\tau'(\sigma) \not\models t.\varphi^{\text{rise}}$. This contradicts applicability of τ to σ.

If $\sigma'' \not\models t.\varphi^{\text{fall}}$. Then, there is a guard $\varphi \in$ ' ('$t.\varphi^{\text{fall}}$) with $\tau''(\sigma) \not\models \varphi$. On one hand, $\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m}$ is applicable to σ. On the other hand, τ is applicable to σ.

We notice that $\Delta_{\mathbf{g}}(\tau) = \Delta_{\mathbf{g}}(\tau|_{c_1}) + \Delta_{\mathbf{g}}(\tau'|_{c_2,\dots,c_m}) + \Delta_{\mathbf{g}}(\tau''|_{c_2,\dots,c_m}) + \Delta_{\mathbf{g}}(t) \geq \Delta_{\mathbf{g}}(\tau|_{c_1}) + \Delta_{\mathbf{g}}(\tau'|_{c_2,\dots,c_m})$. As shared variables are never decreased, it follows that $(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m})(\sigma) \not\models \varphi$. Thus, $\omega(\sigma) \neq \omega(\tau(\sigma))$. This contradicts the assumption on that schedule τ is steady.

Having proved that, we conclude that transition t is applicable to configuration $(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m})(\sigma)$. Thus, by induction $(\tau|_{c_1} \cdot \tau|_{c_2,\dots,c_m})(\sigma)$ is applicable to σ. We conclude that Point *b)* of the theorem holds.

Proof of c). By the commutativity property of vector addition,

$$\Delta_{\boldsymbol{\kappa}}(\tau|_{c_1} \cdot \tau|_{c_2,\dots,c_m}) = \Delta_{\boldsymbol{\kappa}}(\tau|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau|_{c_2,\dots,c_m}) = \sum_{1 \leq i \leq |\tau|} \Delta_{\boldsymbol{\kappa}}(t_i) = \Delta_{\boldsymbol{\kappa}}(\tau).$$

Thus, $(\tau|_{c_1} \cdot \tau|_{c_2,\dots,c_m})(\sigma) = \tau(\sigma)$, and Point *c)* follows.

We have thus shown all three points of Lemma 3.12. □

3.10 Discussion

While an automated verification of FTDAs has been introduced in [KVW17], there remained two bottlenecks for scalability to larger and more complex protocols: First, counter abstraction can lead to spurious counterexamples. As counters range over a finite abstract domain, the semantics of abstract increment and decrements on the counters introduce non-determinism. For instance, the value of a counter can remain unchanged after applying an increment. Intuitively, processes or messages can be "added" or "lost", which results in that, e.g., in the abstract model the number of messages sent is smaller than the number of processes that have sent a message, which obviously is spurious behavior. Second, counter abstraction works well in practice only for processes with a few dozens of local states. It has been observed in [BMWK09] that counter abstraction does not scale to hundreds of local states, and the experiments in [KVW17] confirm this. A conjecture is that this is partly due to the many different interleavings, which result in a large search space.

To address these bottlenecks, we make two crucial *contributions* in this chapter:

1. To eliminate one of the two sources of spurious counterexamples, namely, the non-determinism added by abstract counters, we do bounded model checking using SMT solvers with linear integer arithmetic on the accelerated system, instead of SAT-based bounded model checking on the counter abstraction.

2. We reduce the search space dramatically: We introduce the notion of an *execution schema* that is defined as a sequence of local rules of the TA. By assigning to each rule of a schema an acceleration factor (that models the number of processes simultaneously executing the rule, possibly 0), one obtains a run of the counter system. Hence, due to parameterization, each schema represents infinitely many runs. We show how to construct a set of schemas whose set of reachable states coincides with the set of reachable states of the accelerated counter system.

The resulting method is depicted at the bottom of Figure 3.7. Our construction can be seen as an aggressive form of reduction, where each run has a similar run generated by a schema from the set. To show this, we capture the guards that are locked and unlocked in a *context*. Our key insight is that a bounded number of transitions changes the context in each run. For example, of all transitions increasing a variable x, at most one makes $x \geq n - t$ true, and at most one makes $x < t + 1$ false (the parameters n and t are fixed in a run, and shared variables can only be increased). We fix those transitions that change the context, and apply the ideas of reduction to the subexecutions between these transitions.

From a methodological viewpoint, our approach combines techniques from several areas including compact programs [Lub84], counter abstraction [PXZ02, BMWK09], completeness thresholds for bounded model checking [BCCZ99, CKOS04, KS03], partial order reduction [God90, Val91, Pel93, BKSS11], and offline reduction methods [Lip75, EF82]. Regarding counter automata, as discussed in [KKW18], our result entails *flattability* [LS05] of every counter system of threshold automata: a complete set of schemas immediately gives us a flat counter automaton. Hence, the acceleration-based semi-algorithms [LS05, BFLP08] should in principle terminate on the systems of TAs, though it did not always happen in our experiments. Similar to our SMT queries based on schemas, the *inductive data flow graphs* iDFG introduced in [FKP13] are a succinct representations of schedules (they call them traces) for systems where the number of processes (or threads) is fixed. The work presented in [FKP15] then considers parameterized verification. Further, our execution schemas are inspired by a general notion of *semi-linear path schemas* SLPS [LS04, LS05]. We construct a small complete set of schemas and thus a provably small SLPS. Besides, we distinguish counter systems and counter abstraction: the former counts processes as integers, while the latter uses counters over a finite abstract domain, e.g., $\{0, 1, \mathit{many}\}$ [PXZ02].

Our method uses integer counters and thus does not introduce spurious behavior due to counter abstraction, but still has spurious behavior due to data abstraction on complex FTDAs such as BOSCO, C1CS, and NBAC. In these cases, we manually refine the interval domain by adding new symbolic interval borders, see [JKS$^+$13a]. We believe that these intervals can be obtained directly from threshold automata, and no refinement is necessary. We leave this question to future work.

As predicted by the distributed algorithms literature, our tool finds counterexamples, when we relax the resilience condition. In contrast to counter abstraction, our technique gives us concrete values of the parameters and shows how many processes move at each step of the counterexample.

When doing experiments, we noticed that the only kinds of guards that cannot be treated by our optimizations and blow up the number of schemas are the guards that use independent shared variables. For instance, consider the guards $x_0 \geq n - t$ and $x_1 \geq n - t$ that are counting the number of 0's and 1's sent by the correct processes. Even though they are mutually exlusive under the resilience condition $n > 3t$, our tool has to explore all possible orderings of these guards. We are not aware of a reduction that would prevent our method from exploding in the number of schemas for this example.

Since the schemas can be checked independently, one can check them in parallel. Figure 3.8 shows a distribution of schemas along with the time needed to check an individual schema. There are only a few divergent schemas that required more than seven seconds to get checked, while the large portion of schemas require 1–3 seconds. Hence, a parallel implementation of the tool should verify the algorithms significantly faster, which is confirmed by the experiments discussed in Chapter 5, where the technique requires multiple repetition of verification procedures for slight modifications of the model.

CHAPTER 4

Parameterized Safety and Liveness

In Chapter 3 we have presented a technique for efficient checking of reachability properties. It is based on reduction, that is, the main ingredient is changing the order of transitions in an execution, and proving that the obtained execution reaches the same state as the original one. This technique cannot be directly used for more involved specifications, e.g., those where we also need to preserve a certain property along the execution, apart from reaching a target state.

This chapter deals with verification of safety and liveness properties. The technique presented here extends the one from Chapter 3. The challenge addressed in this chapter is formalized in Section 2.5, and we recall it here:

Challenge 2.2 (Parameterized unsafety & non-liveness). *Given a threshold automaton TA and an $\mathsf{ELTL}_{\mathsf{FT}}$ formula ψ, check whether there are parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, an initial configuration $\sigma_0 \in I$ with $\sigma_0.\mathbf{p} = \mathbf{p}$, and an infinite schedule τ of $\mathsf{Sys}(\mathsf{TA})$ applicable to σ_0 such that $\mathsf{path}(\sigma_0, \tau) \models \psi$.*

We solve this problem by showing how to reduce it to bounded model checking while guaranteeing completeness. To this end, we have to construct a bounded-length encoding of infinite schedules.

In the seminal paper [VW86], Vardi and Wolper showed that if a finite-state transition system M *violates* an LTL formula — which requires *all paths* to satisfy the formula — then there is a path in M that (i) violates the formula and (ii) has a lasso shape. That is, it starts with a finite prefix, and ends in an infinite cycle. As our logic $\mathsf{ELTL}_{\mathsf{FT}}$ specifies counterexamples to the properties of fault-tolerant distributed algorithms, we are interested in this result in the following form: if the transition system *satisfies*

```
algorithm parameterized_model_checking(TA, $\varphi$): // see Challenge 2.2
 $\mathcal{G}$ := cut_graph ($\varphi$) /* Section 4.2.1 */
 $\mathcal{H}$ := threshold_graph(TA) /* Section 4.2.3 */
 for each $\prec$ in topological_orderings($\mathcal{G} \cup \mathcal{H}$) do // e.g., using [CW95]
  check_one_order(TA, $\varphi$, $\mathcal{G}$, $\mathcal{H}$, $\prec$) /* Section 4.4 */
  if SMT_sat() then report the SMT model as a counterexample
```

Figure 4.1: A high-level description of the verification algorithm. For details of `check_one_order`, see Section 4.4.2 and Figure 4.9.

an ELTL formula — which requires *one path* to satisfy the formula — then M has a path that (i) *satisfies* the formula and (ii) has lasso shape.

As counter systems in general are infinite state, one cannot apply the results of [VW86] directly. However, recall that Proposition 2.2 states that paths in our setting visit finitely many configurations. Consequently, we show that a similar result to the one from [VW86] holds for counter systems of threshold automata and $\mathsf{ELTL}_{\mathsf{FT}}$. A complete proof of the following proposition can be found in Section 4.5.

Proposition 4.1. *Given a threshold automaton TA and an $\mathsf{ELTL}_{\mathsf{FT}}$ formula φ, if $Sys(TA) \models \mathbf{E}\varphi$, then there are an initial configuration $\sigma_1 \in I$ and a schedule $\tau \cdot \rho^\omega$ with the following properties:*

(a) the path satisfies the formula: $\mathsf{path}(\sigma_1, \tau \cdot \rho^\omega) \models \varphi$,

(b) application of ρ forms a cycle: $\rho^k(\tau(\sigma_1)) = \tau(\sigma_1)$ for $k \geq 0$.

Although Proposition 4.1 makes it sufficient to search for counterexamples of lasso shape, it is not sufficient for model checking: (i) counter systems are infinite state, so that state enumeration may not terminate, and (ii) Proposition 4.1 does not provide us with bounds on the length of the lassos needed for bounded model checking. Our strategy is to split a lasso schedule in finite segments and to find constraints on lasso schedules that satisfy an $\mathsf{ELTL}_{\mathsf{FT}}$ formula. Then we construct shorter (bounded length) segments.

In more detail, our method can be summarized in the following four steps:

- We observe that if $\mathsf{path}(\sigma_0, \tau) \models \psi$, then there is an initial state σ and two finite schedules ϑ and ρ (of unknown length) that can be used to construct an infinite (lasso-shaped) schedule $\vartheta \cdot \rho^\omega$, such that $\mathsf{path}(\sigma, \vartheta \cdot \rho^\omega) \models \psi$ (Proposition 4.1).
- Now given ϑ and ρ, we prove that we can use a ψ-specific reduction, to cut ϑ and ρ into subschedules $\vartheta_1, \dots, \vartheta_m$ and $\rho_1, \dots, \rho_n$, respectively so that the subschedules satisfy subformulas of ψ (Section 4.2).

- We use an adapted PARA^2 technique, specific to the subformulas of ψ, to construct representative schedules $\mathsf{rep}[\vartheta_i]$ and $\mathsf{rep}[\rho_j]$ that satisfy the required $\mathsf{ELTL_{FT}}$ formulas that are satisfied by ϑ_i and ρ_j, respectively for $1 \le i \le m$ and $1 \le j \le n$. Moreover, $\mathsf{rep}[\vartheta_i]$ and $\mathsf{rep}[\rho_j]$ are fixed sequences of rules, where bounds on the length of the sequences are known (Section 4.3).

- These fixed sequences of rules can be used to encode a query to the SMT solver (Section 4.4.1). We ask whether there is an applicable schedule in the counter system that satisfies the sequence of rules and ψ (Section 4.4.3). If the SMT solver reports unsatisfiability, there exists no counterexample.

Based on these theoretical results, our ByMC tool implements the high-level verification algorithm from Figure 4.1 (in the comments we give the sections that are concerned with the respective steps).

4.1 Overview of the Method

4.1.1 Enumerating Lassos

We characterize all possible shapes of lasso schedules that satisfy an $\mathsf{ELTL_{FT}}$ formula φ. These shapes are characterized by so-called *cut points*: We show that every lasso satisfying φ has a fixed number of cut points, one cut point per a subformula of φ that starts with $\mathbf{F}$. The configuration in the cut point of a subformula $\mathbf{F}\,\psi$ must satisfy ψ, and all configurations between two cut points must satisfy certain propositional formulas, which are extracted from the subformulas of φ that start with $\mathbf{G}$. Our notion of a cut point is motivated by extreme appearances of temporal operators [EVW02].

Example 4.1. Consider the $\mathsf{ELTL_{FT}}$ formula $\varphi \equiv \mathbf{E}\,\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$, where $a, \dots, e$ are propositional formulas, whose structure is not of interest in this section. Formula φ is satisfiable by certain paths that have lasso shapes, i.e., a path consists of a finite prefix and a loop, which is repeated infinitely. These lassos may differ in the actual occurrences of the propositions and the start of the loop: For instance, at some point, a holds, and since then b always holds, then d holds at some point, then e holds at some point, then the loop is entered, and c holds infinitely often inside the loop. This is the case (a) shown in Figure 4.2, where the configurations in the cut points A, B, C, and D must satisfy the propositional formulas a, d, e, and c respectively, and the configurations between A and F must satisfy the propositional formula b. This example does not restrict the propositions between the initial state and the cut point A, so that this lasso shape, for instance, also captures the path where b holds from the beginning. There are 20 different lasso shapes for φ, five of them are shown in the figure. We construct lasso shapes that are sufficient for finding a path satisfying an $\mathsf{ELTL_{FT}}$ formula. In this example, it is sufficient to consider lasso shapes (a) and (b), since the other shapes can be constructed from (a) and (b) by unrolling the loop several times. $\triangleleft$

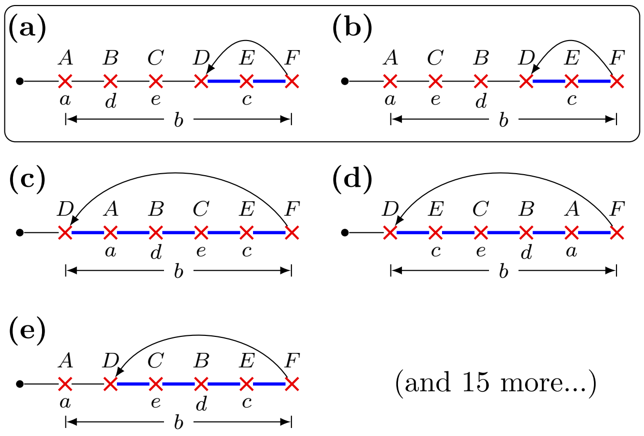


Figure 4.2: The shapes of lassos that satisfy the formula $\mathbf{E\,F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G\,F}\,c)$. The crosses show cut points for: (A) formula $\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G\,F}\,c)$, (B) formula $\mathbf{F}\,d$, (C) formula $\mathbf{F}\,e$, (D) loop start, (E) formula $\mathbf{F}\,c$, and (F) loop end.

4.1.2 Inadequacy of the PARA^2 technique from Chapter 3

Our verification approach focuses on counterexamples, and as discussed in Section 2.5, negations of specifications are expressed in $\mathsf{ELTL_{FT}}$. In the case of reachability properties, counterexamples are finite schedules reaching a bad state from an initial state. We presented an efficient method for finding counterexamples to reachability in Chapter 3. It is based on the short counterexample property. Namely, we proved that for each threshold automaton, there is a constant d such that if there is a schedule that reaches a bad state, then there must also exist an accelerated schedule that reaches that state in at most d transitions (i.e., d is the diameter of the counter system). Recall that the proof in Chapter 3 is based on the following three steps:

1. each finite schedule (which may or may not be a counterexample), can be divided into a few steady schedules,
2. for each of these steady schedules we find a representative, i.e., an accelerated schedule of bounded length, with the same starting and ending configurations as the original schedule,
3. at the end, all these representatives are concatenated in the same order as the original steady schedules.

This result guarantees that the system is correct if no counterexample to reachability properties is found using bounded model checking with bound d. In this section, we extend the technique from Point 2 from reachability properties to $\mathsf{ELTL_{FT}}$ formulas. The central result regarding Point 2 is the following proposition which is a specialization of Theorem 3.11 from page 51:

$$\begin{tikzcd}
\tau_{\mathrm{up}}: & \sigma_1 \arrow[rr, "\boldsymbol{\kappa}[\ell]{++}"] & {} \arrow[ddrr, dashed] & \sigma_2 \arrow[rr, "\boldsymbol{\kappa}[\ell]{--}"] & {} \arrow[ddll, dashed] & \sigma_3 \\
 & & & & & \\
\tau_{\mathrm{down}}: & \sigma_1 \arrow[rr, "\boldsymbol{\kappa}[\ell]{--}"'] & {} & \sigma_2' \arrow[rr, "\boldsymbol{\kappa}[\ell]{++}"'] & {} & \sigma_3
\end{tikzcd}$$

Figure 4.3: Changing the order of transitions can violate $\mathsf{ELTL_{FT}}$ formulas. If $\sigma_1.\boldsymbol{\kappa}[\ell] = 1$, then for the upper schedule τ_{up} holds that $\mathsf{Cfgs}(\sigma_1, \tau_{\mathrm{up}}) \models \boldsymbol{\kappa}[\ell] > 0$, while for the lower one this is not the case, because $\sigma_2' \not\models \boldsymbol{\kappa}[\ell] > 0$.

Proposition 4.2. *Let* $\mathit{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$ *be a threshold automaton. For every configuration* σ *and every steady schedule* τ *applicable to* σ, *there exists a steady schedule* $\mathsf{srep}[\sigma, \tau]$ *with the following properties:*

a) $\mathsf{srep}[\sigma, \tau]$ *is applicable to* σ, *and* $\mathsf{srep}[\sigma, \tau](\sigma) = \tau(\sigma)$, *and*

b) $|\mathsf{srep}[\sigma, \tau]| \leq 2 \cdot |\mathcal{R}|$.

We observe that the proposition refers only to σ and $\tau(\sigma)$, while it ignores intermediate configurations. However, for $\mathsf{ELTL_{FT}}$ formulas, we have to consider all configurations in a schedule, and not just the first and the last one.

Example 4.2. Figure 4.3 shows the result of swapping transitions. The PARA^2 approach for reachability from Chapter 3 is only concerned with the first and the last configurations: it uses the property that after swapping transitions, σ_3 is still reached from σ_1. The argument of PARA^2 for reachability does not take into account the fact that the resulting path visits a different intermediate state (σ_2' instead of σ_2). However, swapping transitions may change the evaluation of $\mathsf{ELTL_{FT}}$ formulas, e.g., $\mathbf{G}\,(\boldsymbol{\kappa}[\ell] > 0)$. If $\sigma_1.\boldsymbol{\kappa}[\ell] = 1$, then $\sigma_2.\boldsymbol{\kappa}[\ell] > 0$, while $\sigma_2'.\boldsymbol{\kappa}[\ell] = 0$. $\lhd$

4.1.3 The Short Counterexample Property

Another challenge in verification of $\mathsf{ELTL_{FT}}$ formulas is that counterexamples to liveness properties are infinite paths. As discussed previously in this chapter, we consider infinite paths of lasso shape $\vartheta \cdot \rho^\omega$. For a finite part of a schedule, $\vartheta \cdot \rho$, satisfying an $\mathsf{ELTL_{FT}}$ formula, we show the existence of a new schedule, $\vartheta' \cdot \rho'$, of bounded length satisfying the same formula as the original one. Regarding the shortening, this approach uses a similar idea as the one from Chapter 3. We follow a modified steps from reachability analysis:

1. We split $\vartheta \cdot \rho$ into several steady schedules, using cut points introduced in Section 4.2. The cut points depend not only on threshold guards, but also on the $\mathsf{ELTL_{FT}}$ formula φ representing the negation of a specification we want to check. Given such a steady schedule τ, each configuration of τ satisfies a set of propositional subformulas of φ, which are covered by the operator $\mathbf{G}$ in φ.

2. For each of these steady schedules we find a representative, that is, an accelerated schedule of bounded length that satisfies the necessary propositional subformulas as in the original schedule (i.e., not just that starting and ending configurations coincide).

3. We concatenate the obtained representatives in the original order.

In Section 4.3 we present the mathematical details for obtaining these representative schedules. In Sections 4.3.1–4.3.3, we prove different cases in Theorem 4.34 and Theorem 4.37, that taken together establish our following main theorem:

Theorem 4.3. *Let $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$ be a threshold automaton, and let $Locs \subseteq \mathcal{L}$ be a set of locations. Let σ be a configuration, let τ be a steady conventional schedule applicable to σ, and let ψ be one of the following formulas:*

$$\bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0, \quad \textit{or} \quad \bigwedge_{\ell \in Locs} \boldsymbol{\kappa}[\ell] = 0.$$

If all configurations visited by τ from σ satisfy ψ, i.e., $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, then there is a steady representative schedule $\mathsf{repr}[\psi, \sigma, \tau]$ with the following properties:

a) The representative is applicable, and ends in the same final state: $\mathsf{repr}[\psi, \sigma, \tau]$ is applicable to σ, and $\mathsf{repr}[\psi, \sigma, \tau](\sigma) = \tau(\sigma)$,

b) The representative has bounded length: $|\mathsf{repr}[\psi, \sigma, \tau]| \leq 6 \cdot |\mathcal{R}|$,

c) The representative maintains the formula ψ, i.e., $\mathsf{Cfgs}(\sigma, \mathsf{repr}[\psi, \sigma, \tau]) \models \psi$,

d) The representative is a concatenation of three representative schedules srep from Proposition 4.2: there exist τ_1, τ_2 and τ_3, (possibly empty) subschedules of τ, such that $\tau_1 \cdot \tau_2 \cdot \tau_3$ is applicable to σ, and it holds that $(\tau_1 \cdot \tau_2 \cdot \tau_3)(\sigma) = \tau(\sigma)$, and

$$\mathsf{repr}[\psi, \sigma, \tau] = \mathsf{srep}[\sigma, \tau_1] \cdot \mathsf{srep}[\tau_1(\sigma), \tau_2] \cdot \mathsf{srep}[(\tau_1 \cdot \tau_2)(\sigma), \tau_3].$$

Our approach is slightly different in the case when the formula ψ has a more complex form: $\bigwedge_{1 \leq m \leq n} \bigvee_{\ell \in Locs_m} \boldsymbol{\kappa}[\ell] \neq 0$, for $Locs_m \subseteq \mathcal{L}$, where $1 \leq m \leq n$ and $n \in \mathbb{N}$. In this case, our proof requires the schedule τ to have sufficiently large counter values. To ensure that there is an infinite schedule with sufficiently large counter values, we first prove that if a counterexample exists in a small system, there also exists one in a larger system, that is, we consider configurations where each counter is multiplied with a constant *finite multiplier* μ. For resilience conditions that do not correspond to parameterized systems (i.e., fix the system size to, e.g., $n = 4$) or pathological threshold automata, such multipliers may not exist. However, all our benchmarks have multipliers, and existence of multipliers can easily be checked using simple queries to SMT solvers in preprocessing. This additional restriction leads to slightly smaller bounds on the lengths of representative schedules:

Theorem 4.4. *Fix a threshold automaton $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$ that has a finite multiplier μ, and a configuration σ. For an $n \in \mathbb{N}$, fix sets of locations $Locs_m \subseteq \mathcal{L}$ for $1 \le m \le n$. If we have*

$$\psi = \bigwedge_{1 \le m \le n} \bigvee_{\ell \in Locs_m} \boldsymbol{\kappa}[\ell] \neq 0,$$

then for every steady conventional schedule τ, applicable to σ, with $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, there exists a schedule $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ with the following properties:

a) The representative is applicable and ends in the same final state: $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ is a steady schedule applicable to $\mu\sigma$, and $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau](\mu\sigma) = \mu\tau(\mu\sigma)$,

b) The representative has bounded length: $|\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]| \le 4 \cdot |\mathcal{R}|$,

c) The representative maintains the formula ψ, i.e., $\mathsf{Cfgs}(\mu\sigma, \mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]) \models \psi$,

d) The representative is a concatenation of two representative schedules srep from Proposition 4.2:

$$\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau] = \mathsf{srep}[\mu\sigma, \tau] \cdot \mathsf{srep}[\tau(\mu\sigma), (\mu - 1)\tau].$$

The main technical challenge for proving Theorems 4.3 and 4.4 is that we want to swap transitions and maintain $\mathsf{ELTL_{FT}}$ formulas at the same time. As discussed in Example 4.2, simply applying the ideas from the reachability analysis in Chapter 3 is not sufficient.

We address this challenge in Section 4.3 by more refined swapping strategies depending on the property ψ of Theorem 4.3. For instance, the intuition behind $\bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$ is that in a given distributed algorithm, there should always be at least one process in one of the states in *Locs*. Hence, we would like to consider individual processes, but in the context of counter systems. Therefore, we introduce a mathematical notion of a *thread*, which is a schedule that can be executed by an individual process. A thread is then characterized depending on whether it starts in *Locs*, ends in *Locs*, or visits *Locs* at some intermediate step. Based on this characterization, we show that $\mathsf{ELTL_{FT}}$ formulas are preserved if we move carefully chosen threads to the beginning of a steady schedule (intuitively, this corresponds to τ_1, and τ_2 from Theorem 4.3). Then, we replace the threads, one by one, by their representative schedules from Proposition 4.2, and append another representative schedule for the remainder of the schedule. In this way, we then obtain the representative schedules in Theorem 4.3(d).

Example 4.3. We consider the TA in Figure 2.1, and show how a schedule $\tau = (r_1, 1), (r_6, 1), (r_4, 1), (r_2, 1), (r_4, 1)$ applicable to σ_1, with $\tau(\sigma_1) = \sigma_2$ can be shortened. Figure 4.4 follows this example where τ is the upper schedule. Note that $\mathsf{Cfgs}(\sigma_1, \tau) \models \boldsymbol{\kappa}[\ell_2] \neq 0$. We want to construct a shorter schedule that produces a path whose all visited configurations satisfy the same formula $\boldsymbol{\kappa}[\ell_2] \neq 0$.

In our theory, subschedule $(r_1, 1), (r_4, 1)$ is a thread of σ_1 and τ for two reasons: (1) the counter of the starting local state of $(r_1, 1)$ is greater than 0, i.e., $\sigma_1.\boldsymbol{\kappa}[\ell_0] = 1$, and (2)

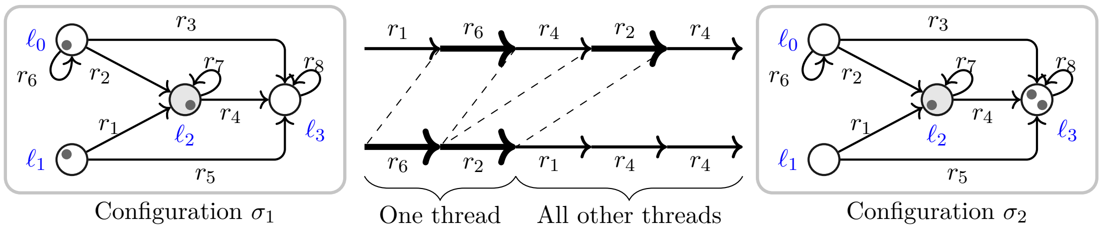


Figure 4.4: Example of constructing a representative schedule by moving a thread to the beginning. The number of dots in the local states correspond to counter values, i.e., $\sigma_1.\boldsymbol{\kappa}[\ell_0] = \sigma_1.\boldsymbol{\kappa}[\ell_1] = \sigma_1.\boldsymbol{\kappa}[\ell_2] = 1$ and $\sigma_1.\boldsymbol{\kappa}[\ell_3] = 0$.

it is a sequence of rules in the control flow of the threshold automaton, i.e., it starts from ℓ_0, then uses $(r_1, 1)$ to go to local state ℓ_2 and then $(r_4, 1)$ to arrive at ℓ_3. The intuition of (2) is that a thread corresponds to a process that executes the threshold automaton. Similarly, $(r_6, 1), (r_2, 1)$ and $(r_4, 1)$ are also threads of σ_1 and τ. In fact, we can show that each schedule can be decomposed into threads. Based on this, we analyze which local states are visited when a thread is executed.

Our formula $\mathsf{Cfgs}(\sigma_1, \tau) \models \boldsymbol{\kappa}[\ell_2] \neq 0$ considers ℓ_2. Thus, we are interested in a thread that ends at ℓ_2, because after executing this thread, intuitively there will always be at least one process in ℓ_2, i.e., the counter $\boldsymbol{\kappa}[\ell_2]$ will be nonzero, as required. Such a thread will be moved to the beginning. We find that thread $(r_6, 1), (r_2, 1)$ meets this requirement. Similarly, we are also interested in a thread that starts from ℓ_2. Before we execute such a thread, at least one process must always be in ℓ_2, i.e., $\boldsymbol{\kappa}[\ell_2]$ will be nonzero. For this, we single out the thread $(r_4, 1)$, as it starts from ℓ_2. These conclusions are formalized in Lemma 4.25 and Lemma 4.26.

Independently of the actual positions of these threads within a schedule, our condition $\boldsymbol{\kappa}[\ell_2] \neq 0$ is true *before* $(r_4, 1)$ starts, and *after* $(r_6, 1), (r_2, 1)$ ends. Hence, we move the thread $(r_6, 1), (r_2, 1)$ to the beginning, and obtain a schedule that ensures our condition in all visited configurations; cf. the lower schedule in Figure 4.4. Then we replace the thread $(r_6, 1), (r_2, 1)$, by a representative schedule from Proposition 4.2, and the remaining part $(r_1, 1)$, $(r_4, 1)$, $(r_4, 1)$, by another one. Indeed in our example, we could merge $(r_4, 1), (r_4, 1)$ into one accelerated transition $(r_4, 2)$ and obtain a schedule which is shorter than τ while maintaining $\boldsymbol{\kappa}[\ell_2] \neq 0$. $\triangleleft$

4.2 Shapes of Schedules that Satisfy $\mathsf{ELTL}_{\mathsf{FT}}$

As discussed in 4.1.1, the first step of our method is to enumerate all possible lasso-shaped schedules that satisfy a given $\mathsf{ELTL}_{\mathsf{FT}}$ formula φ. The goal is to introduce a *cut graph* that follows the nature of the $\mathsf{ELTL}_{\mathsf{FT}}$ formula (Section 4.2.1), and use it to impose the constraints on the lasso (Section 4.2.2). Finally, we also need to take into account the evaluation of threshold guards by introducing a *threshold graph* (Section 4.2.3).

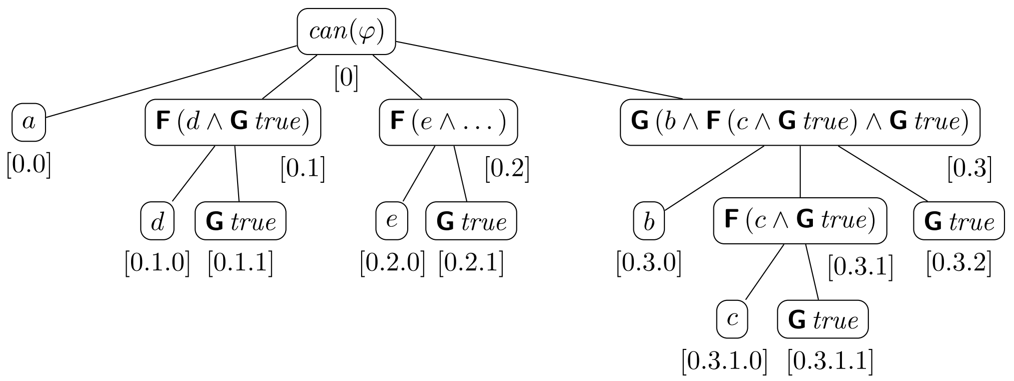

$can(\varphi)$
[0]
a
[0.0]
$\mathbf{F}\,(d \land \mathbf{G}\,true)$
[0.1]
$\mathbf{F}\,(e \land \dots)$
[0.2]
$\mathbf{G}\,(b \land \mathbf{F}\,(c \land \mathbf{G}\,true) \land \mathbf{G}\,true)$
[0.3]
d
[0.1.0]
$\mathbf{G}\,true$
[0.1.1]
e
[0.2.0]
$\mathbf{G}\,true$
[0.2.1]
b
[0.3.0]
$\mathbf{F}\,(c \land \mathbf{G}\,true)$
[0.3.1]
$\mathbf{G}\,true$
[0.3.2]
c
[0.3.1.0]
$\mathbf{G}\,true$
[0.3.1.1]

Figure 4.5: A canonical syntax tree of the $\mathsf{ELTL_{FT}}$ formula $\varphi \equiv \mathbf{F}\,(a \land \mathbf{F}\,d \land \mathbf{F}\,e \land \mathbf{G}\,b \land \mathbf{G}\,\mathbf{F}\,c)$ considered in Example 4.1. The labels $[w]$ denote identifiers of the tree nodes.

4.2.1 Characterizing Shapes of Lasso Schedules by Cut Graphs

We now construct a cut graph of an $\mathsf{ELTL_{FT}}$ formula: Cut graphs constrain the orders in which subformulas that start with the operator $\mathbf{F}$ are witnessed by configurations. The nodes of a cut graph correspond to cut points, while the edges constrain the order between the cut points. Using cut points, we give necessary and sufficient conditions for a lasso to satisfy an $\mathsf{ELTL_{FT}}$ formula in Theorems 4.7 and 4.8. Before defining cut graphs, we give the technical definitions of canonical formulas and canonical syntax trees.

Definition 4.1. *We inductively define canonical $\mathsf{ELTL_{FT}}$ formulas:*

- *if p is a propositional formula, then the formula $p \land \mathbf{G}\,true$ is a canonical formula of rank 0,*
- *if p is a propositional formula and formulas $\psi_1, \dots, \psi_k$ are canonical formulas (of any rank) for some $k \geq 1$, then the formula $p \land \mathbf{F}\,\psi_1 \land \dots \land \mathbf{F}\,\psi_k \land \mathbf{G}\,true$ is a canonical formula of rank 1,*
- *if p is a propositional formula and formulas $\psi_1, \dots, \psi_k$ are canonical formulas (of any rank) for some $k \geq 0$, and ψ_{k+1} is a canonical formula of rank 0 or 1, then the formula $p \land \mathbf{F}\,\psi_1 \land \dots \land \mathbf{F}\,\psi_k \land \mathbf{G}\,\psi_{k+1}$ is a canonical formula of rank 2.*

Example 4.4. Let p and q be propositional formulas. The formulas $p \land \mathbf{G}\,true$ and $true \land \mathbf{F}\,(q \land \mathbf{G}\,true) \land \mathbf{G}\,(p \land \mathbf{G}\,true)$ are canonical, while the formulas p, $\mathbf{F}\,q$, and $\mathbf{G}\,p$ are not canonical. Continuing Example 4.1, the canonical version of the formula $\mathbf{F}\,(a \land \mathbf{F}\,d \land \mathbf{F}\,e \land \mathbf{G}\,b \land \mathbf{G}\,\mathbf{F}\,c)$ is the formula $\mathbf{F}\,(a \land \mathbf{F}\,(d \land \mathbf{G}\,true) \land \mathbf{F}\,(e \land \mathbf{G}\,true) \land \mathbf{G}\,(b \land \mathbf{F}\,(c \land \mathbf{G}\,true) \land \mathbf{G}\,true))$.
$\triangleleft$

We will use formulas in the following canonical form in order to simplify presentation.

Observation 1. The properties of canonical $\mathsf{ELTL_{FT}}$ formulas:

1. Every canonical formula consists of canonical subformulas of the form $p \wedge \mathsf{F}\,\psi_1 \wedge \dots \wedge \mathsf{F}\,\psi_k \wedge \mathsf{G}\,\psi_{k+1}$ for some $k \geq 0$, for a propositional formula p, canonical formulas $\psi_1, \dots, \psi_k$, and a formula ψ_{k+1} that is either canonical, or equals to *true*.
2. If a canonical formula contains a subformula $\mathsf{G}\,(\dots \wedge \mathsf{G}\,\psi)$, then ψ equals *true*.

As expected, every $\mathsf{ELTL}_{\mathsf{FT}}$ formula has an equivalent canonical formula:

Proposition 4.5. *There is a function can* : $\mathsf{ELTL}_{\mathsf{FT}} \to \mathsf{ELTL}_{\mathsf{FT}}$ *that produces for each formula* $\varphi \in \mathsf{ELTL}_{\mathsf{FT}}$ *an equivalent canonical formula* $can(\varphi)$*.*

For an $\mathsf{ELTL}_{\mathsf{FT}}$ formula, there may be several equivalent canonical formulas, e.g., $p \wedge \mathsf{F}\,(q \wedge \mathsf{G}\,true) \wedge \mathsf{F}\,(p \wedge \mathsf{G}\,true) \wedge \mathsf{G}\,true$ and $p \wedge \mathsf{F}\,(p \wedge \mathsf{G}\,true) \wedge \mathsf{F}\,(q \wedge \mathsf{G}\,true) \wedge \mathsf{G}\,true$ differ in the order of F-subformulas. With the function *can* we fix one such a formula.

Indeed, a canonical formula usually contains many trivially valid subformulas, e.g., $can(\mathsf{F}\,p)$ is $true \wedge \mathsf{F}\,(p \wedge \mathsf{G}\,true) \wedge \mathsf{G}\,true$. In the following, we often omit such trivially valid formulas for ease in presentation.

Canonical syntax trees. The canonical syntax tree of the formula introduced in Example 4.1 is shown in Figure 4.5. With $\mathbb{N}_0^*$ we denote the set of all finite words over natural numbers — these words are used as node identifiers.

Definition 4.2. *The* canonical syntax tree *of a formula* $\varphi \in \mathsf{ELTL}_{\mathsf{FT}}$ *is the set* $\mathcal{T}(\varphi) \subseteq \mathsf{ELTL}_{\mathsf{FT}} \times \mathbb{N}_0^*$ *constructed inductively as follows:*

1. *The tree contains the root node labeled with the canonical formula* $can(\varphi)$ *and id* 0*, that is,* $\langle can(\varphi), 0\rangle \in \mathcal{T}(\varphi)$*.*
2. *Consider a tree node* $\langle \psi, w\rangle \in \mathcal{T}(\varphi)$ *such that for some canonical formula* $\psi' \in \mathsf{ELTL}_{\mathsf{FT}}$ *one of the following holds: (a)* $\psi = \psi' = can(\varphi)$*, or (b)* $\psi = \mathsf{F}\psi'$*, or (c)* $\psi = \mathsf{G}\psi'$*.*

   If ψ' *is* $p \wedge \mathsf{F}\psi_1 \wedge \dots \wedge \mathsf{F}\psi_k \wedge \mathsf{G}\psi_{k+1}$ *for some* $k \geq 0$*, then the tree* $\mathcal{T}(\varphi)$ *contains a child node for each of the conjuncts of* ψ'*, that is,* $\langle p, w.0\rangle \in \mathcal{T}(\varphi)$*, as well as* $\langle \mathsf{F}\psi_i, w.i\rangle \in \mathcal{T}(\varphi)$ *and* $\langle \mathsf{G}\psi_j, w.j\rangle \in \mathcal{T}(\varphi)$ *for* $1 \leq i \leq k$ *and* $j = k + 1$*.*

Observation 2. The canonical syntax tree $\mathcal{T}(\varphi)$ of an $\mathsf{ELTL}_{\mathsf{FT}}$ formula φ has the following properties:

- Every node $\langle \psi, w\rangle$ has the unique identifier w, which encodes the path to the node from the root.
- Every intermediate node is labeled with a temporal operator F or G over the conjunction of the formulas in the children nodes.

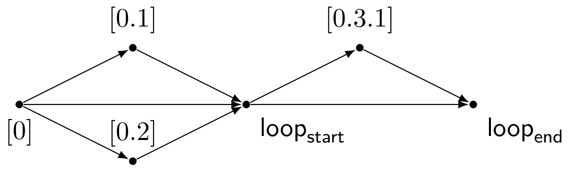


Figure 4.6: The cut graph of the canonical syntax tree in Figure 4.5

- The root node is labeled with the formula φ itself, and φ is equivalent to the conjunction of the root's children formulas, possibly preceded with a temporal operator $\mathbf{F}$ or $\mathbf{G}$.

The temporal formulas that appear under the operator $\mathbf{G}$ have to be dealt with by the loop part of a lasso. To formalize this, we say that a node with id $w \in \mathbb{N}_0^*$ is *covered* by a $\mathbf{G}$-node, if w can be split into two words $u_1, u_2 \in \mathbb{N}_0^*$ with $w = u_1.u_2$, and there is a formula $\psi \in \mathsf{ELTL_{FT}}$ such that $\langle \mathbf{G}\,\psi, u_1\rangle \in \mathcal{T}(\varphi)$.

Cut graphs. Using the canonical syntax tree $\mathcal{T}(\varphi)$ of a formula φ, we capture in a so-called *cut graph* the possible orders in which formulas $\mathbf{F}\,\psi$ should be witnessed by configurations of a lasso-shaped path. We will then use the occurrences of the formula ψ to cut the lasso into bounded finite schedules.

Example 4.5. Figure 4.6 shows the cut graph of the canonical syntax tree in Figure 4.5. It consists of tree node ids for subformulas starting with $\mathbf{F}$, and two special nodes for the start and the end of the loop. In the cut graph, the node with id 0 precedes the node with id 0.1, since at least one configuration satisfying $(a \wedge \mathbf{F}\,(d \wedge \dots) \wedge \dots)$ should occur on a path before (or at the same moment as) a state satisfying $(d \wedge \dots)$. Similarly, the node with id 0 precedes the node with id 0.2. The nodes with ids 0.1 and 0.2 do not have to precede each other, as the formulas d and e can be satisfied in either order. Since the nodes with the ids 0, 0.1, and 0.2 are not covered by a $\mathbf{G}$-node, they both precede the loop start. The loop start precedes the node with id 0.3.1, as this node is covered by a $\mathbf{G}$-node. $\triangleleft$

Definition 4.3. *The* cut graph $\mathcal{G}(\varphi)$ *of an $\mathsf{ELTL_{FT}}$ formula is a directed acyclic graph $(\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$ with the following properties:*

1. *The set of nodes $\mathcal{V}_\mathcal{G} = \{\mathsf{loop_{start}}, \mathsf{loop_{end}}\} \cup \{w \in \mathbb{N}_0^* \mid \exists \psi.\ \langle \mathbf{F}\,\psi, w\rangle \in \mathcal{T}(\varphi)\}$ contains the tree ids that label $\mathbf{F}$-formulas and two special nodes $\mathsf{loop_{start}}$ and $\mathsf{loop_{end}}$, which denote the start and the end of the loop respectively.*

2. *The set of edges $\mathcal{E}_\mathcal{G}$ satisfies the following constraints:*

   a) *Each tree node $\langle \mathbf{F}\,\psi, w\rangle \in \mathcal{T}(\varphi)$ that is* not *covered by a $\mathbf{G}$-node precedes the loop start, i.e., $(w, \mathsf{loop_{start}}) \in \mathcal{E}_\mathcal{G}$.*

b) *For each tree node $\langle \mathbf{F}\,\psi, w\rangle \in \mathcal{T}(\varphi)$ covered by a $\mathbf{G}$-node:*
   - *the loop start precedes w, i.e., $(\mathsf{loop}_{\mathsf{start}}, w) \in \mathcal{E}_\mathcal{G}$, and*
   - *w precedes the loop end, i.e., $(w, \mathsf{loop}_{\mathsf{end}}) \in \mathcal{E}_\mathcal{G}$.*

c) *For each pair of tree nodes $\langle \mathbf{F}\,\psi_1, w\rangle, \langle \mathbf{F}\,\psi_2, w.i\rangle \in \mathcal{T}(\varphi)$ not covered by a $\mathbf{G}$-node, we require $(w, w.i) \in \mathcal{E}_\mathcal{G}$.*

d) *For each pair of tree nodes $\langle \mathbf{F}\,\psi_1, w_1\rangle, \langle \mathbf{F}\,\psi_2, w_2\rangle \in \mathcal{T}(\varphi)$ that are both covered by a $\mathbf{G}$-node, we require either $(w_1, w_2) \in \mathcal{E}_\mathcal{G}$, or $(w_2, w_1) \in \mathcal{E}_\mathcal{G}$ (but not both).*

Definition 4.4. *Given a lasso $\tau \cdot \rho^\omega$ and a cut graph $\mathcal{G}(\varphi) = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, we call a function $\zeta : \mathcal{V}_\mathcal{G} \to \{0, \dots, |\tau| + |\rho| - 1\}$ a* cut function*, if the following holds:*

- *$\zeta(\mathsf{loop}_{\mathsf{start}}) = |\tau|$ and $\zeta(\mathsf{loop}_{\mathsf{end}}) = |\tau| + |\rho| - 1$,*
- *if $(v, v') \in \mathcal{E}_\mathcal{G}$, then $\zeta(v) \le \zeta(v')$.*

We call the indices $\{\zeta(v) \mid v \in \mathcal{V}_\mathcal{G}\}$ the *cut points*. Given a schedule τ and an index $k : 0 \le k < |\tau| + |\rho|$, we say that the index k *cuts* τ into π' and π'', if $\tau = \pi' \cdot \pi''$ and $|\pi'| = k$.

Informally, for a tree node $\langle \mathbf{F}\,\psi, w\rangle \in \mathcal{T}(\varphi)$, a cut point $\zeta(w)$ witnesses satisfaction of $\mathbf{F}\,\psi$, that is, the formula ψ holds at the configuration located at the cut point. It might seem that Definitions 4.3 and 4.4 are too restrictive. For instance, assume that the node $\langle \mathbf{F}\,\psi, w\rangle$ is not covered by a $\mathbf{G}$-node, and there is a lasso schedule $\tau \cdot \rho^\omega$ that satisfies the formula φ at a configuration σ. It is possible that the formula ψ is witnessed only by a cut point inside the loop. At the same time, Definition 4.4 forces $\zeta(w) \le \zeta(\mathsf{loop}_{\mathsf{start}})$. We show that this problem is resolved by unwinding the loop K times for some $K \ge 0$, so that there is a cut function for the lasso with the prefix $\tau \cdot \rho^K$ and the loop ρ:

Proposition 4.6. *Let φ be an $\mathsf{ELTL}_{\mathsf{FT}}$ formula, σ be a configuration and $\tau \cdot \rho^\omega$ be a lasso schedule applicable to σ such that $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$ holds. There is a constant $K \ge 0$ and a cut function ζ such that for every $\langle \mathbf{F}\,\psi, w\rangle \in \mathcal{G}(\mathcal{T}(\varphi))$ if $\zeta(w)$ cuts $(\tau \cdot \rho^K) \cdot \rho$ into π' and π'', then ψ is satisfied at the cut point, that is, $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi$.*

Proof Sketch: The detailed proof is given in Section 4.5. We will present the required constant $K \ge 0$ and the cut function ζ. To this end, we use extreme appearances of $\mathbf{F}$-formulas (cf. [EVW02, Sec. 4.3]) and use them to find ζ. An extreme appearance of a formula $\mathbf{F}\,\psi$ is the furthest point in the lasso that still witnesses ψ. There might be a subformula that is required to be witnessed in the prefix, but in $\tau \cdot \rho^\omega$ it is only witnessed by the loop. To resolve this, we replace τ by a a longer prefix $\tau \cdot \rho^K$, by unrolling the loop ρ several times; more precisely, K times, where K is the number of nodes that should precede the lasso start. In other words, if all extreme appearances of the nodes happen to be in the loop part, and they appear in the order that is against the topological order of the graph $\mathcal{G}(\mathcal{T}(\varphi))$, we unroll the loop K times (the number of nodes that have to be in the prefix) to find the prefix, in which the nodes respect the topological order of the graph. In the unrolled schedule we can now find extreme appearances of the required subformulas in the prefix. □

We show that to satisfy an $\mathsf{ELTL_{FT}}$ formula, a lasso should (i) satisfy propositional subformulas of $\mathbf{F}$-formulas in the respective cut points, and (ii) maintain the propositional formulas of $\mathbf{G}$-formulas from some cut point on. This is formalized as a witness.

In the following definition, we use a short-hand notation for propositional subformulas: given an $\mathsf{ELTL_{FT}}$ formula ψ and its canonical form $can(\psi) = \psi_0 \wedge \mathbf{F}\,\psi_1 \wedge \cdots \wedge \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$, we use the notation $prop(\psi)$ to denote the formula ψ_0.

Definition 4.5. *Given a configuration σ, a lasso $\tau \cdot \rho^\omega$ applicable to σ, and an $\mathsf{ELTL_{FT}}$ formula φ, a cut function ζ of $\mathcal{G}(\mathcal{T}(\varphi))$ is a* witness *of $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$, if the three conditions hold:*

(C1) *For $can(\varphi) \equiv \psi_0 \wedge \mathbf{F}\psi_1 \wedge \cdots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\psi_{k+1}$:*

a) $\sigma \models \psi_0$, and

b) $\mathsf{Cfgs}(\sigma, \tau \cdot \rho) \models prop(\psi_{k+1})$.

(C2) *For $\langle \mathbf{F}\psi, v\rangle \in \mathcal{T}(\varphi)$ with $\zeta(v) < |\tau|$, if $\zeta(v)$ cuts $\tau \cdot \rho$ into π' and π'' and $\psi \equiv \psi_0 \wedge \mathbf{F}\psi_1 \wedge \cdots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\psi_{k+1}$, then:*

a) $\pi'(\sigma) \models \psi_0$, and

b) $\mathsf{Cfgs}(\pi'(\sigma), \pi'') \models prop(\psi_{k+1})$.

(C3) *For $\langle \mathbf{F}\psi, v\rangle \in \mathcal{T}(\varphi)$ with $\zeta(v) \geq |\tau|$, if $\zeta(v)$ cuts $\tau \cdot \rho$ into π' and π'' and $\psi \equiv \psi_0 \wedge \mathbf{F}\psi_1 \wedge \cdots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\psi_{k+1}$, then:*

a) $\pi'(\sigma) \models \psi_0$, and

b) $\mathsf{Cfgs}(\tau(\sigma), \rho) \models prop(\psi_{k+1})$.

Conditions (a) require that propositional formulas hold in a configuration, while conditions (b) require that propositional formulas hold on a finite suffix. Hence, to ensure that a cut function constitutes a witness, one has to check the configurations of a *fixed number of finite* paths (between the cut points). This property is crucial for the path reduction (see Section 4.1.3). Theorems 4.7 and 4.8 show that the existence of a witness is a sound and complete criterion for the existence of a lasso satisfying an $\mathsf{ELTL_{FT}}$ formula.

Theorem 4.7 (Soundness)**.** *Let σ be a configuration, $\tau \cdot \rho^\omega$ be a lasso applicable to σ, and φ be an $\mathsf{ELTL_{FT}}$ formula. If there is a witness of $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$, then the lasso $\tau \cdot \rho^\omega$ satisfies φ, that is $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$.*

Theorem 4.8 (Completeness)**.** *Let φ be an $\mathsf{ELTL}_{\mathsf{FT}}$ formula, σ be a configuration and $\tau \cdot \rho^\omega$ be a lasso applicable to σ such that $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$ holds. There is a witness of $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho^\omega) \models \varphi$ for some $K \geq 0$.*

Theorem 4.7 is proven for subformulas of φ by structural induction on the intermediate nodes of the canonical syntax tree. In the proof of Theorem 4.8 we use Proposition 4.6 to prove the points of Definition 4.5. (The detailed proofs are given in Section 4.5.)

4.2.2 Using Cut Graphs to Enumerate Shapes of Lassos

Proposition 4.1 and Theorem 4.8 suggest that in order to find a schedule that satisfies an $\mathsf{ELTL}_{\mathsf{FT}}$ formula φ, it is sufficient to look for lasso schedules that can be cut in such a way that the configurations at the cut points and the configurations between the cut points satisfy certain propositional formulas. In fact, the cut points as defined by cut functions (Definition 4.4) are *topological orderings* of the cut graph $\mathcal{G}(\mathcal{T}(\varphi))$. Consequently, by enumerating the topological orderings of the cut graph $\mathcal{G}(\mathcal{T}(\varphi))$ we can enumerate the *lasso shapes*, among which there is a lasso schedule satisfying φ (if φ holds on the counter system). These shapes differ in the order, in which $\mathbf{F}$-subformulas of φ are witnessed. For this, one can use fast generation algorithms, e.g., [CW95].

Example 4.6. Consider the cut graph in Figure 4.6. The ordering of its vertices $0, 0.1, 0.2, \mathsf{loop}_{\mathsf{start}}, 0.3.1, \mathsf{loop}_{\mathsf{end}}$ corresponds to the lasso shape (a) shown in Figure 4.2, while the ordering $\mathsf{loop}_{\mathsf{start}}, 0, 0.2, 0.1, \mathsf{loop}_{\mathsf{start}}, 0.3.1, \mathsf{loop}_{\mathsf{end}}$ corresponds to the lasso shape (b). These are the two lasso shapes that one has to analyze, and they are the result of our construction using the cut graph. The other 18 lasso shapes in the figure are not required, and not constructed by our method. ◁

From this observation, we conclude that given a topological ordering $v_1, \dots, v_{|\mathcal{V}_\mathcal{G}|}$ of the cut graph $\mathcal{G}(\mathcal{T}(\varphi)) = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, one has to look for a lasso schedule that can be written as an alternating sequence of configurations σ_i and schedules τ_j:

$$\sigma_0, \tau_0, \sigma_1, \tau_1, \dots, \sigma_\ell, \tau_\ell, \dots, \sigma_{|\mathcal{V}_\mathcal{G}|-1}, \tau_{|\mathcal{V}_\mathcal{G}|}, \sigma_{|\mathcal{V}_\mathcal{G}|}, \tag{4.1}$$

where $v_\ell = \mathsf{loop}_{\mathsf{start}}$, $v_{|\mathcal{V}_\mathcal{G}|} = \mathsf{loop}_{\mathsf{end}}$, and $\sigma_\ell = \sigma_{|\mathcal{V}_\mathcal{G}|}$. Moreover, by Definition 4.5, the sequence of configurations and schedules should satisfy ***(C1)–(C3)***, e.g., if a node v_i corresponds to the formula $\mathbf{F}\,(\psi_0 \wedge \dots \wedge \mathbf{G}\,\psi_{k+1})$ and this formula matches Condition ***(C2)***, then the following should hold:

1. Configuration σ_i satisfies the propositional formula: $\sigma_i \models \psi_0$.

2. All configurations visited by the schedule $\tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}$ from the configuration σ_i satisfy the propositional formula $prop(\psi_{k+1})$. Formally, $\mathsf{Cfgs}(\sigma_i, \tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}) \models prop(\psi_{k+1})$.

Discussion. One can write an SMT query for the sequence (4.1) satisfying Conditions ***(C1)–(C3)***. However, this approach has two problems:

1. The order of rules in schedules $\tau_0, \dots, \tau_{|\mathcal{V}_\mathcal{G}|}$ is not fixed. Non-deterministic choice of rules complicates the SMT query.
2. To guarantee completeness of the search, one requires a bound on the length of schedules $\tau_0, \dots, \tau_{|\mathcal{V}_\mathcal{G}|}$.

For reachability properties these issues were addressed in Chapter 3 by showing that one only has to consider representative schedules, that is, specific orders of the rules. To lift this technique to $\mathsf{ELTL}_{\mathsf{FT}}$, we are left with two issues:

1. The shortening technique applies to steady schedules, i.e., the schedules that do not change evaluation of the guards. Thus, we have to break the schedules $\tau_0, \dots, \tau_{|\mathcal{V}_\mathcal{G}|}$ into steady schedules. This issue is addressed in Section 4.2.3.
2. The shortening technique preserves state reachability, e.g., after shortening of τ_i, the resulting schedule still reaches configuration σ_{i+1}. But it may violate an invariant such as $\mathsf{Cfgs}(\sigma_i, \tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}) \models \mathit{prop}(\psi_{k+1})$. This issue is addressed in Section 4.3.

4.2.3 Cutting Lassos with Threshold Guards

We introduce threshold graphs to cut a lasso into steady schedules, in order to apply the shortening technique of Section 4.1.3. Then, we combine the cut graphs and threshold graphs to cut a lasso into smaller finite segments, which can be first shortened and then checked with the similar approach as the one introduced in Chapter 3.

Given a configuration σ, its context $\omega(\sigma)$ is the set that consists of the rising guards unlocked in σ and the falling guards locked in σ, i.e., $\omega(\sigma) = \Omega^{\mathrm{rise}} \cup \Omega^{\mathrm{fall}}$, where $\Omega^{\mathrm{rise}} = \{g \in \Phi^{\mathrm{rise}} \mid \sigma \models g\}$ and $\Omega^{\mathrm{fall}} = \{g \in \Phi^{\mathrm{fall}} \mid \sigma \not\models g\}$. As discussed previously, e.g., in Example 2.7 on page 25, since the shared variables are never decreased, the contexts in a path are monotonically non-decreasing. This was formalized in Section 3.2.2 as follows:

Proposition 3.2. *If a transition t is enabled in a configuration σ, then either $\omega(\sigma) \sqsubset \omega(t(\sigma))$, or $\omega(\sigma) = \omega(t(\sigma))$.*

Example 4.7. Continuing Example 2.7, which considers the TA in Figure 2.1. Both threshold guards γ_1 and γ_2 are false in the initial state σ. Thus, $\omega(\sigma) = \emptyset$. The transition $t = (r_1, 1)$ unlocks the guard γ_1, i.e., $\omega(t(\sigma)) = \{\gamma_1\}$. ◁

As the transitions of the counter system $\mathsf{Sys}(\mathsf{TA})$ never decrease shared variables, the loop of a lasso schedule must be steady:

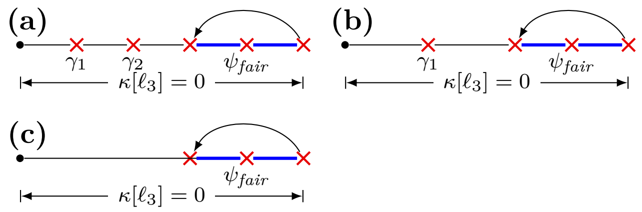


Figure 4.7: The shapes of lassos to check the correctness property in Example 2.8. Recall that γ_1 and γ_2 are the threshold guards, defined as $x \geq t + 1 - f$ and $x \geq n - t - f$ respectively.

Proposition 4.9. *For each configuration σ and a schedule $\tau \cdot \rho^\omega$ applicable to σ, if $\rho^k(\tau(\sigma)) = \tau(\sigma)$ for $k \geq 0$, then the loop ρ is steady for $\tau(\sigma)$, that is, $\omega(\rho(\tau(\sigma))) = \omega(\tau(\sigma))$.*

In Chapter 3, Proposition 3.2 was used to cut a finite path into segments, one per context. We introduce threshold graphs and their topological orderings to apply this idea to lasso schedules.

Definition 4.6. *A* threshold graph *is $\mathcal{H}(\mathsf{TA}) = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ such that:*

- *The vertices set $\mathcal{V}_\mathcal{H}$ contains the threshold guards and the special node $\mathsf{loop}_\mathsf{start}$, i.e., $\mathcal{V}_\mathcal{H} = \Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}} \cup \{\mathsf{loop}_\mathsf{start}\}$.*
- *There is an edge from a guard $g_1 \in \Phi^{\mathrm{rise}}$ to a guard $g_2 \in \Phi^{\mathrm{rise}}$, if g_2 cannot be unlocked before g_1, i.e., $(g_1, g_2) \in \mathcal{E}_\mathcal{H}$, if for each configuration $\sigma \in \Sigma$, $\sigma \models g_2$ implies $\sigma \models g_1$.*
- *There is an edge from a guard $g_1 \in \Phi^{\mathrm{fall}}$ to a guard $g_2 \in \Phi^{\mathrm{fall}}$, if g_2 cannot be locked before g_1, i.e., $(g_1, g_2) \in \mathcal{E}_\mathcal{H}$, if for each configuration $\sigma \in \Sigma$, $\sigma \not\models g_2$ implies $\sigma \not\models g_1$.*

Note that the conditions in Definition 4.6 can be easily checked with an SMT solver, for all configurations.

Example 4.8. The threshold graph of the TA in Figure 2.1 has the vertices $\mathcal{V}_\mathcal{H} = \{\gamma_1, \gamma_2, \mathsf{loop}_\mathsf{start}\}$ and the edges $\mathcal{E}_\mathcal{H} = \{(\gamma_1, \gamma_2)\}$. ◁

Similar to Section 4.2.2, we consider a topological ordering $g_1, \dots, g_\ell, \dots, g_{|\mathcal{V}_\mathcal{H}|}$ of the vertices of the threshold graph. The node $g_\ell = \mathsf{loop}_\mathsf{start}$ indicates the point where a loop should start, and thus by Proposition 4.9, after that point the context does not change. Thus, we consider only the subsequence $g_1, \dots, g_{\ell-1}$ and split the path $\mathsf{path}(\sigma, \tau \cdot \rho)$ of a lasso schedule $\tau \cdot \rho^\omega$ into an alternating sequence of configurations σ_i and schedules τ_0 and $t_j \cdot \tau_j$, for $1 \le j < \ell$, ending up with the loop ρ (starting in $\sigma_{\ell-1}$ and ending in $\sigma_\ell = \sigma_{\ell-1}$):

$$\sigma_0, \tau_0, \sigma_1, (t_1 \cdot \tau_1), \dots, \sigma_{\ell-2}, (t_{\ell-1} \cdot \tau_{\ell-1}), \sigma_{\ell-1}, \rho, \sigma_\ell \tag{4.2}$$

Transitions $t_1, \dots, t_{\ell-1}$ change the context, and schedules $\tau_0, \tau_1, \dots, \tau_{\ell-1}, \rho$ are steady. Finally, we interleave a topological ordering of the vertices of the cut graph with a topological ordering of the vertices of the threshold graph. More precisely, we use a topological ordering of the vertices of the union of the cut graph and the threshold graph. We use the resulting sequence to cut a lasso schedule following the approach in Section 4.2.2 (cf. Equation (4.1)). By enumerating all such interleavings, we obtain all lasso shapes. Again, the lasso is a sequence of steady schedules and context-changing transitions.

Example 4.9. Continuing Example 2.1 given on page 27, we consider the lasso shapes that satisfy the $\mathsf{ELTL_{FT}}$ formula $\mathbf{G}\,\mathbf{F}\,\psi_{\text{fair}} \wedge \boldsymbol{\kappa}[\ell_0] = 0 \wedge \mathbf{G}\,\boldsymbol{\kappa}[\ell_3] = 0$. Figure 4.7 shows the lasso shapes that have to be inspected by an SMT solver. In case (a), both threshold guards γ_1 and γ_2 are eventually changed to true, while the counter $\kappa[\ell_3]$ is never increased in a fair execution. For $n = 3t$, this is actually a counterexample to the correctness property explained in Example 2.1. In cases (b) and (c) at most one threshold guard is eventually changed to true, so these lasso shapes cannot produce a counterexample. $\triangleleft$

In the following section, we will show how to shorten steady schedules, while maintaining Conditions ***(C1)–(C3)*** of Definition 4.5, required to satisfy the $\mathsf{ELTL_{FT}}$ formula.

4.3 Property specific PARA^2 for Safety and Liveness

Let us now formally describe the construction of representative schedules, that was discussed in Section 4.1.3. We start by introducing the necessary definitions and some properties of the introduced notions. As the construction itself depends on the shape of the formula we want to preserve while swapping transitions, in Sections 4.3.1–4.3.3 we address all possible strategies. Technical proofs together with auxiliary lemmas can be found in Section 4.6. We fix a threshold automaton $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$ and conduct our analysis in this section for this TA.

We start by formalizing the notion of a thread.

Definition 4.7 (Thread)**.** *For a configuration σ and a schedule $\tau = \tau_1 \cdot t_1 \cdot \tau_2 \cdot \ldots \cdot t_k \cdot \tau_{k+1}$ applicable to σ, we define the sequence of transitions $\vartheta = t_1, \dots, t_k$, $k > 0$ to be a thread of σ and τ if*

1. *t_i.factor $= 1$, for every $1 \le i \le k$,*
2. *t_i.to $= t_{i+1}$.from, for every $1 \le i < k$.*

For thread ϑ, by $\vartheta.from$ and $\vartheta.to$ we denote $t_1.from$ and $t_k.to$, respectively.

Definition 4.8 (Naming, Projection, and Decomposition)**.** *A* naming *is a function $\eta\colon \mathbb{N} \to \mathbb{N}$. For a schedule τ, and a set $S \subseteq \mathbb{N}$, by $\tau|_{\eta,S}$ we denote the sequence of transitions $\tau[j]$ satisfying $\eta(j) \in S$ that preserves the order of transitions from τ, i.e., for all j_1, j_2, l_1, l_2, $j_1 < j_2$, if $\tau|_{\eta,S}[l_1] = \tau[j_1]$ and $\tau|_{\eta,S}[l_2] = \tau[j_2]$, then $l_1 < l_2$. If S is a one-element set $\{i\}$, we write $\tau|_{\eta,i}$ instead of $\tau|_{\eta,\{i\}}$. We use the notation $\Theta(\sigma, \tau, \eta)$ for the set $\{i\colon \tau|_{\eta,i}$ is a thread of σ and $\tau\}$. For a configuration σ and a schedule τ, a naming η is called a* decomposition *of σ and τ if*

1. *for all $i \in \mathbb{N}$, the schedule $\tau|_{\eta,i}$ is either a thread of σ and τ, or the empty sequence,*
2. *for every local state $\ell \in \mathcal{L}$, the value of its counter in σ is greater than or equal to the number of threads starting in that state, that is,*

$$\sigma.\boldsymbol{\kappa}[\ell] \geq |\{i\colon i \in \Theta(\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.from = \ell\}|.$$

Example 4.10. Let us reconsider Example 4.3 and Figure 4.4, with configuration σ_1 where $\sigma_1.\boldsymbol{\kappa}[\ell_0] = \sigma_1.\boldsymbol{\kappa}[\ell_1] = \sigma_1.\boldsymbol{\kappa}[\ell_2] = 1$ and $\sigma_1.\boldsymbol{\kappa}[\ell_3] = 0$, and the schedule $\tau = (r_1, 1)$, $(r_6, 1)$, $(r_4, 1)$, $(r_2, 1)$, $(r_4, 1)$. The function $\eta\colon \mathbb{N} \to \mathbb{N}$ with $\eta(1) = \eta(5) = 1$, $\eta(2) = \eta(4) = 2$, $\eta(3) = 3$, and $\eta(k) = 4$ for every $k \geq 6$, is a naming. We will now see that η is a decomposition by checking the two points.

(1) Since $\eta(1) = \eta(5) = 1$, the projection $\tau|_{\eta,1}$ consists of the first and the fifth transition, in that particular order, i.e., $\tau|_{\eta,1} = (r_1, 1), (r_4, 1)$. This is a thread, as the factor of both transitions is 1, and $r_1.to = \ell_2 = r_4.from$. Similarly, $\tau|_{\eta,2} = (r_6, 1), (r_2, 1)$, and $\tau|_{\eta,3} = (r_4, 1)$ are threads. Besides, $\tau|_{\eta,4}$ is the empty sequence, as numbers mapping to 4 are $n \geq 6$, and τ has length 5, i.e., there is no transition $\tau[n]$ for $n \geq 6$. Further, for every $i > 4$, $\tau|_{\eta,i}$ is the empty sequence, as there is no $m \in \mathbb{N}$, with $\eta(m) = i$. Thus, $\Theta(\sigma, \tau, \eta) = \{1, 2, 3\}$. Note that $\tau|_{\eta,\mathbb{N}\setminus\{2\}} = \tau|_{\eta,\{1,3\}} = (r_1, 1), (r_4, 1), (r_4, 1)$.

(2) As $\tau|_{\eta,2} = r_6.from = \ell_0$, we obtain $\{i\colon i \in \Theta(\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.from = \ell_0\} = \{2\}$, and $\sigma_1.\boldsymbol{\kappa}[\ell_0] = 1 \geq |\{2\}|$. Similarly we check this inequality for other local states, and conclude that η is a decomposition of σ_1 and τ. ◁

Note that a prefix of a schedule has at least as many threads as the original schedule.

Proposition 4.10. *If σ is a configuration, τ is a steady schedule applicable to σ, and η is a decomposition of σ and τ, then for each prefix τ' of τ, the naming η is a decomposition of σ and τ'. Further $\Theta(\sigma, \tau', \eta) \subseteq \Theta(\sigma, \tau, \eta)$.*

From [KVW17, Prop. 12] we directly obtain:

Proposition 4.11. *If σ is a configuration, τ is a steady schedule applicable to σ, and η is a decomposition of σ and τ, then for all ℓ in $\mathcal{L}$ the following holds:*

$$\tau(\sigma).\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell] + |\{i\colon i \in \Theta(\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.to = \ell\}| - |\{i\colon i \in \Theta(\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.from = \ell\}|.$$

The following proposition says that every steady conventional schedule can be decomposed into threads. The detailed proof can be found in Section 4.6

Proposition 4.12. *If σ is a configuration, τ is a steady conventional schedule applicable to σ, then there exists a decomposition of σ and τ.*

The following lemma will be useful for proving that a transition t_i of a thread is applicable to a configuration, as it assures that there is a process in $t_i.from$. Its proof can be found in Section 4.6.

Proposition 4.13. *If σ is a configuration, $\tau = \tau_1 \cdot t_{i-1} \cdot t_i \cdot \tau_2$ is a steady schedule applicable to σ, η is a decomposition of σ and τ, and $\eta(i-1) \neq \eta(i)$, then $\tau_1(\sigma).\boldsymbol{\kappa}[t_i.from] \geq 1$.*

As in a steady schedule guards do not change, and thus do not play a role, applicability depends only on the process that can execute the transition. Therefore, Proposition 4.13 implies that a transition of a thread commutes with the adjacent transition of a different thread. This will allow us to move a whole thread.

Definition 4.9 (Move). *For a schedule τ, and a natural number i, $1 < i \leq |\tau|$, the schedule $\tau_{i\leftarrow}$ is obtained by moving the ith transition of τ to the left, and naming $\eta_{i\leftarrow}(k)$ is defined accordingly, for every $k \in \mathbb{N}$, i.e.,*

$$\tau_{i\leftarrow}[k] = \begin{cases} \tau[i] & \text{if } k = i-1 \\ \tau[i-1] & \text{if } k = i \\ \tau[k] & \text{otherwise,} \end{cases} \qquad \text{and} \qquad \eta_{i\leftarrow}(k) = \begin{cases} \eta(i) & \text{if } k = i-1 \\ \eta(i-1) & \text{if } k = i \\ \eta(k) & \text{otherwise.} \end{cases}$$

For natural numbers n and m, where $1 \leq n \leq m \leq |\tau|$, we define $\tau_{n\leftarrow m}$ to be the schedule obtained from τ by moving the mth transition of τ to the nth position (that is $m-n$ times to the left), and naming $\eta_{n\leftarrow m}(k)$ accordingly, for every $k \in \mathbb{N}$, i.e.,

$$\tau_{n\leftarrow m} = (\dots((\tau_{m\leftarrow})_{m-1\leftarrow})\dots)_{n+1\leftarrow} \quad \text{and}$$

$$\eta_{n\leftarrow m}(k) = (\dots((\eta_{m\leftarrow})_{m-1\leftarrow})\dots)_{n+1\leftarrow}(k).$$

Example 4.11. Note that if we have $m = n$, then it holds that $\tau_{n\leftarrow m} = \tau$ and $\eta_{n\leftarrow m} = \eta$. If $\tau = t_1, t_2, \dots, t_{|\tau|}$, then for $i, n, m \in \mathbb{N}$ with $n < m \leq |\tau|$, and $i \leq |\tau|$, it is

$$\tau_{i\leftarrow} = t_1, \dots, t_{i-2}, t_i, t_{i-1}, t_{i+1}, \dots, t_{|\tau|}, \text{ and}$$

$$\tau_{n\leftarrow m} = t_1, \dots, t_{n-1}, t_m, t_n, t_{n+1}, \dots, t_{m-1}, t_{m+1}, \dots, t_{|\tau|}.$$

◁

Finally, we show that swapping a transition of a thread with a transition of a different thread, gives us a schedule that reaches the same state as the original schedule. We leave the formal proof for Section 4.6.

Proposition 4.14. *Let σ be a configuration, let τ be a steady schedule applicable to σ, and let η be a decomposition of σ and τ. For every $i \in \mathbb{N}$, if $1 < i \leq |\tau|$ and $\eta(i-1) \neq \eta(i)$, then the following holds:*

1. *$\tau_{i\leftarrow}$ is a steady schedule applicable to σ,*
2. *$\eta_{i\leftarrow}$ is a decomposition of σ and $\tau_{i\leftarrow}$, and $\tau_{i\leftarrow}|_{\eta_{i\leftarrow},j} = \tau|_{\eta,j}$, for every $j \in \Theta(\sigma, \tau, \eta)$,*
3. *$\tau_{i\leftarrow}(\sigma) = \tau(\sigma)$.*

Consequently, the same holds if we apply the previous proposition multiple times.

Proposition 4.15. *Let σ be a configuration, let τ be a steady schedule applicable to σ, and let η be a decomposition of σ and τ. If for $n, m \in \mathbb{N}$ holds that $1 \leq n \leq m \leq |\tau|$ and $\eta(m) \neq \eta(i)$, for every i with $n \leq i < m$, then*

1. *$\tau_{n\leftarrow m}$ is a steady schedule applicable to σ,*
2. *$\eta_{n\leftarrow m}$ is a decomposition of σ and $\tau_{n\leftarrow m}$, and $\tau_{n\leftarrow m}|_{\eta_{n\leftarrow m},j} = \tau|_{\eta,j}$, for every $j \in \Theta(\sigma, \tau, \eta)$,*
3. *$\tau_{n\leftarrow m}(\sigma) = \tau(\sigma)$.*

Proof. This statement is a consequence of Proposition 4.14 applied inductively $m - n$ times, as Definition 4.9 suggests. In the case when $m = n$, the statement is trivially satisfied. □

The previous reasoning helps us to move one whole thread in a certain way.

Proposition 4.16. *Let σ be a configuration, let τ be a steady schedule applicable to σ, and let η be a decomposition of σ and τ. Fix an $i \in \Theta(\sigma, \tau, \eta)$. Let us denote*

$$\tau^* = \tau' \cdot \tau|_{\eta,i} \cdot \tau'',$$

such that τ' is a possibly empty prefix of τ which contains no transitions from $\tau|_{\eta,i}$, and $\tau' \cdot \tau'' = \tau|_{\eta,\mathbb{N}\setminus\{i\}}$. Then we have the following:

1. *τ^* is a steady schedule applicable to σ,*
2. *there exists a decomposition η^* of σ and τ^* such that $\tau^*|_{\eta^*,l} = \tau|_{\eta,l}$, for every $l \in \Theta(\sigma, \tau, \eta)$.*
3. *$\tau^*(\sigma) = \tau(\sigma)$.*

Proof Sketch: We enumerate all transitions from $\tau|_{\eta,i}$, for example, $\tau|_{\eta,i} = t_{n_1}, t_{n_2}, \dots, t_{n_k}$, for $1 \le n_1 < n_2 < \dots < n_k \le |\tau|$. Thus, $\tau' = t_1, \dots, t_s$, for $0 \le s < n_1$. The idea is that we move transitions from $\tau|_{\eta,i}$, one by one, to the left, namely t_{n_1} to the place $(s+1)$ in τ, then t_{n_2} to the place $s+2$, and so on, by repeatedly applying Proposition 4.14, that preserves the required properties. Formally, $\tau^* = (\dots((\tau_{s+1\leftarrow n_1})_{s+2\leftarrow n_2})\dots)_{s+k\leftarrow n_k}$.

Inductively, we prove in details in Section 4.6 that all three requirements hold. □

Special cases of Proposition 4.16 allow us to single out threads. Proposition 4.17 and Proposition 4.18 are such special cases, and they will be crucial for finding witnessing threads later in Section 4.3.1 and Section 4.3.2, respectively.

Proposition 4.17. *Let σ be a configuration, let τ be a steady schedule applicable to σ, and let η be a decomposition of σ and τ. If $i \in \Theta(\sigma,\tau,\eta)$, and we denote*

$$\tau^* = \tau|_{\eta,i} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i\}},$$

then the following holds:

1. *τ^* is a steady schedule applicable to σ,*
2. *there exists a decomposition η^* of σ and τ^* such that $\tau^*|_{\eta^*,l} = \tau|_{\eta,l}$, for every $l \in \Theta(\sigma,\tau,\eta)$,*
3. *$\tau^*(\sigma) = \tau(\sigma)$.*

Proof. This Proposition is a special case of Proposition 4.16, when τ' is the empty schedule. □

Proposition 4.18. *Let σ be a configuration, let τ be a steady schedule applicable to σ, and let η be a decomposition of σ and τ. Fix $i,j \in \Theta(\sigma,\tau,\eta)$. If $\tau|_{\eta,j}$ can be written as $\tau|^1_{\eta,j} \cdot \tau|^2_{\eta,j}$, for some schedules $\tau|^1_{\eta,j}$ and $\tau|^2_{\eta,j}$, and if we denote*

$$\tau^* = \tau|^1_{\eta,j} \cdot \tau|_{\eta,i} \cdot \tau|^2_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i,j\}},$$

then the following holds:

1. *τ^* is a steady schedule applicable to σ,*
2. *there exists a decomposition η^* of σ and τ^* such that $\tau^*|_{\eta^*,i} = \tau|_{\eta,i}$ and $\tau^*|_{\eta^*,j} = \tau|_{\eta,j}$, and*
3. *$\tau^*(\sigma) = \tau(\sigma)$.*

Proof. Firstly, we apply Proposition 4.17 for configuration σ, schedule τ, decomposition η and $j \in \Theta(\sigma, \tau, \eta)$. Then we obtain schedule $\rho = \tau|_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{j\}}$ and decomposition η_ρ of σ and ρ. Then we apply Proposition 4.16 for configuration σ, schedule ρ, decomposition η_ρ, $i \in \Theta(\sigma, \tau, \eta)$, and a prefix $\tau|^1_{\eta,j}$ of ρ (as τ' from the proposition). $\square$

Until now, we discussed when transitions can be moved. In Chapter 3, the goal of this moving is to transform a schedule into a *representative schedule* that reaches the same final configuration (cf. Example 4.2). These representative schedules are highly accelerated, and their length can be bounded. We recall how these representative schedules are constructed in Chapter 3, but with our restrictions on loops, this procedure becomes simplified. Next, Proposition 4.21 establishes that representatives maintain an important trace property.

Constructing Representative Schedules. Given a configuration σ, and a steady schedule τ applicable to σ, $\mathsf{srep}[\sigma, \tau]$ is generated from τ by repeatedly swapping two neighboring transitions t_1 and t_2 if $[t_2] \prec^{lin}_{C} [t_1]$ until no more such transitions exist. Then all neighboring transitions that belong to the same rule are merged into a single (possibly accelerated) transition.

Then the transitions belonging to loops are replaced by a quite involved construction from Section 3.4. As discussed in Section 2.1, in this chapter we consider the restriction that loops are simple. Hence, we can have a simplified construction: Assume for some j, the rules $r_1, r_2, \ldots, r_j$ build a loop, and assume τ_{loop} is a subschedule of τ that contains all transitions of this particular loop in τ. Let σ_0 and σ_{end} be two configurations defined such that $\sigma_{\text{end}} = \tau_{\text{loop}}(\sigma_0)$. In order to construct a representative of τ, we need to replace τ_{loop} using the following steps:

1. first we construct $\tau' = (r_1, f_1), (r_2, f_2), \ldots, (r_j, f_j), (r_1, f_{j+1}), (r_2, f_{j+2}), \ldots, (r_j, f_{2j})$ with the acceleration factors obtained as follows:
   - If $r_1, \ldots, r_j$ appear in τ_{loop}: inductively assigning values to the acceleration factors f_i, for $1 \le i \le 2j$ as follows:
     - for $1 \le i \le j$:
       $f_i = \sigma_{i-1}.\boldsymbol{\kappa}[r_i.\mathit{from}] - \min(\sigma_0.\boldsymbol{\kappa}[r_i.\mathit{from}], \sigma_{\text{end}}.\boldsymbol{\kappa}[r_i.\mathit{from}])$
       and for $t_i = (r_i, f_i)$, we get $\sigma_i = t_i(\sigma_{i-1})$
     - for $j + 1 \le i \le 2j$:
       $f_i = \sigma_{i-1}.\boldsymbol{\kappa}[r_{i-j}.\mathit{from}] - \sigma_{\text{end}}.\boldsymbol{\kappa}[r_{i-j}.\mathit{from}]$ and
       for $t_i = (r_i, f_i)$, we obtain $\sigma_i = t_i(\sigma_{i-1})$
   - otherwise, that is, if some rules are missing in the schedule, then we set their acceleration factors to zero. Note that due to the missing rules, the loop falls apart into several independent chains. Each of this chains is a subschedule of τ', we just have to sum up the acceleration factors for the present rules. Formally, we proceed in two steps: First, if r_i is not present in τ_{loop}, and for all

$k < i$, r_k is present in τ_{loop} then $f_\ell = 0$, for all l that satisfy $l \leq i$ or $l \geq j + i$. Second, for l with $i < l \leq j$, the factor f_l is the sum of the acceleration factors of transitions in τ_{loop} with the rule r_l. For l with $j < l < i + j$, f_l is the sum of the acceleration factors of transitions in τ_{loop} with the rule r_{l-j}.

2. τ'' is obtained from τ' by removing all transitions with zero acceleration factors

3. we replace τ_{loop} with τ''.

By this construction, every location that is non-empty at the beginning and at the end of τ_{loop}, must also be non-empty in every visited configuration of its representative.

Proposition 4.19. *Let τ_{loop} be a schedule applicable to σ_0 that consists of transitions whose rules all belong to the same loop. For all $\ell \in \mathcal{L}$, if $\sigma_0.\boldsymbol{\kappa}[\ell] > 0$ and $\tau_{\text{loop}}(\sigma_0).\boldsymbol{\kappa}[\ell] > 0$, then $\mathsf{Cfgs}(\sigma_0, \mathsf{srep}[\sigma_0, \tau_{\text{loop}}]) \models \boldsymbol{\kappa}[\ell] > 0$.*

In this way, in contrast to the representatives from Section 3.4, here $\mathsf{srep}[\sigma, \tau]$ contains a subset of the rules of τ but ordered according to the linear extension $\prec_C^{lin}$ of the control flow of the automaton. Thus, from the above construction we directly obtain:

Proposition 4.20. *Let σ be a configuration and let τ be a steady schedule applicable to σ. The rules contained in transitions of $\mathsf{srep}[\sigma, \tau]$ are a subset of the rules contained in transitions of τ.*

From Proposition 4.2, we know that we can replace a schedule by its representative, and maintain the same final state. In the following propositions, we show that the representative schedule also maintains non-zero counters.

Proposition 4.21. *Let σ be a configuration, and let τ be a steady schedule applicable to σ. For every $\ell \in \mathcal{L}$, it holds that $\mathsf{Cfgs}(\sigma, \tau) \models \boldsymbol{\kappa}[\ell] > 0$ implies $\mathsf{Cfgs}(\sigma, \mathsf{srep}[\sigma, \tau]) \models \boldsymbol{\kappa}[\ell] > 0$.*

Proof. Schedule $\mathsf{srep}[\sigma, \tau]$ is constructed by first swapping transitions and then reducing loops. We first show that swapping maintains $\boldsymbol{\kappa}[\ell] > 0$, and then that reducing loops does so, too.

Consider the sub-path $\sigma_{i-1}, t_i, \sigma_i, t_{i+1}, \sigma_{i+1}$ of one schedule in the construction and $\sigma_{i-1}, t_{i+1}, \sigma_i', t_i, \sigma_{i+1}$ be the path obtained by swapping. Assume by ways of contradiction that $\sigma_{i-1}.\boldsymbol{\kappa}[\ell] > 0$, $\sigma_i.\boldsymbol{\kappa}[\ell] > 0$, and $\sigma_{i+1}.\boldsymbol{\kappa}[\ell] > 0$, but $\sigma_i'.\boldsymbol{\kappa}[\ell] = 0$. As $\boldsymbol{\kappa}[\ell]$ reduces from σ_{i-1} to σ_i', we get $t_{i+1}.\mathit{from} = \ell$. By similar reasoning on σ_i' and σ_{i+1} we obtain $t_i.\mathit{to} = \ell$. It thus holds that $t_i \prec_P t_{i+1}$, which contradicts that these transitions are swapped in the construction of $\mathsf{srep}[\sigma, \tau]$.

Let $\ldots, \sigma, \tau, \sigma', \ldots$ be the path before the loops are replaced and τ consist of all the transition belonging to one loop. From the above paragraph we know that $\sigma.\boldsymbol{\kappa}[\ell] > 0$ and $\sigma'.\boldsymbol{\kappa}[\ell] > 0$. We may thus apply Proposition 4.19 and the proposition follows. □

Discussion. For threshold-guarded fault-tolerant algorithms, the restrictions we put on threshold automata are well justified. In this chapter we used the assumption that all the cycles in threshold automata are simple. In fact this assumption is a generalization of the TAs we found in our benchmarks. In Chapter 3 we do not make this assumption. As a consequence, we have to explicitly treat contexts (the guards that currently evaluate to true), which lead to context-specific representative schedules. Our restriction allows us to use only one way to construct simple representative schedules (cf. Section 4.3). In addition, with this restriction, we can easily prove Proposition 4.20, while this proposition is not true under the assumptions in Chapter 3, since there we often introduce new transitions. We conjecture that even without this restriction, a proposition similar to our Proposition 4.21 can be proven, so that our results can be extended. As our analysis already is quite involved, these restrictions allow us to concentrate on our central results without obfuscating the notation and theoretical results. Still, from a theoretical viewpoint it might be interesting to lift the restrictions on loops and obtain results similar to canonical automata in Chapter 3. Note that more general forms of loops were investigated in [KKW18] for reachability properties.

We have now seen how to construct simple representative schedules. In the following Sections 4.3.1 to 4.3.3, we will see how we can construct representative schedules that maintain different forms of temporal properties.

4.3.1 Representative Schedules that maintain $\bigvee_{i\in Locs} \boldsymbol{\kappa}[i] \neq 0$

Maintaining formulas of the shape $\bigvee_{i\in Locs} \boldsymbol{\kappa}[i] \neq 0$ means preserving the existence of a process in one of the critical states, that is a state from *Locs*. Our strategy for constructing representatives in this case is quite complex, as it requires finding a witness for the non-emptiness. This witness depends on the features of the existing threads in a schedule. Here we define all possible types of threads in a schedule, and illustrate them intuitively in Figure 4.8.

Definition 4.10 (Thread Types). *Let σ be a configuration, τ be a schedule applicable to σ, $\vartheta = t_1, \ldots, t_n$ be a thread of σ and τ, $\mathrm{first}(\vartheta) = t_1.from$, $\mathrm{last}(\vartheta) = t_n.to$, $\mathrm{middle}(\vartheta) = \{t_i.to \colon 1 \le i < n\}$, and $Locs \subseteq \mathcal{L}$. We say that ϑ is of Locs-type:*

- *A, if* $\{\mathrm{first}(\vartheta), \mathrm{last}(\vartheta)\} \cup \mathrm{middle}(\vartheta) \subseteq Locs$*;*
- *B, if* $\mathrm{first}(\vartheta) \in Locs$*,* $\mathrm{last}(\vartheta) \notin Locs$*;*
- *C, if* $\mathrm{first}(\vartheta) \notin Locs$*,* $\mathrm{last}(\vartheta) \in Locs$*;*
- *D, if* $\mathrm{first}(\vartheta) \notin Locs$*,* $\mathrm{last}(\vartheta) \notin Locs$*,* $\mathrm{middle}(\vartheta) \cap Locs \neq \emptyset$*;*
- *E, if* $\mathrm{first}(\vartheta) \in Locs$*,* $\mathrm{last}(\vartheta) \in Locs$*,* $\mathrm{middle}(\vartheta) \not\subseteq Locs$*;*
- *F, if* $(\{\mathrm{first}(\vartheta), \mathrm{last}(\vartheta)\} \cup \mathrm{middle}(\vartheta)) \cap Locs = \emptyset$*.*

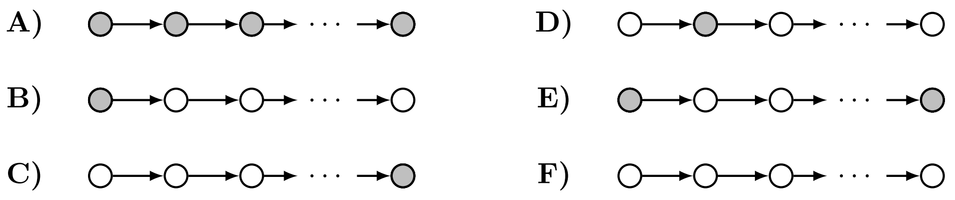


Figure 4.8: We graphically illustrate all six *Locs*-types of threads. Nodes represent (not necessarily different) local states visited by the thread, and the gray nodes are locations that belong to *Locs*. The leftmost location of each thread ϑ is first(ϑ), and the rightmost one is last(ϑ).

Example 4.12. Let us consider the threshold automaton from Figure 4.4, and the subset of local states $Locs = \{\ell_2\}$. Schedule $(r_4, 1)$ is of *Locs*-type B, schedule $(r_6, 1), (r_2, 1)$ is of *Locs*-type C, and $(r_1, 1), (r_4, 1)$ is of *Locs*-type D. $\triangleleft$

Proposition 4.22. *Given a configuration σ, a schedule τ applicable to σ, and a subset of local states Locs, every thread ϑ of σ and τ is of exactly one Locs-type.*

Proof. We consider an arbitrary thread ϑ of σ and τ. There are two possibilities for first(ϑ), namely, first(ϑ) $\in$ *Locs* or first(ϑ) $\notin$ *Locs*, and similarly for last(ϑ), last(ϑ) $\in$ *Locs* or last(ϑ) $\notin$ *Locs*. Combining these possibilities, we obtain four cases:

- Assume first(ϑ) $\in$ *Locs* and last(ϑ) $\in$ *Locs*. If middle(ϑ) $\subseteq$ *Locs*, then ϑ is of *Locs*-type A. Otherwise, if middle(ϑ) $\nsubseteq$ *Locs*, then ϑ is of *Locs*-type E.
- If first(ϑ) $\in$ *Locs* and last(ϑ) $\notin$ *Locs*, then ϑ is of *Locs*-type B.
- If first(ϑ) $\notin$ *Locs* and last(ϑ) $\in$ *Locs*, then ϑ is of *Locs*-type C.
- Finally, assume first(ϑ) $\notin$ *Locs* and last(ϑ) $\notin$ *Locs*. If middle(ϑ) $\cap$ *Locs* $\neq \emptyset$, then ϑ is of *Locs*-type D. Otherwise, if middle(ϑ) $\cap$ *Locs* $= \emptyset$, then ϑ is of *Locs*-type F.

□

In the following proposition we see the benefit of our simplified construction of representatives (in comparison to Chapter 3), allowed by the loop restriction. Namely, we show that the representative of a thread of *Locs*-type A is still a thread of *Locs*-type A.

Proposition 4.23. *Let σ be a configuration, and let τ be a steady conventional schedule applicable to σ. If there exists a decomposition η of σ and τ that satisfies $|\Theta(\sigma, \tau, \eta)| = 1$, and if τ is of Locs-type A, then* $\mathsf{srep}[\sigma, \tau]$ *is a thread of Locs-type A.*

Proof. By definition of a thread, the transitions in τ are ordered by the flow relation $\prec_P$. Due to our restriction of loops and the construction of representative schedules, $\mathsf{srep}[\sigma, \tau]$ does not contain rules that are not contained in τ. Hence, no new intermediate states are added in the construction of $\mathsf{srep}[\sigma, \tau]$ which proves the proposition. □

Moreover, a process that moves along a thread of *Locs*-type A is a witness that there is always a process in the critical section *Locs*.

Proposition 4.24. *Let σ be a configuration, and let τ be a steady conventional schedule applicable to σ. Fix a set Locs $\subseteq \mathcal{L}$. If there exists a decomposition η of σ and τ that satisfies $|\Theta(\sigma, \tau, \eta)| = 1$ and τ is of Locs-type A, then we have that*

$$\mathsf{Cfgs}(\sigma, \mathsf{srep}[\sigma, \tau]) \models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0.$$

Proof. Let $\mathsf{srep}[\sigma, \tau] = t_1, \ldots, t_n$, for an $n \in \mathbb{N}$. Since τ is of *Locs*-type A, by Proposition 4.23, $\mathsf{srep}[\sigma, \tau]$ is of *Locs*-type A, which yields that for all $1 \leq i \leq n$ both $t_i.\mathit{from}$ and $t_i.\mathit{to}$ are in *Locs*.

- Since $\mathsf{srep}[\sigma, \tau]$ is applicable to σ, it must be the case that $\sigma \models \boldsymbol{\kappa}[\ell^*] \neq 0$, where $\ell^* = t_1.\mathit{from} \in Locs$.
- If $\tau' = t_1, \ldots, t_k$, $1 \leq k \leq n$, is a nonempty prefix of $\mathsf{srep}[\sigma, \tau]$, then, by definition of a counter system from Section 2.2, we have that $\tau'(\sigma).\boldsymbol{\kappa}[t_k.\mathit{to}] > 0$, and also $t_k.\mathit{to} \in Locs$.

Therefore, $\mathsf{Cfgs}(\sigma, \mathsf{srep}[\sigma, \tau]) \models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$. □

If a decomposition gives no threads of *Locs*-type A, then finding a witness for non-emptiness of *Locs* is nontrivial. The following lemma suggests a new direction if the last location of a thread is in *Locs*. It claims that after executing a thread in a path, its last state remains non-empty.

Lemma 4.25. *Let σ be a configuration, let τ be a steady conventional schedule applicable to σ, and let η be a decomposition of σ and τ. If $k \in \Theta(\sigma, \tau, \eta)$ and $n \in \mathbb{N}$ are such that t_n is the last transition from $\tau|_{\eta,k}$, that is, n is the maximal number with $\eta(n) = k$, then for every prefix τ' of τ, of length $|\tau'| \geq n$, we have that $\tau'(\sigma).\boldsymbol{\kappa}[\ell] \neq 0$, for $\ell = \mathrm{last}(\tau|_{\eta,k})$.*

Proof. Fix a prefix τ' of τ of length at least n. Then, by Proposition 4.10, η is a decomposition of σ and τ'. Note that $k \in \Theta(\sigma, \tau', \eta)$, and $\tau|_{\eta,k} = \tau'|_{\eta,k}$. Therefore $\tau'|_{\eta,k}.\mathit{to} = \tau|_{\eta,k}.\mathit{to} = t_n.\mathit{to}$. Proposition 4.11, when applied to τ', yields

$$\begin{aligned}\tau'(\sigma).\boldsymbol{\kappa}[t_n.\mathit{to}] = \sigma.\boldsymbol{\kappa}[t_n.\mathit{to}]& \\ &+ |\{i \colon i \in \Theta(\sigma, \tau', \eta) \wedge \tau'|_{\eta,i}.\mathit{to} = t_n.\mathit{to}\}| \\ &\qquad - |\{i \colon i \in \Theta(\sigma, \tau', \eta) \wedge \tau'|_{\eta,i}.\mathit{from} = t_n.\mathit{to}\}|\end{aligned}$$

By Definition 4.8, we have that $\sigma.\boldsymbol{\kappa}[t_n.to] - |\{i\colon i \in \Theta(\sigma,\tau',\eta) \wedge \tau'|_{\eta,i}.from = t_n.to\}| \geq 0$. Since $\eta(n) = k \in \{i\colon i \in \Theta(\sigma,\tau',\eta) \wedge \tau'|_{\eta,i}.to = t_n.to\}$, we conclude that $|\{i\colon i \in \Theta(\sigma,\tau',\eta) \wedge \tau'|_{\eta,i}.to = t_n.to\}| \geq 1$. Thus $\tau'(\sigma).\boldsymbol{\kappa}[t_n.to] \geq 1$. □

Previous lemma yields a witness after a thread has been executed. The following lemma tells us what happens before a thread is executed. Namely, the first location of a thread is non-empty in every configuration before executing the thread. This gives us a witness when the first state is in *Locs*.

Lemma 4.26. *Let σ be a configuration, let τ be a steady conventional schedule applicable to σ, and let η be a decomposition of σ and τ. If $k \in \Theta(\sigma,\tau,\eta)$ and $n \in \mathbb{N}$ are such that t_n is the first transition from $\tau|_{\eta,k}$, i.e., n is the minimal number with $\eta(n) = k$, then for every prefix τ' of τ, of length $|\tau'| < n$, we have that $\tau'(\sigma).\boldsymbol{\kappa}[\ell] \neq 0$, for $\ell = \mathrm{first}(\tau|_{\eta,k})$.*

Proof. By repeated application of Proposition 4.14, the first transition of $\tau|_{\eta,k}$ can be moved to the beginning of the schedule. Applying Proposition 4.13 to the resulting schedule proves this lemma. □

In order to be sure that $\mathsf{Cfgs}(\sigma,\tau) \models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$, we need a witness for the whole path, and not only before or after a certain thread. Thus, we have to combine threads. We prove that this is possible even in the non-trivial case, that is, when there is no a single local state $\ell \in Locs$ that is non-empty in every configuration from $\mathsf{Cfgs}(\sigma,\tau)$. First we show which threads we are guaranteed to have in this case. The detailed proof can be found in Section 4.6.1.

Proposition 4.27. *Let σ be a configuration, let $\tau = t_1,\dots,t_{|\tau|}$ be a nonempty steady conventional schedule applicable to σ, and let η be a decomposition of σ and τ. Fix a set Locs of local states. If there is no local state $\ell \in Locs$ such that $\mathsf{Cfgs}(\sigma,\tau) \models \boldsymbol{\kappa}[\ell] \neq 0$, but it holds that $\mathsf{Cfgs}(\sigma,\tau) \models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$, then at least one of the following cases is true:*

1. *There is at least one thread of σ and τ, which is of Locs-type A;*
2. *There is a thread of Locs-type B or E, and an additional of Locs-type C or E;*
3. *There is a thread of Locs-type E, and one of Locs-type D.*

Applying thread types. Now when we know which threads we can have, we consider them one by one. Proposition 4.17 and Proposition 4.18 come to play here, as they allow us to move threads in a convenient way.

If there is a thread of *Locs*-type A and if we move it to the beginning of its schedule, then by Lemma 4.25 even after executing it, there is a witness that *Locs* is non-empty.

Proposition 4.28. *Let σ be a configuration, let τ be a steady conventional schedule applicable to σ, and let η be a decomposition of σ and τ. Fix a set Locs of local states, and an $i \in \Theta(\sigma, \tau, \eta)$. If $\tau|_{\eta,i}$ is a thread of Locs-type A, and if we denote $\ell^* = \mathrm{last}(\tau|_{\eta,i})$, then $\ell^* \in Locs$, and*

$$\mathit{Cfgs}(\tau|_{\eta,i}(\sigma), \tau|_{\eta,\mathbb{N}\setminus\{i\}}) \models \boldsymbol{\kappa}[\ell^*] \neq 0.$$

Proof. Firstly note that $\tau|_{\eta,i} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i\}}$ is a steady schedule applicable to σ, and $\tau|_{\eta,i} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i\}}(\sigma) = \tau(\sigma)$, by Proposition 4.17. Let τ' be a prefix of $\tau|_{\eta,\mathbb{N}\setminus\{i\}}$. Then $\tau|_{\eta,i} \cdot \tau'$ is a prefix of $\tau|_{\eta,i} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i\}}$ of length $l \geq |\tau|_{\eta,i}|$. By Lemma 4.25, it is $\tau|_{\eta,i} \cdot \tau'(\sigma).\boldsymbol{\kappa}[\ell^*] \neq 0$, where $\ell^* = \mathrm{last}(\tau|_{\eta,i})$. As $\tau|_{\eta,i}$ is of *Locs*-type A, then $\ell^* \in Locs$. □

Next we look for the witnesses of non-emptiness of *Locs* when we are given one thread of *Locs*-type either B or E, and an additional one of *Locs*-type either C or E.

Proposition 4.29. *Let σ be a configuration, let τ be a steady conventional schedule applicable to σ, let η be a decomposition of σ and τ, and let Locs be a subset of $\mathcal{L}$. If $i, j \in \Theta(\sigma, \tau, \eta)$ are such that $i \neq j$, $\tau|_{\eta,i}$ is a thread of Locs-type B or E, and $\tau|_{\eta,j}$ is a thread of Locs-type C or E, then it holds that*

1) *$\mathit{Cfgs}(\sigma, \tau|_{\eta,j}) \models \boldsymbol{\kappa}[\ell_1] \neq 0$, for $\ell_1 = \mathrm{first}(\tau|_{\eta,i}) \in Locs$,*

2) *$\mathit{Cfgs}(\tau|_{\eta,j}(\sigma), \tau|_{\eta,\mathbb{N}\setminus\{j\}}) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, for $\ell_2 = \mathrm{last}(\tau|_{\eta,j}) \in Locs$.*

Proof. Firstly note that $\tau|_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{j\}}$ is a steady schedule applicable to σ, and $\tau|_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{j\}}(\sigma) = \tau(\sigma)$, by Proposition 4.17.

1) Let τ' be a prefix of $\tau|_{\eta,j}$. Note that in this case τ' is a prefix of $\tau|_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{j\}}$ of length $l \leq |\tau|_{\eta,j}|$. From Lemma 4.26 we obtain $\tau'(\sigma) \models \boldsymbol{\kappa}[\ell_1] \neq 0$, where $\ell_1 = \mathrm{first}(\tau|_{\eta,i})$. Since $\tau|_{\eta,i}$ is of type B or E, we have that $\ell_1 \in Locs$.

2) Let τ' be a prefix of $\tau|_{\eta,\mathbb{N}\setminus\{j\}}$. In this case, $\tau|_{\eta,j} \cdot \tau'$ is a prefix of $\tau|_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{j\}}$ of length $l \geq |\tau|_{\eta,j}|$. By Lemma 4.25 we have that $\tau|_{\eta,j} \cdot \tau'(\sigma) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, or, equivalently, $\tau'(\tau|_{\eta,j}(\sigma)) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, where $\ell_2 = \mathrm{last}(\tau|_{\eta,j})$. Since $\tau|_{\eta,j}$ is of *Locs*-type C or E, then $\ell_2 \in Locs$. □

And finally, we find witnesses also in the case when there is a thread of *Locs*-type E, and a thread of *Locs*-type D.

Proposition 4.30. *Let σ be a configuration, let τ be a steady conventional schedule applicable to σ, and let η be a decomposition of σ and τ. Fix a subset Locs of $\mathcal{L}$. For $i, j \in \Theta(\sigma, \tau, \eta)$, let $\tau|_{\eta,i}$ be a thread of Locs-type E, and let $\tau|_{\eta,j}$ be a thread of Locs-type D. Let us write $\tau|_{\eta,j}$ as $\tau|^1_{\eta,j} \cdot \tau|^2_{\eta,j}$, where $\mathrm{last}(\tau|^1_{\eta,j}) \in Locs$. If we denote*

$$\tau^* = \tau|^1_{\eta,j} \cdot \tau|_{\eta,i} \cdot \tau|^2_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i,j\}},$$

then we obtain that

1. $\mathsf{Cfgs}(\sigma, \tau|^1_{\eta,j}) \models \boldsymbol{\kappa}[\ell_1] \neq 0$, *for* $\ell_1 = \mathrm{first}(\tau|_{\eta,i}) \in \mathit{Locs}$,

2. $\mathsf{Cfgs}(\tau|^1_{\eta,j}(\sigma), \tau|_{\eta,i}) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, *for* $\ell_2 = \mathrm{last}(\tau|^1_{\eta,j}) \in \mathit{Locs}$,

3. $\mathsf{Cfgs}(\tau|^1_{\eta,j} \cdot \tau|_{\eta,i}(\sigma), \tau|^2_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i,j\}}) \models \boldsymbol{\kappa}[\ell_3] \neq 0$, *for* $\ell_3 = \mathrm{last}(\tau|_{\eta,i}) \in \mathit{Locs}$.

Proof. By Proposition 4.18, τ^* is a steady schedule applicable to σ, $\tau^*(\sigma) = \tau(\sigma)$, and there exists a decomposition η^* of σ and τ^* such that $\tau|_{\eta,i} = \tau^*|_{\eta^*,i}$ and $\tau|_{\eta,j} = \tau^*|_{\eta^*,j}$. Let $l_1 = |\tau|^1_{\eta,j}|$ and $l_2 = |\tau|_{\eta,i}|$.

1) Let τ' be a prefix of $\tau|^1_{\eta,j}$, and therefore a prefix of τ^* of length $l \leq l_1$. By Lemma 4.26, we have that $\tau'(\sigma) \models \boldsymbol{\kappa}[\ell_1] \neq 0$, where $\ell_1 = \mathrm{first}(\tau|_{\eta,i})$. Since $\tau|_{\eta,i}$ is of *Locs*-type E, it is $\ell_1 \in \mathit{Locs}$.

2) Let τ' be a prefix of $\tau|_{\eta,i}$. Then $\tau|^1_{\eta,j} \cdot \tau'$ is a prefix of $\tau|^1_{\eta,j} \cdot \tau|_{\eta,i}$ of length $l \geq l_1$. We apply Lemma 4.25 for the configuration σ, the schedule $\tau|^1_{\eta,j} \cdot \tau|_{\eta,i}$, and the decomposition η^*. With the decomposition η^*, schedule $\tau|^1_{\eta,j}$ is a thread of σ and $\tau|^1_{\eta,j} \cdot \tau|_{\eta,i}$, and therefore from Lemma 4.25 we obtain that $\tau|^1_{\eta,j} \cdot \tau'(\sigma) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, or, equivalently, $\tau'(\tau|^1_{\eta,j}(\sigma)) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, where $\ell_2 = \mathrm{last}(\tau|^1_{\eta,j})$. From the construction of $\tau|^1_{\eta,j}$ follows that $\ell_2 \in \mathit{Locs}$.

3) Let τ' be a prefix of $\tau|^2_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i,j\}}$. Then $\tau'' = \tau|_{\eta,i} \cdot \tau'$ is a prefix of $\tau^*_1 = \tau|_{\eta,i} \cdot \tau|^2_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i,j\}}$ of length $l \geq l_2$. We define a naming η_1 of $\tau|^1_{\eta,j}(\sigma)$ and τ^*_1, for every $n \in \mathbb{N}$, as follows:

$$\eta_1(n) = \eta^*(n + l_1).$$

Note that η_1 is a decomposition of $\tau|^1_{\eta,j}(\sigma)$ and τ^*_1, and $\tau^*_1|_{\eta_1,i} = \tau^*|_{\eta^*,i} = \tau|_{\eta,i}$. We apply Lemma 4.25 for the configuration $\tau|^1_{\eta,j}(\sigma)$, the schedule τ^*_1, and the decomposition η_1, and obtain that for the prefix τ'' of τ^*_1 holds $\tau''(\tau|^1_{\eta,j}(\sigma)).\boldsymbol{\kappa}[\ell_3] \geq 1$, or, equivalently,

$$\tau'(\tau|_{\eta,i}(\tau|^1_{\eta,j}(\sigma))).\boldsymbol{\kappa}[\ell_3] \geq 1,$$

where $\ell_3 = \mathrm{last}(\tau^*_1|_{\eta_1,i}) = \mathrm{last}(\tau|_{\eta,i})$. Again, as $\tau|_{\eta,i}$ is of *Locs*-type E, then $\ell_3 \in \mathit{Locs}$. □

Constructing representative schedules. We first consider building a representative for the trivial case, when there is a location that is globally non-empty in a path. It remains non-empty in the representative path, and therefore, the representative maintains the required property.

Proposition 4.31. *Let σ be a configuration, and let τ be a steady conventional schedule applicable to σ. Fix a set Locs $\subseteq \mathcal{L}$. If there exist a local state $\ell^* \in$ Locs such that $\mathsf{Cfgs}(\sigma, \tau) \models \boldsymbol{\kappa}[\ell^*] \neq 0$, then*

$$\mathsf{Cfgs}(\sigma, \mathsf{srep}[\sigma, \tau]) \models \bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0.$$

Proof. If there is a local state $\ell^* \in \mathit{Locs}$ such that $\mathsf{Cfgs}(\sigma, \tau) \models \boldsymbol{\kappa}[\ell^*] \neq 0$, then we have $\mathsf{Cfgs}(\sigma, \mathsf{srep}[\sigma, \tau]) \models \boldsymbol{\kappa}[\ell^*] \neq 0$, by Proposition 4.21. Therefore, $\mathsf{Cfgs}(\sigma, \mathsf{srep}[\sigma, \tau]) \models \bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0$. □

When constructing a representative schedule, a standard strategy is to find representatives of its subschedules, and concatenate them.

Proposition 4.32. *Let σ be a configuration, let $\tau = \tau_1 \cdot \ldots \cdot \tau_n$, for $n \geq 1$, be a steady conventional schedule applicable to σ. Fix a set $\mathit{Locs} \subseteq \mathcal{L}$. If we denote*

$$\tau^* = \mathsf{srep}[\sigma, \tau_1] \cdot \mathsf{srep}[\tau_1(\sigma), \tau_2] \cdot \ldots \cdot \mathsf{srep}[\tau_1 \cdot \ldots \cdot \tau_{n-1}(\sigma), \tau_n],$$

then the following holds:

a) τ^* is applicable to σ, and $\tau^*(\sigma) = \tau(\sigma)$,

b) $|\tau^*| \leq 2 \cdot n \cdot |\mathcal{R}|$.

Proof. Firstly note that for every k with $1 \leq k \leq n$, τ_k is a steady conventional schedule applicable to $\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma)$. Therefore, by Proposition 4.2, for every k with $1 \leq k \leq n$ holds that

- $\mathsf{srep}[\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \tau_k]$ is applicable to $\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma)$,
- $\mathsf{srep}[\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \tau_k](\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma)) = \tau_k(\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma)) = \tau_1 \cdot \ldots \cdot \tau_k(\sigma)$,
- $|\mathsf{srep}[\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \tau_k]| \leq 2 \cdot |\mathcal{R}|$.

The first two observations imply the statement a), and the third one implies b). □

This allows us to construct representatives that maintain formula $\bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0$ whenever subschedules are either of *Locs*-type A, or if there is a witness during their execution, as it was the case in Propositions 4.28–4.30.

Proposition 4.33. *Fix a set $\mathit{Locs} \subseteq \mathcal{L}$. Let σ be a configuration, let $\tau_1 \cdot \ldots \cdot \tau_n$, for $n \geq 1$, be a steady conventional schedule applicable to σ, and let $\psi \equiv \bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0$. If for every k with $1 \leq k \leq n$ holds at least one of the following:*

a) τ_k is a thread of σ and $\tau_1 \cdot \ldots \cdot \tau_n$, of Locs-type A,

b) $\mathsf{Cfgs}(\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \tau_k) \models \boldsymbol{\kappa}[\ell] \neq 0$, for some $\ell \in \mathit{Locs}$,

then $\mathsf{Cfgs}(\sigma, \tau^*) \models \psi$, for $\tau^* = \mathsf{srep}[\sigma, \tau_1] \cdot \mathsf{srep}[\tau_1(\sigma), \tau_2] \cdot \ldots \cdot \mathsf{srep}[\tau_1 \cdot \ldots \cdot \tau_{n-1}(\sigma), \tau_n]$.

Proof. For every k, $1 \leq k \leq n$, we know that τ_k is a steady conventional schedule applicable to $\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma)$. We prove the statement by showing that for every k with $1 \leq k \leq n$ holds that

$$\mathsf{Cfgs}(\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \mathsf{srep}[\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \tau_k]) \models \psi.$$

If we fix one such k, then there are two cases:

- If τ_k is a thread of σ and $\tau_1 \cdot \ldots \cdot \tau_n$ of *Locs*-type A, then Proposition 4.24 yields the required.
- If there exists an $\ell \in Locs$ such that $\mathsf{Cfgs}(\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \tau_k) \models \boldsymbol{\kappa}[\ell] \neq 0$, then by Proposition 4.21 we know that $\mathsf{Cfgs}(\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \mathsf{srep}[\tau_1 \cdot \ldots \cdot \tau_{k-1}(\sigma), \tau_k]) \models \boldsymbol{\kappa}[\ell] \neq 0$, which implies the required.

□

Bringing it all together. Let us now prove that we can always find a representative schedule of bounded length that maintains $\bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$. A complete proof of the following theorem is in Section 4.6.1.

Theorem 4.34. *Fix a threshold automaton $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$, and a set $Locs \subseteq \mathcal{L}$. Let σ be a configuration such that $\omega(\sigma) = \Omega$, and let $\psi \equiv \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$. Then for every steady conventional schedule τ, applicable to σ, with $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, there is a steady schedule $\mathsf{repr}_\vee[\psi, \sigma, \tau]$ with the properties:*

a) $\mathsf{repr}_\vee[\psi, \sigma, \tau]$ *is applicable to σ, and* $\mathsf{repr}_\vee[\psi, \sigma, \tau](\sigma) = \tau(\sigma)$,

b) $|\mathsf{repr}_\vee[\psi, \sigma, \tau]| \leq 6 \cdot |\mathcal{R}|$,

c) $\mathsf{Cfgs}(\sigma, \mathsf{repr}_\vee[\psi, \sigma, \tau]) \models \psi$,

d) there exist τ_1, τ_2 and τ_3, (not necessarily nonempty) subschedules of τ, such that $\tau_1 \cdot \tau_2 \cdot \tau_3$ is applicable to σ, it holds that $\tau_1 \cdot \tau_2 \cdot \tau_3(\sigma) = \tau(\sigma)$, and

$$\mathsf{repr}_\vee[\psi, \sigma, \tau] = \mathsf{srep}[\sigma, \tau_1] \cdot \mathsf{srep}[\tau_1(\sigma), \tau_2] \cdot \mathsf{srep}[\tau_1 \cdot \tau_2(\sigma), \tau_3].$$

Proof Sketch: Proposition 4.27 provides us with a case distinction. To prove the theorem, for each of the cases we construct a representative schedule. We do so by repeatedly using Proposition 4.14, to reorder transitions in the following way: In Case 1 we move the thread of *Locs*-type A to the beginning of the schedule. Then, the representative schedule is obtained by applying Proposition 4.2 to the thread of *Locs*-type A and then to the rest. In Case 2 we move the thread of *Locs*-type C or E to the beginning, and again apply Proposition 4.2 to the thread and the rest. Case 3 is the most involved construction. A prefix of the *Locs*-type D thread is moved to the beginning followed by the complete *Locs*-type E thread. Proposition 4.2 is applied to the prefix, the thread and the rest. In the trivial case, when the assumption of the proposition is not satisfied, that is, if there is a local state $\ell \in Locs$ such that $\mathsf{Cfgs}(\sigma, \tau) \models \boldsymbol{\kappa}[\ell] \neq 0$, then we just apply Proposition 4.2 to τ. □

4.3.2 Representative Schedules maintaining $\bigwedge_{Locs \in Y} \bigvee_{i \in Locs} \boldsymbol{\kappa}[i] \neq 0$

The construction we give in this section requires us to apply the same schedule twice. We confirm that we can do that by proving in Proposition 4.35 that if a counterexample exists in a small system, there also exists one in a bigger system. In the context of counter systems we formalize this using a multiplier:

Definition 4.11 (Multiplier). *A* multiplier *μ of a threshold automaton is a number $\mu \in \mathbb{N}$, such that for every guard φ, if $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g}, \sigma.\mathbf{p}) \models \varphi$, then also $(\mu \cdot \sigma.\boldsymbol{\kappa}, \mu \cdot \sigma.\mathbf{g}, \mu \cdot \sigma.\mathbf{p}) \models \varphi$, and $\mu \cdot \sigma.\mathbf{p} \in \mathbf{P}_{RC}$.*

For specific pathological threshold automata, such multipliers may not exist. However, all our benchmarks have multipliers, and as can be seen from the definitions, existence of multipliers can easily be checked using simple queries to SMT solvers in preprocessing.

Definition 4.12. *If σ is a configuration, and $\mu \geq 1$ is a multiplier, then we define $\mu\sigma$ to be the configuration with $(\mu\sigma).\boldsymbol{\kappa} = \mu \cdot \sigma.\boldsymbol{\kappa}$, $(\mu\sigma).\mathbf{g} = \mu \cdot \sigma.\mathbf{g}$, and $(\mu\sigma).\mathbf{p} = \mu \cdot \sigma.\mathbf{p}$. If τ is a conventional schedule, we define $\mu\tau = \underbrace{\tau \cdot \ldots \cdot \tau}_{\mu \text{ times}}$.*

Let us present some properties of a multiplied system.

Proposition 4.35. *Let σ_1, σ_1' and σ_2 be configurations, let τ be a steady conventional schedule applicable to σ_1 and σ_2, and let ℓ be an arbitrary local state. If a multiplier is $\mu > 1$, then the following holds:*

1. *$\mu\tau$ is applicable to $\mu\sigma_1$, and if $\tau(\sigma_1) = \sigma_1'$ then $\mu\tau(\mu\sigma_1) = \mu\sigma_1'$,*
2. *for every propositional formula ψ, if $\sigma \models \psi$, then $\mu\sigma \models \psi$,*
3. *if $\sigma_1.\boldsymbol{\kappa}[\ell] < \sigma_2.\boldsymbol{\kappa}[\ell]$, then $\tau(\sigma_1).\boldsymbol{\kappa}[\ell] < \tau(\sigma_2).\boldsymbol{\kappa}[\ell]$,*
4. *if $\sigma_1.\boldsymbol{\kappa}[\ell] > 0$ then $\mu\sigma_1.\boldsymbol{\kappa}[\ell] > \sigma_1.\boldsymbol{\kappa}[\ell]$.*

Proof. All properties follow directly from the definition of a counter system. □

Proposition 4.36. *Let σ be a configuration, let τ be a steady conventional schedule applicable to σ, and let ψ be a propositional formula. If μ is a multiplier and if $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, then $\mathsf{Cfgs}(\mu\sigma, \mu\tau) \models \psi$.*

Proof. Paths $\mathsf{Cfgs}(\sigma,\tau)$ and $\mathsf{Cfgs}(\mu\sigma,\mu\tau)$ are trace equivalent by Proposition 4.35 (1) and (2). Therefore, by [BK08, Corollary 3.8], they satisfy the same linear temporal properties. □

Following a similar argument as in the previous cases, namely by cutting a schedule to convenient subschedules, gives us a strategy for constructing representatives that maintain formula $\bigwedge_{1\le m\le n}\bigvee_{\ell\in Locs_m}\boldsymbol{\kappa}[\ell]\neq 0$.

Theorem 4.4. *Fix a threshold automaton $\mathsf{TA}=(\mathcal{L},\mathcal{V},\mathcal{R},RC)$ that has a finite multiplier μ, and a configuration σ. For an $n\in\mathbb{N}$, fix sets of locations $Locs_m\subseteq\mathcal{L}$ for $1\le m\le n$. If we have*

$$\psi=\bigwedge_{1\le m\le n}\bigvee_{\ell\in Locs_m}\boldsymbol{\kappa}[\ell]\neq 0,$$

then for every steady conventional schedule τ, applicable to σ, with $\mathsf{Cfgs}(\sigma,\tau)\models\psi$, there exists a schedule $\mathsf{repr}_{\wedge\vee}[\psi,\mu\sigma,\mu\tau]$ with the following properties:

a) The representative is applicable and ends in the same final state: $\mathsf{repr}_{\wedge\vee}[\psi,\mu\sigma,\mu\tau]$ is a steady schedule applicable to $\mu\sigma$, and $\mathsf{repr}_{\wedge\vee}[\psi,\mu\sigma,\mu\tau](\mu\sigma)=\mu\tau(\mu\sigma)$,

b) The representative has bounded length: $|\mathsf{repr}_{\wedge\vee}[\psi,\mu\sigma,\mu\tau]|\le 4\cdot|\mathcal{R}|$,

c) The representative maintains the formula ψ, i.e., $\mathsf{Cfgs}(\mu\sigma,\mathsf{repr}_{\wedge\vee}[\psi,\mu\sigma,\mu\tau])\models\psi$,

d) The representative is a concatenation of two representative schedules srep from Proposition 4.2:

$$\mathsf{repr}_{\wedge\vee}[\psi,\mu\sigma,\mu\tau]=\mathsf{srep}[\mu\sigma,\tau]\cdot\mathsf{srep}[\tau(\mu\sigma),(\mu-1)\tau].$$

Proof Sketch: To prove the theorem, we use the schedule $\mu\tau$. As in the previous cases, we divide it into two parts, namely to τ and $(\mu-1)\tau$, and then apply Proposition 4.2 to both of them separately. For the proof, we use statements (3) and (4) from Proposition 4.35. Detailed proof is in Section 4.6.2. □

4.3.3 Representative Schedules maintaining $\bigwedge_{i\in Locs}\boldsymbol{\kappa}[i]=0$

This case is the simplest one, so that $\mathsf{srep}[\sigma,\tau]$ from Section 4.3 can directly be used as representative schedule.

Theorem 4.37. *Fix a threshold automaton $\mathsf{TA}=(\mathcal{L},\mathcal{V},\mathcal{R},RC)$, and a configuration σ. If we have that*

$$\psi\equiv\bigwedge_{i\in Locs}\boldsymbol{\kappa}[i]=0,$$

for $Locs\subseteq\mathcal{L}$, then for every steady schedule τ applicable to σ, and with $\mathsf{Cfgs}(\sigma,\tau)\models\psi$, schedule $\mathsf{srep}[\sigma,\tau]$ satisfies:

a) $\mathsf{srep}[\sigma,\tau]$ *is applicable to* σ, *and* $\mathsf{srep}[\sigma,\tau](\sigma) = \tau(\sigma)$,

b) $|\mathsf{srep}[\sigma,\tau]| \le 2 \cdot |\mathcal{R}|$,

c) $\mathit{Cfgs}(\sigma, \mathsf{srep}[\sigma,\tau]) \models \psi$.

Proof. Since $\mathsf{Cfgs}(\sigma,\tau) \models \psi$, we know that for every transition t from τ and for every local state $\ell \in \mathit{Locs}$ it holds $t.\mathit{from} \neq \ell$ and $t.\mathit{to} \neq \ell$. Let $\mathsf{repr}_\vee[\psi,\sigma,\tau] = \mathsf{srep}[\sigma,\tau]$. By Proposition 4.20, $\mathsf{srep}[\sigma,\tau]$ contains a subset of the rules that appear in τ. Hence, $\mathsf{repr}_\vee[\psi,\sigma,\tau]$ does not change counters of states in Locs. Other properties follow from Proposition 4.2 □

4.3.4 Proof of the Central Theorem 4.3

Now we are ready to prove our main theorem, given in Section 4.1.3, that we recall here:

Theorem 4.3. *Let* $\mathit{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$ *be a threshold automaton, and let* $\mathit{Locs} \subseteq \mathcal{L}$ *be a set of locations. Let* σ *be a configuration, let* τ *be a steady conventional schedule applicable to* σ, *and let* ψ *be one of the following formulas:*

$$\bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0, \quad \text{or} \quad \bigwedge_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] = 0.$$

If all configurations visited by τ *from* σ *satisfy* ψ, *i.e.,* $\mathit{Cfgs}(\sigma,\tau) \models \psi$, *then there is a steady representative schedule* $\mathsf{repr}[\psi,\sigma,\tau]$ *with the following properties:*

a) The representative is applicable, and ends in the same final state: $\mathsf{repr}[\psi,\sigma,\tau]$ *is applicable to* σ, *and* $\mathsf{repr}[\psi,\sigma,\tau](\sigma) = \tau(\sigma)$,

b) The representative has bounded length: $|\mathsf{repr}[\psi,\sigma,\tau]| \le 6 \cdot |\mathcal{R}|$,

c) The representative maintains the formula ψ, *i.e.,* $\mathit{Cfgs}(\sigma, \mathsf{repr}[\psi,\sigma,\tau]) \models \psi$,

d) The representative is a concatenation of three representative schedules srep *from Proposition 4.2: there exist* τ_1, τ_2 *and* τ_3, *(possibly empty) subschedules of* τ, *such that* $\tau_1 \cdot \tau_2 \cdot \tau_3$ *is applicable to* σ, *and it holds that* $(\tau_1 \cdot \tau_2 \cdot \tau_3)(\sigma) = \tau(\sigma)$, *and*

$$\mathsf{repr}[\psi,\sigma,\tau] = \mathsf{srep}[\sigma,\tau_1] \cdot \mathsf{srep}[\tau_1(\sigma),\tau_2] \cdot \mathsf{srep}[(\tau_1 \cdot \tau_2)(\sigma),\tau_3].$$

The theorem follows from Theorem 4.34 and 4.37.

4.4 Application of the Short Counterexample Property and Experiments

4.4.1 SMT Encoding

We use the theoretical results from Section 4.1.3 and Section 4.3 to give an efficient encoding of lasso-shaped executions in SMT with linear integer arithmetic.

We encode the lassos in SMT using the same idea as in the previous chapter in Section 3.7.2. Namely, given a schedule τ, we encode in linear integer arithmetic the paths that follow this schedule from an initial state as follows:

$$E(\tau) \equiv \mathrm{init}(0) \wedge T(0,1) \wedge T(1,2) \wedge \dots,$$

where $\mathrm{init}(0)$ is the predicate describing the initial state, and $T(i-1,i)$ encodes the i-th transition in τ.

If τ encodes a lasso shape, and the SMT solver reports a satisfying assignment, this assignment is a counterexample. If the SMT solver reports unsatisfiability of the formula on all lassos discussed in Section 4.2.3, then there does not exist a counterexample and the algorithm is verified.

Example 4.13. In Example 2.8 we have seen the fairness requirement ψ_{fair}, which is a property of a configuration that can be encoded as $\mathrm{fair}(i) \equiv \boldsymbol{\kappa}_1^i = 0 \wedge (x^i \geq t+1 \rightarrow \boldsymbol{\kappa}_0^i = 0 \wedge \boldsymbol{\kappa}_1^i = 0) \wedge (x^i \geq n-t \rightarrow \boldsymbol{\kappa}_0^i = 0 \wedge \boldsymbol{\kappa}_2^i = 0)$, which is a formula in linear integer arithmetic. Then, e.g., $\mathrm{fair}(5)$ encodes that the fifth configuration satisfies the predicate. Such state formulas can be added as conjunct to the formula $E(\tau)$ that encodes a path. $\triangleleft$

As discussed in Sections 4.2 and 4.2.3 we have to encode lassos of the form $\vartheta \cdot \rho^\omega$ starting from an initial configuration σ. We immediately obtain a finite representation by encoding the fixed length execution $E(\vartheta \cdot \rho)$ as in Section 3.7.2, and adding the constraint that applying ρ returns to the start of the lasso loop, that is, $\vartheta(\sigma) = \rho(\vartheta(\sigma))$. In SMT this is directly encoded as equality on integer variables.

4.4.2 Generating the SMT Queries

The high-level structure of the verification algorithm is given in Figure 4.1 on page 76. In this section, we give the details of the procedure `check_one_order`, whose pseudo code is given in Figure 4.9. It receives as the input the following parameters: a threshold automaton TA, an $\mathsf{ELTL}_{\mathsf{FT}}$ formula φ, a cut graph $\mathcal{G}$ of φ, a threshold graph $\mathcal{H}$ of TA, and a topological order $\prec$ on the vertices of the graph $\mathcal{G} \cup \mathcal{H}$.

The procedure `check_one_order` constructs SMT assertions about the configurations of the lassos that correspond to the order $\prec$. Similarly as in Section 3.7.2, an ith configuration is defined by the vectors of SMT variables $(\boldsymbol{\kappa}^i, \mathbf{g}^i, \mathbf{p})$. We use two global variables: the number fn of the configuration under construction, and the number fs of

```
variables fn, fs; // the current configuration number and the loop start
// Try to find a witness lasso for: a threshold automaton TA,
// an $\mathsf{ELTL}_{\mathsf{FT}}$ formula $\varphi$, a cut graph $\mathcal{G}$, a threshold graph $\mathcal{H}$, and
// a topological order $\prec$ on the nodes of $\mathcal{G} \cup \mathcal{H}$.
procedure check_one_order(TA, $\varphi$, $\mathcal{G}$, $\mathcal{H}$, $\prec$):
  fn := 0; fs := 0;
  SMT_start(); // start (or reset) the SMT solver
  assume($can(\varphi) = \psi_0 \wedge \mathbf{F}\,\psi_1 \wedge \dots \wedge \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$);
  SMT_assert($\boldsymbol{\kappa}^0, \mathbf{g}^0, \mathbf{p} \models \mathsf{init}(0) \wedge \psi_0 \wedge \psi_{k+1}$);
  $v_0 := \min_{\prec}(\mathcal{V}_{\mathcal{G}} \cup \mathcal{V}_{\mathcal{H}})$; // the minimal node w.r.t. the linear order $\prec$
  check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, $v_0$, $\psi_{k+1}$, $\emptyset$);

// Try to find a witness lasso starting with the node $v$ and the context $\Omega$,
// while preserving the invariant $\psi_{inv}$.
recursive procedure check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, $v$, $\psi_{inv}$, $\Omega$):
  if not SMT_sat() then:
    return no_witness;
  case (a) $v \in \mathcal{V}_{\mathcal{G}} \setminus \{\mathsf{loop}_{\mathsf{start}}, \mathsf{loop}_{\mathsf{end}}\}$:
    find $\psi$ s.t. $\langle \mathbf{F}\,\psi, v\rangle \in \mathcal{T}(\varphi)$; // $v$ labels a formula in the syntax tree
    assume($\psi = \psi_0 \wedge \mathbf{F}\,\psi_1 \wedge \dots \wedge \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$);
    SMT_assert($\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p} \models \psi_0$);
    push_segment($\psi_{inv} \wedge \psi_{k+1}$);
    $v' := \min_{\prec}(\mathcal{V}_{\mathcal{G}} \cup \mathcal{V}_{\mathcal{H}}) \cap \{w : v \prec w\}$; // the next node after $v$
    check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, $v'$, $\psi_{inv} \wedge \psi_{k+1}$, $\Omega$);
  case (b) $v \in \mathcal{V}_{\mathcal{H}} \setminus \{\mathsf{loop}_{\mathsf{start}}, \mathsf{loop}_{\mathsf{end}}\}$: // $v$ is a threshold guard
    if $v \in \Phi^{\mathrm{rise}}$ then: // $v$ is an unlocking guard, e.g., $x \geq t + 1 - f$
      push_segment($\psi_{inv}$); // one rule unlocks $v$
      SMT_assert($\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p} \models v$); // $v$ is unlocked
      push_segment($\psi_{inv}$); // execute all unlocked rules
      $v' := \min_{\prec}(\mathcal{V}_{\mathcal{G}} \cup \mathcal{V}_{\mathcal{H}}) \cap \{w : v \prec w\}$; // the next node after $v$
      check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, $v'$, $\psi_{inv}$, $\Omega \cup \{v\}$);
    else: /* $v \in \Phi^{\mathrm{fall}}$, e.g., $x < f$, similar to the locking case: use $\neg v$ */
  case (c) $v = \mathsf{loop}_{\mathsf{start}}$:
    fs := fn; // the loop starts at the current configuration
    push_segment($\psi_{inv}$); // execute all unlocked rules
    $v' := \min_{\prec}(\mathcal{V}_{\mathcal{G}} \cup \mathcal{V}_{\mathcal{H}}) \cap \{w : v \prec w\}$; // the next node after $v$
    check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, $v'$, $\psi_{inv}$, $\Omega$);
  case (d) $v = \mathsf{loop}_{\mathsf{end}}$:
    SMT_assert($\boldsymbol{\kappa}^{\mathsf{fn}} = \boldsymbol{\kappa}^{\mathsf{fs}} \wedge \mathbf{g}^{\mathsf{fn}} = \mathbf{g}^{\mathsf{fs}}$); // close the loop
    if SMT_sat() then:
      return witness(SMT_model())

// Encode a segment of rules as prescribed by [KVW15] and Theorems 4.3–4.4.
procedure push_segment($\psi_{inv}$):
  // find the number of schedules to repeat in (d) of Theorems 4.3, 4.4
  nrepetitions := compute_repetitions($\psi_{inv}$);
  $r_1, \dots, r_k$ := compute_rules($\Omega$); // use $\mathsf{sschema}_{\Omega}$ from [KVW15]
  for _ from 1 to nrepetitions:
    for j from 1 to $k$:
      SMT_assert($\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p} \models \psi_{inv}$);
      SMT_assert($T(\mathsf{fn}, r_j)$);
      fn := fn + 1; // move to the next configuration
```

Figure 4.9: Checking one topological order with SMT.

the configuration that corresponds to the loop start. Thus, with the expressions $\boldsymbol{\kappa}^{\mathsf{fn}}$ and $\mathbf{g}^{\mathsf{fn}}$ we refer to the SMT variables of the configuration whose number is stored in fn.

In the pseudocode in Figure 4.9, we call `SMT_assert(`$\boldsymbol{\kappa}^{\mathsf{fn}}$`,` $\mathbf{g}^{\mathsf{fn}}$`,` $\mathbf{p} \models \psi$`)` to add an assertion ψ about the configuration $(\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p})$ to the SMT query. Finally, the call `SMT_sat()` returns true, only if there is a satisfying assignment for the assertions collected so far. Such an assignment can be accessed with `SMT_model()` and gives the values for the configurations and acceleration factors, which together constitute a witness lasso.

The procedure `check_one_order` creates the assertions about the initial configurations. The assertions consist of: the assumptions $\mathsf{init}(0)$ about the initial configurations of the threshold automaton, the top-level propositional formula ψ_0, and the invariant propositional formula ψ_{k+1} that should hold from the initial configuration on. By writing `assume(`$\psi = \psi_0 \wedge \mathbf{F} \wedge \psi_1 \ldots \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$`)`, we extract the subformulas of a canonical formula ψ (see Section 4.2.1). The procedure finds the minimal node in the order $\prec$ on the nodes of the graph $\mathcal{G} \cup \mathcal{H}$ and calls the procedure `check_node` for the initial node, the initial invariant ψ_{k+1}, and the empty context $\emptyset$.

The procedure `check_node` is called with a node v of the graph $\mathcal{G} \cup \mathcal{H}$ as a parameter. It adds assertions that encode a finite path and constraints on the configurations of this path. The finite path leads from the configuration that corresponds to the node v to the configuration that corresponds to v's successor in the order $\prec$. The constraints depend on v's origin: (a) v labels a formula $\mathbf{F}\,\psi$ in the syntax tree of φ, (b) v carries a threshold guard from the set $\Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}}$, (c) v denotes the loop start, or (d) v denotes the loop end. In case (a), we add an SMT assertion that the current configuration satisfies the propositional formula $\mathit{prop}(\psi)$ (line 21), and add a sequence of rules that leads to v's successor while maintaining the invariants ψ_{inv} of the preceding nodes and the v's invariant ψ_{k+1} (line 22). In case (b), in line 27, we add a sequence of rules, one of which should unlock (resp. lock) the threshold guard in $v \in \Phi^{\mathrm{rise}}$ (resp. $v \in \Phi^{\mathrm{fall}}$). Then, in line 29, we add a sequence of rules that leads to a configuration of v's successor. All added configurations are required to satisfy the current invariant ψ_{inv}. As the threshold guard in v is now unlocked (resp. locked), we include the guard (resp. its negation) in the current context Ω. In case (c), we store the current configuration as the loop start in the variable fs and, as in (a) and (b), add a sequence of rules leading to v's successor. Finally, in case (d), we should have reached the ending configuration that coincides with the loop start. To this end, in line 39, we add the constraint that forces the counters of both configurations to be equal. At this point, all the necessary SMT constraints have been added, and we call `SMT_sat` to check whether there is an assignment that satisfies the constraints. If there is one, we report it as a lasso witnessing the $\mathsf{ELTL}_{\mathsf{FT}}$-formula φ that consists of: the concrete parameter values, the values of the counters and shared variables for each configuration, and the acceleration factors. Otherwise, we report that there is no witness lasso for the formula φ.

The procedure `push_segment` constructs a sequence of currently unlocked rules, as in the case of reachability from Chapter 3. However, this sequence should be repeated several

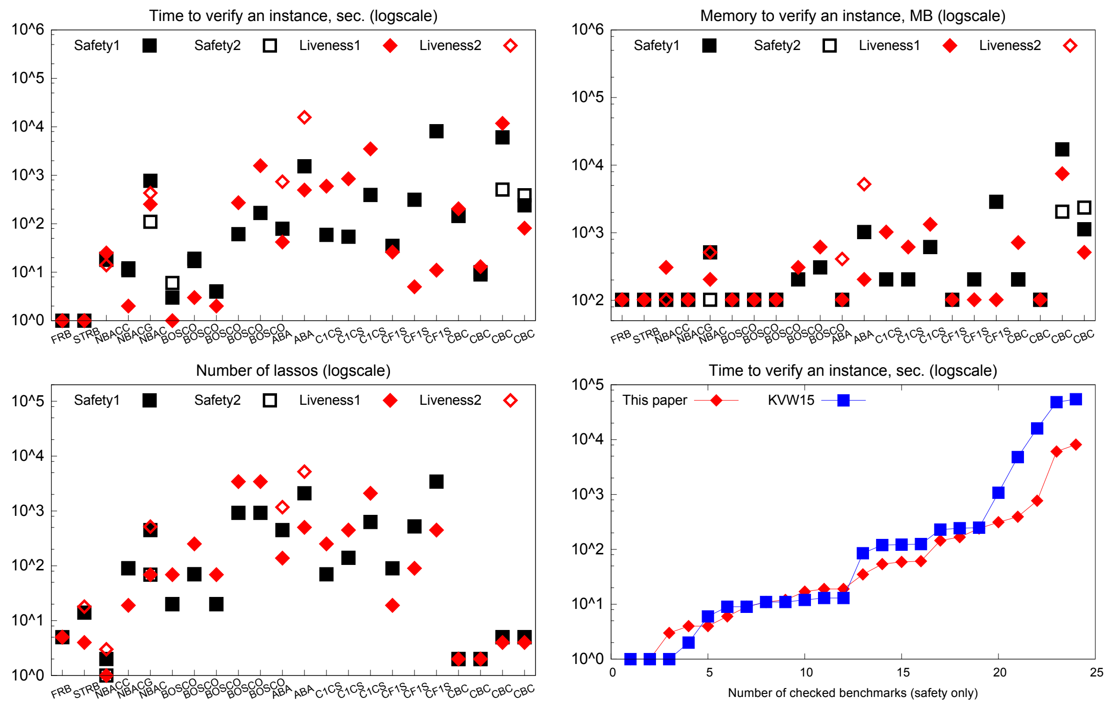


Figure 4.10: The plots summarize the following results of running our implementation on all benchmarks: used time in seconds (top left), used memory in megabytes (top right), the number of checked lassos (bottom left), time used both by our implementation and [KVW15] to check *safety only* (bottom right). Several occurrences of the same benchmark correspond to different cases, such as $f > 1$, $f = 1$, and $f = 0$. Symbols ■ and □ correspond to the safety properties of each benchmark, while symbols ♦ and ♢ correspond to the liveness properties.

times, as required by Theorems 4.3 and 4.4. Moreover, the freshly added configurations are required to satisfy the current invariant ψ_{inv}.

4.4.3 Experiments

Negations of the safety and liveness specifications of our benchmarks — written in $\mathsf{ELTL_{FT}}$ — follow three patterns: unsafety $\mathbf{E}\,(p \wedge \mathbf{F}\,q)$, non-termination $\mathbf{E}\,(p \wedge \mathbf{G}\,\mathbf{F}\,r \wedge \mathbf{G}\,q)$, and non-response $\mathbf{E}\,(\mathbf{G}\,\mathbf{F}\,r \wedge \mathbf{F}\,(p \wedge \mathbf{G}\,q))$. The propositions p, q, and r follow the syntax of *pform* (cf. Table 2.1), e.g., $p \equiv \bigwedge_{\ell \in Locs_1} \kappa[\ell] = 0$ and $q \equiv \bigvee_{\ell \in Locs_2} \kappa[\ell] \neq 0$ for some sets of locations $Locs_1$ and $Locs_2$.

The results of these experiments are summarized in Figure 4.10. Given the properties of the distributed algorithms found in the literature, we checked for each benchmark one or two safety properties (depicted with ■ and □) and one or two liveness properties (depicted with ♦ and ♢). For each benchmark, we display the running times and the memory used together by ByMC and the SMT solver Z3 [MB08], as well as the number of exercised lasso shapes as discussed in Section 4.2.3.

For safety properties, we also show the comparison of this implementation against the initial implementation presented in the conference paper [KVW15]. The results are summarized the bottom right plot in Figure 4.10, which shows that there is no clear winner. For instance, this implementation is 170 times faster on BOSCO for the case $n > 5t$. However, for the benchmark ABA we experienced a tenfold slowdown. In these experiments, attempts to improve the SMT encoding for liveness usually impaired safety results.

This implementation has verified safety and liveness of all ten parameterized algorithms in less than a day. Moreover, the tool reports counterexamples to liveness of CF1S and BOSCO exactly for the cases predicted by the distributed algorithms literature, i.e., when there are not enough correct processes to reach consensus in one communication step. Noteworthy, liveness of only the two simplest benchmarks (STRB and FRB) had been automatically verified before [JKS$^+$13a].

4.5 Detailed Proofs for Section 4.2

Proposition 4.1. *Given a threshold automaton* TA *and an* $\mathsf{ELTL}_{\mathsf{FT}}$ *formula* φ, *if* $\mathsf{Sys}(\mathsf{TA}) \models \mathbf{E}\varphi$, *then there are an initial configuration* $\sigma_1 \in I$ *and a schedule* $\tau \cdot \rho^\omega$ *with the following properties:*

(a) the path satisfies the formula: $\mathsf{path}(\sigma_1, \tau \cdot \rho^\omega) \models \varphi$,

(b) application of ρ *forms a cycle:* $\rho^k(\tau(\sigma_1)) = \tau(\sigma_1)$ *for* $k \geq 0$.

Proof. We do not give details on Büchi automata and the construction by Vardi and Wolper, since this construction is well-known and can be found in the original paper [VW86] as well as in a number of textbooks, e.g., [CGP99][Ch. 9] and [BK08][Ch. 5].

Using the construction from [VW86], we translate the formula φ into a Büchi automaton $B = (AP, Q, \Delta, Q^0, F)$, which has a finite set Q of states, a finite set $Q_0 \subseteq Q$ of initial states, a finite set F of accepting states, a finite alphabet AP of atomic propositions (which corresponds to the propositional formulas derived from *pform*), and the transition relation $\Delta \subseteq Q \times AP \times Q$. The key property is that the automaton B recognizes exactly those sequences of propositions that satisfy the formula φ.

Let (Σ, I, R) be the counter system $\mathsf{Sys}(\mathsf{TA})$ as defined in Section 2.2. The system $\mathsf{Sys}(\mathsf{TA})$ is a transition system, so following [VW86] we can construct the product Büchi automaton $\mathsf{Sys}(\mathsf{TA}) \otimes B$ that corresponds to the synchronous product of $\mathsf{Sys}(\mathsf{TA})$ and B. Formally, $\mathsf{Sys}(\mathsf{TA}) \otimes B$ is the Büchi automaton $(AP, Q_P, \Delta_P, Q_P^0, F_P)$ defined as follows:

- The set of states Q_P is the Cartesian product $(\Sigma \cup \{\iota\}) \times Q$, where $\iota \notin \Sigma$ is a dummy configuration, which is used to delay initialization of the counter system by one step.
- The set of initial states Q_P^0 is the Cartesian product $\{\iota\} \times Q^0$.
- The set of accepting states F_P is the Cartesian product $\Sigma \times F$.
- The transition relation Δ_P includes the following triples:
  - an initial transition $((\iota, q_0), p, (\sigma, q))$ for $q_0 \in Q_0$, $\sigma \in I$, and $q \in Q$ such that $(q_0, p, q) \in \Delta$ and $\sigma \models p$.
  - a transition $((\sigma, q), p, (\sigma', q'))$ for $q, q' \in Q$ and $\sigma, \sigma' \in \Sigma$ such that $(q, p, q') \in \Delta$ and $\sigma' \models p$.

A run of the product automaton is an infinite sequence (ι, q_0), $(\sigma_1, q_1), \dots, (\sigma_i, q_i), \dots$ such that $((\iota, q_0), p_0, (\sigma_1, q_1)) \in \Delta$ and $((\sigma_i, q_i), p_i, (\sigma_{i+1}, q_{i+1})) \in \Delta$ for $i \geq 1$ and some propositions $p_0, p_1, \dots \in AP$. The run is accepting, if there is a state $(\sigma_j, q_j) \in F_P$ that appears infinitely often in the run.

In contrast to [VW86], the product automaton $\mathsf{Sys}(\mathsf{TA}) \otimes B$ has infinitely many states. However, by Proposition 2.2, every path of $\mathsf{Sys}(\mathsf{TA})$ visits only finitely many states, and thus every run of the product automaton visits finitely many states too. Hence, in each run there are finitely many accepting states. Due to this, and since, by assumption, $\mathsf{Sys}(\mathsf{TA}) \models \mathbf{E}\,\varphi$, the product has an accepting run $(\iota, q_0), (\sigma_1, q_1), \dots, (\sigma_i, q_i), \dots$ with state $(\sigma_j, q_j) \in F_P$ appearing infinitely often for some $j \geq 1$. Hence, there is an index $k \geq 0$ such that $(\sigma_{j+k+1}, q_{j+k+1}) = (\sigma_j, q_j)$. Consequently, we construct a lasso run by taking the sequence of states $(\iota, q_0), (\sigma_1, q_1), \dots, (\sigma_{j-1}, q_{j-1})$ as a prefix and the sequence $(\sigma_j, q_j), \dots, (\sigma_{j+k}, q_{j+k})$ as a loop, which is repeated infinitely. This lasso run is also an accepting run of the product automaton.

It is immediate from the construction, that the infinite sequence of configurations $\sigma_1, \dots, \sigma_{j-1}, (\sigma_j, \dots, \sigma_{j+k})^\omega$ corresponds to a path of $\mathsf{Sys}(\mathsf{TA})$ starting from an initial configuration $\sigma_1 \in I$, and this path satisfies the formula φ. Thus, there are schedules $\tau = t_1, \dots, t_{j-1}$ and $\rho = t_j, \dots, t_{j+k}$ such that:

1. Schedule τ is applicable to σ_1 and the prefixes of τ visit the intermediate configurations:
$$(t_1, \dots, t_i)(\sigma_1) = \sigma_i \text{ for } 1 \leq i < j,$$
2. Schedule ρ is applicable to σ_j, the prefixes of ρ visit the intermediate configurations, and ρ closes the loop:
$$(t_j, \dots, t_m)(\sigma_j) = \begin{cases} \sigma_m, & \text{when } j \leq m < j + k \\ \sigma_j, & \text{when } m = j + k. \end{cases}$$

The infinite schedule $\tau \cdot \rho^\omega$ is the required schedule. Indeed, $\mathsf{path}(\sigma_1, \tau \cdot \rho^\omega) \models \varphi$ and $\rho^i(\tau(\sigma_1)) = \tau(\sigma_1)$ for $i \geq 0$. $\square$

In this particular proof we use Büchi automata, but not in the rest of the thesis. Since $\mathsf{ELTL_{FT}}$ uses only the temporal operators $\mathbf{F}$ and $\mathbf{G}$, we found it much easier to reason about the structure of $\mathsf{ELTL_{FT}}$ formulas directly (in the spirit of [EVW02]) and then apply path reductions, rather than constructing the synchronous product of a Büchi automaton and of a counter system and then finding proper path reductions.

Proposition 4.6. *Let φ be an $\mathsf{ELTL_{FT}}$ formula, σ be a configuration and $\tau \cdot \rho^\omega$ be a lasso schedule applicable to σ such that $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$ holds. There is a constant $K \geq 0$ and a cut function ζ such that for every $\langle \mathbf{F}\,\psi, w\rangle \in \mathcal{G}(\mathcal{T}(\varphi))$ if $\zeta(w)$ cuts $(\tau \cdot \rho^K) \cdot \rho$ into π' and π'', then ψ is satisfied at the cut point, that is, $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi$.*

Proof. For each node $\langle \mathbf{F}\,\psi, w\rangle \in \mathcal{T}(\varphi)$, we define an extreme appearance $EA(w)$ as follows:

1. If there is an index $k \in \{|\tau|, \dots, |\tau| + |\rho| - 1\}$ such that k cuts $\tau \cdot \rho$ in $\tau \cdot \rho'$ and ρ'', and it holds that $\mathsf{path}((\tau \cdot \rho')(\sigma), \rho'' \cdot \rho^\omega) \models \psi$, then we set $EA(w)$ to the maximal such $k \geq |\tau|$.
2. Otherwise, we set $EA(w)$ to the maximal $k < |\tau|$ such that k cuts τ in τ', τ'' and $\mathsf{path}(\tau'(\sigma), \tau'' \cdot \rho^\omega) \models \psi$. (Such k exists, as the case 1 does not apply, it holds $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \mathbf{F}\,\psi$, and since temporal formulas are connected only with the conjunction $\wedge$.)

Consider a topologically ordered sequence $v_1, v_2, \dots, v_{|\mathcal{V}_\mathcal{G}|}$ of the vertices of the cut graph $\mathcal{G}(\mathcal{T}(\varphi)) = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, that is, the condition $(v_i, v_j) \in \mathcal{E}_\mathcal{G}$ implies $i < j$ for $1 \leq i, j \leq |\mathcal{V}_\mathcal{G}|$. Such a sequence exists, since the graph $\mathcal{G}(\mathcal{T}(\varphi))$ is a directed acyclic graph. Let $\ell \in \{1, \dots, |\mathcal{V}_\mathcal{G}|\}$ be the index of the node $\mathsf{loop_{start}}$, i.e., $v_\ell = \mathsf{loop_{start}}$.

We unroll the loop $K = \ell - 1$ times. Formally, for $1 \leq i \leq |\mathcal{V}_\mathcal{G}|$, we set the cut point $\zeta(v_i)$ as follows:

$$\zeta(v_i) = \begin{cases} |\tau| + |\rho| \cdot K, & \text{if } v_i = \mathsf{loop_{start}} \\ |\tau| + |\rho| \cdot (K+1) - 1, & \text{if } v_i = \mathsf{loop_{end}} \\ EA(v_i), & \text{if } EA(v_i) < |\tau| \\ EA(v_i) + |\rho| \cdot (i-1), & \text{if } i < \ell \text{ and } |\tau| \leq EA(v_i) \\ EA(v_i) + |\rho| \cdot K, & \text{if } i \geq \ell \end{cases}$$

It is easy to see that ζ satisfies Definition 4.4. By the construction of extreme appearances, for a node $\langle \mathbf{F}\,\psi, w\rangle$, the formula ψ is satisfied at the extreme appearance $EA(w)$. Since $\zeta(w) - EA(w) = |\rho| \cdot i$ for some $i \geq 0$, it follows that if $\zeta(w)$ cuts $(\tau \cdot \rho^K) \cdot \rho^\omega$ into π' and π'', then $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi$ holds. $\square$

Lemma 4.38. *Let σ be a configuration, $\tau \cdot \rho^\omega$ be a lasso schedule applicable to σ, and φ be an $\mathsf{ELTL_{FT}}$ formula. If an index $k < |\tau|$ cuts τ into π' and π'' and $\mathsf{Cfgs}(\pi'(\sigma), \pi'' \cdot \rho) \models \varphi$ holds, then $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\varphi$ holds.*

Proof. From $\mathsf{Cfgs}(\pi'(\sigma), \pi'' \cdot \rho) \models \varphi$, we immediately conclude that two subsets of $\mathsf{Cfgs}(\pi'(\sigma), \pi'' \cdot \rho)$ also satisfy φ:

$$\mathsf{Cfgs}(\pi'(\sigma), \pi'') \models \varphi \tag{4.3}$$

$$\mathsf{Cfgs}(\tau(\sigma), \rho) \models \varphi \tag{4.4}$$

Since $\tau \cdot \rho^\omega$ is a lasso schedule, we have $\rho^i(\tau(\sigma)) = \tau(\sigma)$ for $i \geq 0$. From this and Equation (4.4), we conclude that $\mathsf{path}(\tau(\sigma), \rho^\omega) \models \mathbf{G}\,\varphi$ holds. By combining this with Equation (4.3), we arrive at the required property $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\varphi$. □

Lemma 4.39. *Let σ be a configuration, $\tau \cdot \rho^\omega$ be a lasso schedule applicable to σ, and φ be an $\mathsf{ELTL_{FT}}$ formula. If an index $k : |\tau| \leq k < |\tau| + |\rho|$ cuts $\tau \cdot \rho$ into π' and π'' and $\mathsf{Cfgs}(\tau(\sigma), \rho) \models \varphi$ holds, then $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\varphi$ holds.*

Proof. Since $\tau \cdot \rho^\omega$ is a lasso schedule, we have $\rho^i(\tau(\sigma)) = \tau(\sigma)$ for $i \geq 0$. Thus, $\mathsf{Cfgs}(\tau(\sigma), \rho) \models \varphi$ implies $\mathsf{path}(\tau(\sigma), \rho^\omega) \models \mathbf{G}\,\varphi$. As $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega)$ is a subsequence of $\mathsf{path}(\tau(\sigma), \rho^\omega)$, we arrive at $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\varphi$. □

Theorem 4.7. *Let σ be a configuration, $\tau \cdot \rho^\omega$ be a lasso applicable to σ, and φ be an $\mathsf{ELTL_{FT}}$ formula. If there is a witness of $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$, then the lasso $\tau \cdot \rho^\omega$ satisfies φ, that is $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$.*

Proof. Let the cut graph $\mathcal{G}(\mathcal{T}(\varphi))$ be $(\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$. We start with defining the notion of the parent cutpoint for a formula that has the form $\mathbf{G}\,\psi$. Given a tree node $\langle \mathbf{G}\,\psi, u.j \rangle \in \mathcal{T}(\varphi)$ with $\psi \neq \mathit{true}$, we denote with $\mathit{p\text{-}node}(u.j)$ the parent node $\langle \psi', u \rangle \in \mathcal{T}(\varphi)$. (By the definition of a canonical syntax tree, the formula $\mathbf{G}\,\psi$ alone cannot be the formula of the root node.) Note that the id u always points to either the root node, or a node of the form $\langle \mathbf{F}\,\psi'', u \rangle$ for some formula $\psi'' \in \mathsf{ELTL_{FT}}$. We define the parent cutpoint as follows:

$$\mathit{p\text{-}cutpoint}(w) = \begin{cases} \zeta(u), & \text{when } u \in \mathcal{V}_\mathcal{G}, w = u.j \text{ for some } j \in \mathbb{N}_0, \\ 0, & \text{otherwise.} \end{cases}$$

We prove the following statements about the intermediate tree nodes using structural induction on the tree $\mathcal{T}(\varphi)$:

(i) for a node $\langle \mathbf{G}\,\psi, w \rangle$ with $\psi \neq \mathit{true}$, if $\mathit{p\text{-}cutpoint}(w)$ cuts $\tau \cdot \rho$ into π' and π'', then $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\psi$.

(ii) for a node $\langle \mathbf{F}\,\psi, w \rangle$, if $\zeta(w)$ cuts $\tau \cdot \rho$ into π' and π'', then $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi$.

Based on this we finally prove

(iii) for the root node $\langle can(\varphi), 0\rangle \in \mathcal{T}(\varphi)$, it holds that $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models can(\varphi)$.

which establishes the theorem.

Proving (i). Fix a tree node $\langle \mathbf{G}\,\psi, w\rangle$ with $\psi \neq \mathit{true}$. Let $p\text{-}cutpoint(w)$ cut $\tau \cdot \rho$ into π' and π''. We have to show that $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\psi$. Since $\psi \neq \mathit{true}$, by the definition of a canonical formula, ψ has the form $\psi_0 \wedge \mathbf{F}\,\psi_1 \dots \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\mathit{true}$ for some $k \geq 0$, a propositional formula ψ_0 and canonical formulas $\psi_1, \dots, \psi_k$. It is sufficient to show that: (a) $\pi'(\sigma) \models \mathbf{G}\,\psi_0$, and (b) $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\mathbf{F}\,\psi_i$ for $1 \leq i \leq k$.

To show (a), we consider three cases:

1. Case: $p\text{-}node(w)$ is the root and φ is not of the form $\mathbf{F}\,(\dots)$. Then $can(\varphi) = \dots \wedge \mathbf{G}\,\psi$, and by Condition ***(C1)*** of Definition 4.5, we have $\mathsf{Cfgs}(\sigma, \tau \cdot \rho) \models prop(\psi)$. As $\tau \cdot \rho^\omega$ is a lasso, i.e., $(\tau \cdot \rho^k(\sigma)) = \tau(\sigma)$ for $k \geq 0$, we have that $\sigma \models \mathbf{G}\,prop(\psi)$. From this, and $prop(\psi) = \psi_0$, we conclude that $\pi'(\sigma) \models \mathbf{G}\,\psi_0$, as $p\text{-}cutpoint(w) = 0$ and thus π' is the empty schedule.

2. Case: $p\text{-}node(w) = \langle \mathbf{F}\,\psi'', u\rangle$ for some $\psi'' \in \mathsf{ELTL}_{\mathsf{FT}}$ and $u \in \mathbb{N}_0^\omega$, and $\zeta(u) < |\tau|$. In this case, $\psi'' = \dots \wedge \mathbf{G}\,\psi$. By Condition ***(C2)*** of Definition 4.5, $\mathsf{Cfgs}(\pi'(\sigma), \pi'') \models prop(\psi)$. By noticing $prop(\psi) = \psi_0$ and applying Lemma 4.38, we have $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\psi_0$.

3. Case: $p\text{-}node(w) = \langle \mathbf{F}\,\psi'', u\rangle$ for some $\psi'' \in \mathsf{ELTL}_{\mathsf{FT}}$ and $u \in \mathbb{N}_0^\omega$, and $|\tau| \leq \zeta(u) < |\tau| + |\rho|$. In this case, $\psi'' = \dots \wedge \mathbf{G}\,\psi$. By Condition ***(C3)*** of Definition 4.5, $\mathsf{Cfgs}(\tau(\sigma), \rho) \models prop(\psi)$. By noticing $prop(\psi) = \psi_0$ and applying Lemma 4.39, we arrive at $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\psi_0$.

To show (b), we fix an index $i : 1 \leq i \leq k$ and prove $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\mathbf{F}\,\psi_i$. Let w_i be the node id of the ψ's subformula $\mathbf{F}\,\psi_i$ in the syntax tree $\mathcal{T}(\varphi)$. Note that $\langle \mathbf{F}\,\psi_i, w_i\rangle$ is covered by a $\mathbf{G}$-node, since it is created from a subformula of $\mathbf{G}\,\psi$. Thus, $(\mathsf{loop}_{\mathsf{start}}, w_i) \in \mathcal{E}_{\mathcal{G}}$, and by the definition of the cut function ζ, we have $\zeta(w_i) \geq \zeta(\mathsf{loop}_{\mathsf{start}}) \geq |\tau|$. Let $\zeta(w_i)$ cut $\tau \cdot \rho$ in $\tau \cdot \beta'$ and β''. By the inductive hypothesis, Point (ii) holds for the tree node w_i, and thus $\mathsf{path}((\tau \cdot \beta')(\sigma), \beta'' \cdot \rho^\omega) \models \psi_i$ holds. Since $\tau \cdot \rho^\omega$ is a lasso-shaped schedule, we have $\tau(\sigma) = (\tau \cdot \rho^j)(\sigma)$ for $j \geq 0$, that is, the state $\tau(\sigma)$ occurs infinitely often in the path $\mathsf{path}((\tau \cdot \beta')(\sigma), \beta'' \cdot \rho^\omega)$. Hence, we arrive at:

$$\mathsf{path}((\tau \cdot \beta')(\sigma), \beta'' \cdot \rho^\omega) \models \mathbf{G}\,\mathbf{F}\,\psi_i, \text{ for } 1 \leq i \leq k.$$

From (a) and (b), and the standard LTL property $(\mathbf{G}\,A) \wedge (\mathbf{G}\,B) \Rightarrow \mathbf{G}\,(A \wedge B)$, Point (i) follows for the tree node $\langle \mathbf{G}\,\psi, w\rangle$.

Proving (ii). Fix a tree node $\langle \mathbf{F}\,\psi, w\rangle$, and let $\zeta(w)$ cut $\tau \cdot \rho$ into π' and π''. We have to show that $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi$ holds.

By the definition of a canonical formula, ψ has the form $\psi_0 \wedge \mathbf{F}\,\psi_1 \wedge \dots \wedge \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$ for some $k \geq 0$, a propositional formula ψ_0, canonical formulas $\psi_1, \dots, \psi_k$, and a formula ψ_{k+1} that is either a canonical formula, or equals *true*. We will show that: (a) $\pi'(\sigma) \models \psi_0$, and (b) $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\,\psi_{k+1}$, and (c) $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{F}\,\psi_i$ for $1 \leq i \leq k$. From (a)–(c), the required statement immediately follows.

To show (a), we notice that there are two cases: $\zeta(w) < |\tau|$, or $\zeta(w) \geq |\tau|$. In these cases, either Assumption (***(C2)***) or Assumption (***(C3)***) implies that $\pi'(\sigma) \models \psi_0$.

To show (b), we focus on the case $\psi_{k+1} \neq \mathit{true}$, as the case $\psi_{k+1} = \mathit{true}$ is trivial. Notice that by the definition of the syntax tree $\mathcal{T}(\varphi)$, the subformula $\mathbf{G}\,\psi_{k+1}$ has the id $w.j$ for $j = k+1$, and thus $\textit{p-cutpoint}(w.j) = \zeta(w)$. Thus, ($b$) follows directly from the inductive hypothesis (i), which has already been shown to hold for the tree node $\langle \mathbf{G}\,\psi_{k+1}, w.j\rangle$.

To show (c), fix an index $i \in \{1, \dots, k\}$. Let $\zeta(w.i)$ cut $\tau \cdot \rho$ into β' and β''. The inductive hypothesis (ii) has been shown to hold for the tree node $\langle \mathbf{F}\,\psi_i, w.i\rangle \in \mathcal{T}(\varphi)$, and thus we have:

$$\mathsf{path}(\beta'(\sigma), \beta'' \cdot \rho^\omega) \models \psi_i. \tag{4.5}$$

We consider three cases, based on whether w and $w.i$ are covered by a $\mathbf{G}$-node:

1. Case: neither w, nor $w.i$ is covered by a $\mathbf{G}$-node. By the definition of the cut graph, $(w, w.i) \in \mathcal{E}_\mathcal{G}$, and thus by the definition of the cut function ζ, it holds that $\zeta(w) \leq \zeta(w.i)$. From this and Equation (4.5), it follows that $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{F}\,\psi_i$.

2. Case: $w.i$ is covered by a $\mathbf{G}$-node. By the definition of the cut graph, the node $w.i$ has to be inside the loop: $(\mathsf{loop}_{\mathsf{start}}, w.i) \in \mathcal{E}_\mathcal{G}$. Consequently, by the definition of the cut function ζ, it holds that $\zeta(w.i) \geq |\tau|$. Let β'_l be the suffix of β' inside the loop, i.e., $\beta' = \tau \cdot \beta'_l$. As $\tau \cdot \rho^\omega$ is a lasso-shaped schedule, we have $(\tau \cdot \rho \cdot \beta'_l)(\sigma) = \beta'(\sigma)$. Consequently, we can advance one iteration of the loop and derive the following from Equation (4.5):

$$\mathsf{path}((\tau \cdot \rho \cdot \beta'_l)(\sigma), \beta'' \cdot \rho^\omega) \models \psi_i \tag{4.6}$$

   Notice that in Equation (4.6) we use $\tau \cdot \rho \cdot \beta''_l$, not $\tau \cdot \rho$. The definition of ζ requires that $\zeta(w) < |\tau| + |\rho|$. Since $|\tau \cdot \rho| \leq |\tau \cdot \rho \cdot \beta'_l|$, we have $\zeta(w) \leq |\tau \cdot \rho \cdot \beta'_l|$, that is, the formula ψ_i is satisfied at the state $(\tau \cdot \rho \cdot \beta'_l)(\sigma)$ that either coincides with the state $\pi'(\sigma)$ or occurs after the state $\pi'(\sigma)$ in the path $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega)$. From this and Equation (4.6), we conclude that $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{F}\,\psi_i$ holds.

3. Case: w is covered by a $\mathbf{G}$-node, while $w.i$ is not. This case is impossible, since the node with id $w.i$ is the child of the node with id w in the syntax tree $\mathcal{T}(\varphi)$.

From (a)–(c), Point (ii) follows for the tree node $\langle \mathbf{F}\,\psi, w\rangle$.

Proving (iii). Let $can(\varphi) \equiv \psi_0 \wedge \mathsf{F}\,\psi_1 \wedge \dots \wedge \mathsf{F}\,\psi_k \wedge \mathsf{G}\,\psi_{k+1}$, for some $k \geq 0$, a propositional formula ψ_0, canonical formulas $\psi_1, \dots, \psi_k$, and a formula ψ_{k+1} that is either a canonical formula, or equals to *true*. We have to show $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models can(\varphi)$. To this end, we will show that: (a) $\sigma \models \psi_0$, and (b) $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \mathsf{G}\,\psi_{k+1}$, and (c) $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \mathsf{F}\,\psi_i$ for $1 \leq i \leq k$. From (a)–(c), the required statement immediately follows.

Point (a) follows directly from Condition ***(C1)*** of Definition 4.5.

To show (b), we focus on the case $\psi_{k+1} \neq \mathit{true}$, as the case $\psi_{k+1} = \mathit{true}$ is trivial. Notice that by the definition of the syntax tree $\mathcal{T}(\varphi)$, the subformula $\mathsf{G}\,\psi_{k+1}$ has the id $0.j$ for $j = k+1$, and thus $p\text{-}cutpoint(0.j) = 0$, which cuts $\tau \cdot \rho$ into the empty schedule and $\tau \cdot \rho$ itself. Thus, (b) follows directly from the inductive hypothesis (i), which has already been shown to hold for the tree node $\langle \mathsf{G}\,\psi_{k+1}, 0.j\rangle$.

To show (c), fix an index $i \in \{1, \dots, k\}$. Let $\zeta(0.i)$ cut $\tau \cdot \rho$ into β' and β''. The inductive hypothesis (ii) has been shown to hold for the tree node $\langle \mathsf{F}\,\psi_i, 0.i\rangle \in \mathcal{T}(\varphi)$, and thus we have:

$$\mathsf{path}(\beta'(\sigma), \beta'' \cdot \rho^\omega) \models \psi_i. \tag{4.7}$$

Since $\mathsf{path}(\beta'(\sigma), \beta'' \cdot \rho^\omega)$ is a suffix of $\mathsf{path}(\sigma, \tau \cdot \rho^\omega)$, from Equation (4.7), we immediately obtain the required statement: $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \mathsf{F}\,\psi_i$.

By collecting Points (a)–(c), we immediately arrive at: $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models can(\varphi)$. By Definition 4.5, the formula φ is equivalent to $can(\varphi)$. This finishes the proof. □

Theorem 4.8. *Let φ be an $\mathsf{ELTL}_{\mathsf{FT}}$ formula, σ be a configuration and $\tau \cdot \rho^\omega$ be a lasso applicable to σ such that $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$ holds. There is a witness of $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho^\omega) \models \varphi$ for some $K \geq 0$.*

Proof. We apply Proposition 4.6 to find the required number $K \geq 0$ and the cut function ζ. It remains to show that Conditions ***(C1)–(C3)*** of Definition 4.5 are satisfied for the configuration σ and the lasso $(\tau \cdot \rho^K) \cdot \rho^\omega$.

Showing Condition *(C1)*. This condition does not depend on the structure of ζ. Let $can(\varphi) = \psi_0 \wedge \mathsf{F}\,\psi_1 \wedge \dots \mathsf{F}\,\psi_k \wedge \mathsf{G}\,\psi_{k+1}$. Since $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho^\omega) \models \varphi$, we immediately have $\sigma \models \psi_0$ and $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho^\omega) \models \mathsf{G}\,\psi_{k+1}$. By the semantics of LTL, the latter implies that for all configurations σ' visited by the path $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho^\omega)$, it holds that $\sigma' \models prop(\psi_{k+1})$. Since $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho)$ is a subsequence of $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho^\omega)$, we immediately arrive at $\mathsf{Cfgs}(\sigma, (\tau \cdot \rho^K) \cdot \rho) \models prop(\psi_{k+1})$.

Showing Conditions *(C2)* and *(C3)*. Let $\psi = \psi_0 \wedge \mathsf{F}\,\psi_1 \wedge \dots \wedge \mathsf{F}\,\psi_k \wedge \mathsf{G}\,\psi_{k+1}$. Further, assume that $\zeta(v)$ cuts $(\tau \cdot \rho^K) \cdot \rho$ into π' and π''. By Proposition 4.6, we have

$\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi$. Thus, we have the following:

$$\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi_0 \tag{4.8}$$

$$\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \mathbf{G}\, \mathit{prop}(\psi_{k+1}) \tag{4.9}$$

It remains to prove the specific statements about ***(C2)*** and ***(C3)***:

1. Case $\zeta(v) < |\tau \cdot \rho^K|$. We have to show Condition ***(C2)***.

   The path $\mathsf{path}(\pi'(\sigma), \pi'')$ is a subsequence of $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega)$. Thus, from Equation (4.9), we obtain that for every configuration σ' visited by the finite path $\mathsf{path}(\pi'(\sigma), \pi'')$, it holds that $\sigma' \models \mathit{prop}(\psi_{k+1})$. In other words:

   $$\mathsf{Cfgs}(\pi'(\sigma), \pi'') \models \mathit{prop}(\psi_{k+1}) \tag{4.10}$$

   Equations (4.8) and (4.10) give us Condition ***(C2)***.

2. Case $\zeta(v) \geq |\tau \cdot \rho^K|$. We have to show Condition ***(C3)***. In this case, $\pi' = (\tau \cdot \rho^K) \cdot \pi'_l$ for some schedule π'_l.

   Consider the configuration $\sigma' = (\tau \cdot \rho^K \cdot \rho \cdot \pi'_l)(\sigma)$, that is, σ' is the result of applying to σ the prefix $\tau \cdot \rho^K$, one iteration of the loop ρ, and then the first part of the loop π'_l. The configuration σ' is located at the cut point $\zeta(v)$ in the loop, and the path $\mathsf{path}(\sigma', \pi'' \cdot \pi'_l)$ reaches the same configuration again, i.e., $(\pi'' \cdot \pi'_l)(\sigma') = \sigma'$. From Equation (4.9), we have that the propositional formula $\mathit{prop}(\psi_{k+1})$ holds on the path $\mathsf{path}(\sigma', \pi'' \cdot \pi'_l)$. Since both paths $\mathsf{path}(\sigma', \pi'' \cdot \pi'_l)$ and $\mathsf{path}((\tau \cdot \rho^K)(\sigma), \rho)$ visit all configurations of the loop, we have:

   $$\mathsf{Cfgs}((\tau \cdot \rho^K)(\sigma), \rho) \models \mathbf{G}\, \mathit{prop}(\psi_{k+1}) \tag{4.11}$$

   Equations (4.8) and (4.11) give us Condition ***(C3)***.

The theorem follows. □

4.6 Detailed Proofs for Section 4.3

Let us first present some useful properties of a decomposition.

Proposition 4.12. *If σ is a configuration, τ is a steady conventional schedule applicable to σ, then there exists a decomposition of σ and τ.*

Proof. We have to prove the two properties of Definition 4.8. We do so by induction on the length of τ.

- $|\tau| = 1$. Let $\tau = t_1$ for a transition t_1, and let η be the identity function. Then, $\tau|_{\eta,1} = t_1$ is a thread and for all $i > 1$, the sequence $\tau|_{\eta,i}$ is empty. As τ is applicable to σ, $\sigma.\boldsymbol{\kappa}[t_1.\mathit{from}] \geq 1$.

- $|\tau| > 1$. Let $\tau = \tau' \cdot t_{|\tau|}$, and let η' be a decomposition of σ and τ', which exists by the induction hypothesis. We distinguish two cases for $T = \{i \colon i \in \Theta(\sigma, \tau', \eta') \wedge \tau'|_{\eta',i}.\mathit{to} = t_{|\tau|}.\mathit{from}\}$:

  - If $T \neq \emptyset$, then for some $j \in T$ let

  $$\eta(k) = \begin{cases} j & \text{if } k = |\tau| \\ \eta'(k) & \text{otherwise,} \end{cases}$$

  that is, we append transition $t_{|\tau|}$ to thread j. Therefore, $\Theta(\sigma, \tau, \eta) = \Theta(\sigma, \tau', \eta')$ and consequently for all $\ell \in \mathcal{L}$ we have $\{i \colon i \in \Theta(\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.\mathit{from} = \ell\} = \{i \colon i \in \Theta(\sigma, \tau', \eta') \wedge \tau'|_{\eta',i}.\mathit{from} = \ell\}$. Hence, it follows from the induction hypothesis that for all $\ell \in \mathcal{L}$, $\sigma.\boldsymbol{\kappa}[\ell] \geq |\{i \colon i \in \Theta(\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.\mathit{from} = \ell\}|$.

  - If $T = \emptyset$, then for some $j \notin \Theta(\sigma, \tau', \eta')$ let

  $$\eta(k) = \begin{cases} j & \text{if } k = |\tau| \\ \eta'(k) & \text{otherwise,} \end{cases}$$

  that is, we add a new thread consisting of $t_{|\tau|}$ only. From applicability of τ to σ follows that $\tau'(\sigma).\boldsymbol{\kappa}[t_{|\tau|}.\mathit{from}] \geq 1$. Now from Proposition 4.11 follows that

  $$\sigma.\boldsymbol{\kappa}[t_{|\tau|}.\mathit{from}] \geq 1 - |T| + |\{i \colon i \in \Theta(\sigma, \tau', \eta') \wedge \tau'|_{\eta',i}.\mathit{from} = t_{|\tau|}.\mathit{from}\}|.$$

  As $|T| = 0$ in this case and since by construction $|\{i \colon i \in \Theta(\sigma, \tau', \eta') \wedge \tau'|_{\eta',i}.\mathit{from} = t_{|\tau|}.\mathit{from}\}| = |\{i \colon i \in \Theta(\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.\mathit{from} = t_{|\tau|}.\mathit{from}\}| - 1$, we obtain that $\sigma.\boldsymbol{\kappa}[t_{|\tau|}.\mathit{from}] \geq |\{i \colon i \in \Theta(\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.\mathit{from} = t_{|\tau|}.\mathit{from}\}|$ as required. For the other components of $\sigma.\boldsymbol{\kappa}$, the proposition follows from the induction hypothesis as $\Theta(\sigma, \tau, \eta) = \Theta(\sigma, \tau', \eta') \cup \{j\}$.

□

Our goal is to prove Proposition 4.16, that allows us to move a whole thread in a schedule, while still reaching the same configuration as the original schedule. For that we first need to prove in the following two propositions, that we are allowed to swap one transition of a thread, with transitions of different threads.

Proposition 4.13. *If σ is a configuration, $\tau = \tau_1 \cdot t_{i-1} \cdot t_i \cdot \tau_2$ is a steady schedule applicable to σ, η is a decomposition of σ and τ, and $\eta(i-1) \neq \eta(i)$, then $\tau_1(\sigma).\boldsymbol{\kappa}[t_i.\mathit{from}] \geq 1$.*

Proof. From Proposition 4.11 it follows that

$$\tau_1(\sigma).\boldsymbol{\kappa}[t_i.\mathit{from}] = \sigma.\boldsymbol{\kappa}[t_i.\mathit{from}] + |\{k \colon \tau_1|_{\eta,k}.\mathit{to} = t_i.\mathit{from}\}| - |\{k \colon \tau_1|_{\eta,k}.\mathit{from} = t_i.\mathit{from}\}|.$$

We distinguish two cases:

- If $t_i = \tau|_{\eta,\eta(i)}[1]$, that is, if t_i is the first transition in a thread, then we have that $|\{k \colon \tau|_{\eta,k}.\mathit{from} = t_i.\mathit{from}\}| > |\{k \colon \tau_1|_{\eta,k}.\mathit{from} = t_i.\mathit{from}\}|$. By assumption, $\sigma.\boldsymbol{\kappa}[t_i.\mathit{from}] \geq |\{k \colon \tau|_{\eta,k}.\mathit{from} = t_i.\mathit{from}\}|$. Thus, $\tau_1(\sigma).\boldsymbol{\kappa}[t_i.\mathit{from}] > |\{k \colon \tau_1|_{\eta,k}.\mathit{to} = t_i.\mathit{from}\}| \geq 0$, which proves the proposition in this case.
- Otherwise, by Definition 4.8 (2), we have that $\sigma.\boldsymbol{\kappa}[t_i.\mathit{from}] - |\{k \colon \tau_1|_{\eta,k}.\mathit{from} = t_i.\mathit{from}\}| \geq 0$. Therefore, it holds that $\tau_1(\sigma).\boldsymbol{\kappa}[t_i.\mathit{from}] \geq |\{k \colon \tau_1|_{\eta,k}.\mathit{to} = t_i.\mathit{from}\}|$. As τ_1 contains the prefix of $\tau|_{\eta,\eta(i)}$, we find that $|\{k \colon \tau_1|_{\eta,k}.\mathit{to} = t_i.\mathit{from}\}| \geq 1$ such that $\tau_1(\sigma).\boldsymbol{\kappa}[t_i.\mathit{from}] \geq 1$ as required.

□

Proposition 4.14. *Let σ be a configuration, let τ be a steady schedule applicable to σ, and let η be a decomposition of σ and τ. For every $i \in \mathbb{N}$, if $1 < i \leq |\tau|$ and $\eta(i-1) \neq \eta(i)$, then the following holds:*

1. *$\tau_{i\leftarrow}$ is a steady schedule applicable to σ,*
2. *$\eta_{i\leftarrow}$ is a decomposition of σ and $\tau_{i\leftarrow}$, and $\tau_{i\leftarrow}|_{\eta_{i\leftarrow},j} = \tau|_{\eta,j}$, for every $j \in \Theta(\sigma, \tau, \eta)$,*
3. *$\tau_{i\leftarrow}(\sigma) = \tau(\sigma)$.*

Proof. (1) To prove this we have to show that (1a) $\tau[i]$ is applicable to $\tau^{i-2}(\sigma)$, and that (1b) $\tau[i-1]$ is applicable to $\tau^{i-2} \cdot \tau[i](\sigma)$. Point (1) then follows from commutativity of addition and subtraction on the counters.

1a Since τ is a steady schedule, then it suffices to show that $\tau^{i-2}(\sigma).\boldsymbol{\kappa}[\tau[i].\mathit{from}] \geq 1$, which follows from Proposition 4.13.

1b If $\tau[i].\mathit{from} \neq \tau[i-1].\mathit{from}$, then $\tau^{i-2} \cdot \tau[i](\sigma).\boldsymbol{\kappa}[\tau[i-1].\mathit{from}] \geq \tau^{i-2}(\sigma).\boldsymbol{\kappa}[\tau[i-1].\mathit{from}]$ and the statement follows from applicability of τ to σ. Otherwise, from applicability of τ to σ for the case $\tau[i-1].\mathit{from} = \tau[i-1].\mathit{to}$ it follows that $\tau^{i-2}(\sigma).\boldsymbol{\kappa}[\tau[i-1].\mathit{from}] \geq 1$, and for $\tau[i-1].\mathit{from} \neq \tau[i-1].\mathit{to}$ it follows that $\tau^{i-2}(\sigma).\boldsymbol{\kappa}[\tau[i-1].\mathit{from}] \geq 2$. In both cases the statement follows.

(2) We firstly show that every transition from $\tau_{i\leftarrow}|_{\eta_{i\leftarrow},j}$ is also in $\tau|_{\eta,j}$. Let $\tau_{i\leftarrow}[k]$ be a transition from $\tau_{i\leftarrow}|_{\eta_{i\leftarrow},j}$. Thus, $\eta_{i\leftarrow}(k) = j$. We want to show that $\tau_{i\leftarrow}[k]$ is also in $\tau|_{\eta,j}$. We consider three cases:

- If $k = i-1$, then $\tau_{i\leftarrow}[k] = \tau_{i\leftarrow}[i-1] = \tau[i]$ and $\eta(i) = \eta_{i\leftarrow}(i-1) = \eta_{i\leftarrow}(k) = j$. As $\eta(i) = j$, then $\tau[i]$ belongs to $\tau|_{\eta,j}$. Now $\tau[i] = \tau_{i\leftarrow}[k]$ gives the required.
- If $k = i$, then $\tau_{i\leftarrow}[k] = \tau_{i\leftarrow}[i] = \tau[i-1]$ and $\eta(i-1) = \eta_{i\leftarrow}(i) = \eta_{i\leftarrow}(k) = j$. As $\eta(i-1) = j$, then $\tau[i-1] = \tau_{i\leftarrow}[k]$ belongs to $\tau|_{\eta,j}$.
- If $k \neq i-1$ and $k \neq i$, then by Definition 4.9 we have $\tau_{i\leftarrow}[k] = \tau[k]$ and $\eta(k) = \eta_{i\leftarrow}(k) = j$. Since $\eta(k) = j$, then $\tau[k]$ is in $\tau|_{\eta,j}$. Now $\tau[k] = \tau_{i\leftarrow}[k]$ gives the required.

Proving that every transition from $\tau|_{\eta,j}$ is also in $\tau_{i\leftarrow}|_{\eta_{i\leftarrow},j}$, is analogous to the previous direction.

Now we know that for every $j \in \Theta(\sigma,\tau,\eta)$, schedules $\tau_{i\leftarrow}|_{\eta_{i\leftarrow},j}$ and $\tau|_{\eta,j}$ contain same transitions. The order of these transitions remains the same, since the only two transitions with different positions in τ and $\tau_{i\leftarrow}$ are adjacent transitions from two different threads.

Now, knowing that η is a decomposition of σ and τ, and that all threads remain the same, we conclude that $\eta_{i\leftarrow}$ is a decomposition of σ and $\tau_{i\leftarrow}$.

(3) Follows from the step (2) and Proposition 4.11. □

Proposition 4.16. *Let σ be a configuration, let τ be a steady schedule applicable to σ, and let η be a decomposition of σ and τ. Fix an $i \in \Theta(\sigma,\tau,\eta)$. Let us denote*

$$\tau^* = \tau' \cdot \tau|_{\eta,i} \cdot \tau'',$$

such that τ' is a possibly empty prefix of τ which contains no transitions from $\tau|_{\eta,i}$, and $\tau' \cdot \tau'' = \tau|_{\eta,\mathbb{N}\setminus\{i\}}$. Then we have the following:

1. *τ^* is a steady schedule applicable to σ,*
2. *there exists a decomposition η^* of σ and τ^* such that $\tau^*|_{\eta^*,l} = \tau|_{\eta,l}$, for every $l \in \Theta(\sigma,\tau,\eta)$.*
3. *$\tau^*(\sigma) = \tau(\sigma)$.*

Proof. Let us firstly enumerate all transitions from $\tau|_{\eta,i}$, for example, let $\tau|_{\eta,i} = t_{n_1}, t_{n_2}, \ldots, t_{n_k}$, for $1 \leq n_1 < n_2 < \cdots < n_k \leq |\tau|$. Thus, $\tau' = t_1, \ldots, t_s$, for $0 \leq s < n_1$. The idea is that we move transitions from $\tau|_{\eta,i}$, one by one, to the left, namely t_{n_1} to the place $(s+1)$ in τ, then t_{n_2} to the place $s+2$, and so on, by repeatedly applying Proposition 4.14, that preserves the required properties. Formally, $\tau^* = (\ldots((\tau_{s+1\leftarrow n_1})_{s+2\leftarrow n_2})\ldots)_{s+k\leftarrow n_k}$.

For every j with $1 \leq j \leq k$, we denote

$$\tau_j = (\ldots((\tau_{s+1\leftarrow n_1})_{s+2\leftarrow n_2})\ldots)_{s+j\leftarrow n_j} \text{ and}$$

$$\eta_j = (\ldots((\eta_{s+1 \leftarrow n_1})_{s+2 \leftarrow n_2})\ldots)_{s+j \leftarrow n_j}.$$

We prove by induction that for every j, with $1 \leq j \leq k$, it holds that:

a) τ_j is a steady schedule applicable to σ,

b) η_j is a decomposition of σ and τ_j, and $\tau_j|_{\eta_j,l} = \tau|_{\eta,l}$, for every $l \in \Theta(\sigma,\tau,\eta)$,

c) $\tau_j(\sigma) = \tau(\sigma)$.

If $j = 1$, then $\tau_j = \tau_{s+1 \leftarrow n_1}$. Note that $\eta(n_1) \neq \eta(m)$, for every m with $s+1 \leq m < n_1$, since t_{n_1} is the first transition in $\tau|_{\eta,i}$, or in other words, the smallest number mapped to i by η. Now, as $1 \leq s+1 \leq n_1 \leq |\tau|$, the required holds by Proposition 4.15.

Assume that the statement holds for j, and let us show that then it holds for $j+1$ as well. Note that $\tau_{j+1} = (\tau_j)_{(s+j+1) \leftarrow n_{(j+1)}}$. We show that we can apply Proposition 4.15 to σ, τ_j, η_j, $s+j+1$ and n_{j+1}. By induction hypothesis, τ_j is a steady schedule applicable to σ, and η_j is a decomposition of σ and τ_j. From the assumption that $1 \leq s+1 \leq n_1 < n_2 < \cdots < n_k \leq |\tau|$, follows that $1 \leq s+j+1 \leq n_{j+1} \leq |\tau|$. By construction, τ_j has a form $\tau' \cdot t_{n_1} \cdot t_{n_2} \cdot \ldots \cdot t_{n_j} \cdot \rho_1 \cdot t_{n_{j+1}} \cdot \rho_2$, where $\rho_1 \cdot \rho_2 = \tau''$. Note that no transition from ρ_1 is in $\tau|_{\eta,i}$, which is, by induction hypothesis, same as $\tau_j|_{\eta_j,i}$. Thus, $\eta_j(n_{j+1}) \neq \eta_j(m)$, for every m with $s+j+1 < m \leq n_{j+1}$. Now we can apply Proposition 4.15, and obtain the required. □

4.6.1 Detailed Proofs for Section 4.3.1

Proposition 4.27. *Let σ be a configuration, let $\tau = t_1,\ldots,t_{|\tau|}$ be a nonempty steady conventional schedule applicable to σ, and let η be a decomposition of σ and τ. Fix a set Locs of local states. If there is no local state $\ell \in$ Locs such that $\mathsf{Cfgs}(\sigma,\tau) \models \boldsymbol{\kappa}[\ell] \neq 0$, but it holds that $\mathsf{Cfgs}(\sigma,\tau) \models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$, then at least one of the following cases is true:*

1. *There is at least one thread of σ and τ, which is of Locs-type A;*

2. *There is a thread of Locs-type B or E, and an additional of Locs-type C or E;*

3. *There is a thread of Locs-type E, and one of Locs-type D.*

Proof. Firstly, if $|\Theta(\sigma,\tau,\eta)| = 1$, we prove by contradiction that τ must be of *Locs*-type A. Namely, if we suppose the opposite, we distinguish three cases:

- If τ is of *Locs*-type C, D or F, then $\sigma \not\models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$, and therefore $\mathsf{Cfgs}(\sigma,\tau) \not\models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$;
- If τ is of *Locs*-type B, then $\tau(\sigma) \not\models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$, and therefore again $\mathsf{Cfgs}(\sigma,\tau) \not\models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$;

- If τ is of *Locs*-type E, and a k, $1 \leq k < |\tau|$, is such that $t_k.to \notin Locs$, then for the prefix τ' of τ of length k holds that $\tau'(\sigma) \not\models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$.

Thus, for all three options we get a contradiction, which tells us that τ cannot be of any other type, and leaves the only remaining option: that τ is of *Locs*-type A. This gives us the case 1.

Otherwise, if $|\Theta(\sigma, \tau, \eta)| \geq 2$, we have two options:

- If one of the threads is of *Locs*-type A, then this is the case 1.
- If there is no thread of *Locs*-type A, we consider two possibilities:
  - There is a thread $\tau|_{\eta,i}$ of *Locs*-type E, for some $i \in \Theta(\sigma, \tau, \eta)$. Then, by definition, there is a $k \in \mathbb{N}$ such that $\eta(k) = i$ and $t_k.to \notin Locs$. Assume by contradiction that we are not is cases 2. nor 3. Then, among the other threads, there are no threads of *Locs*-type A, B, C, D, nor E. In other words, all the other threads are of *Locs*-type F. Then the prefix τ' of τ of length k has the property that $\tau'(\sigma) \not\models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$. This is a contradiction with the assumption that $\mathsf{Cfgs}(\sigma, \tau) \models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$.
  - There is no thread of *Locs*-type E. Since $\sigma \models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$, there exists an $\ell' \in Locs$ such that $\sigma \models \boldsymbol{\kappa}[\ell'] \neq 0$. From the assumption that $\mathsf{Cfgs}(\sigma, \tau) \not\models \boldsymbol{\kappa}[\ell'] \neq 0$, we obtain that there must exist a thread ϑ_1 with $\text{first}(\vartheta_1) = \ell' \in Locs$. Since in this case there are no threads of *Locs*-type A nor E, this implies that ϑ_1 is of *Locs*-type B.

    Similarly, since $\tau(\sigma) \models \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$, there exists an $\ell'' \in Locs$ such that $\tau(\sigma) \models \boldsymbol{\kappa}[\ell''] \neq 0$. Now, from the assumption that $\mathsf{Cfgs}(\sigma, \tau) \not\models \boldsymbol{\kappa}[\ell''] \neq 0$, we obtain that there exists a thread ϑ_2 with $\text{last}(\vartheta_2) = \ell'' \in Locs$. Thus, ϑ_2 is of *Locs*-type C, and this case is the case 2.

Therefore, at least one of the given cases is true. $\square$

Theorem 4.34. *Fix a threshold automaton* $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$*, and a set* $Locs \subseteq \mathcal{L}$*. Let* σ *be a configuration such that* $\omega(\sigma) = \Omega$*, and let* $\psi \equiv \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$*. Then for every steady conventional schedule* τ*, applicable to* σ*, with* $\mathsf{Cfgs}(\sigma, \tau) \models \psi$*, there is a steady schedule* $\mathsf{repr}_\vee[\psi, \sigma, \tau]$ *with the properties:*

a) $\mathsf{repr}_\vee[\psi, \sigma, \tau]$ *is applicable to* σ*, and* $\mathsf{repr}_\vee[\psi, \sigma, \tau](\sigma) = \tau(\sigma)$*,*

b) $|\mathsf{repr}_\vee[\psi, \sigma, \tau]| \leq 6 \cdot |\mathcal{R}|$*,*

c) $\mathsf{Cfgs}(\sigma, \mathsf{repr}_\vee[\psi, \sigma, \tau]) \models \psi$*,* *d) there exist τ_1, τ_2 and τ_3, (not necessarily nonempty) subschedules of τ, such that $\tau_1 \cdot \tau_2 \cdot \tau_3$ is applicable to σ, it holds that $\tau_1 \cdot \tau_2 \cdot \tau_3(\sigma) = \tau(\sigma)$, and*

$$\mathsf{repr}_{\vee}[\psi, \sigma, \tau] = \mathsf{srep}[\sigma, \tau_1] \cdot \mathsf{srep}[\tau_1(\sigma), \tau_2] \cdot \mathsf{srep}[\tau_1 \cdot \tau_2(\sigma), \tau_3].$$

Proof. We give a constructive proof, and therefore τ_1, τ_2, τ_3 and its properties will be obvious from the construction.

If there is a local state $\ell^* \in \mathit{Locs}$ such that $\mathsf{Cfgs}(\sigma, \tau) \models \boldsymbol{\kappa}[\ell^*] \neq 0$, by Proposition 4.31 we have $\mathsf{Cfgs}(\sigma, \mathsf{srep}[\sigma, \tau]) \models \psi$. Using properties of $\mathsf{srep}[\sigma, \tau]$ described in Proposition 4.2, we see that the required schedule is

$$\mathsf{repr}_{\vee}[\psi, \sigma, \tau] = \mathsf{srep}[\sigma, \tau].$$

If this is not the case, and η is a decomposition of σ and τ, then, since $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, by Proposition 4.27 at least one of the following cases is true:

(1) Assume there is an $i \in \Theta(\sigma, \tau, \eta)$ such that $\tau|_{\eta,i}$ is of *Locs*-type A. We claim that the required schedule is

$$\mathsf{repr}_{\vee}[\psi, \sigma, \tau] = \mathsf{srep}[\sigma, \tau|_{\eta,i}] \cdot \mathsf{srep}[\tau|_{\eta,i}(\sigma), \tau|_{\eta,\mathbb{N}\setminus\{i\}}].$$

By Proposition 4.17, $\tau|_{\eta,i} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i\}}$ is a steady schedule applicable to σ, and $\tau|_{\eta,i} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i\}}(\sigma) = \tau(\sigma)$. Therefore, we can apply Proposition 4.32 to obtain *a*) and *b*). Since $\tau|_{\eta,i}$ is a thread of σ and τ, of *Locs*-type A, and by Proposition 4.28 there is an $\ell^* \in \mathit{Locs}$ such that $\mathsf{Cfgs}(\tau|_{\eta,i}(\sigma), \tau|_{\eta,\mathbb{N}\setminus\{i\}}) \models \boldsymbol{\kappa}[\ell^*] \neq 0$, then *c*) holds by Proposition 4.33.

(2) Here we assume there exist $i, j \in \Theta(\sigma, \tau, \eta)$ such that $i \neq j$, $\tau|_{\eta,j}$ is of *Locs*-type B or E, and $\tau|_{\eta,i}$ is of *Locs*-type C or E. We show that the required schedule is

$$\mathsf{repr}_{\vee}[\psi, \sigma, \tau] = \mathsf{srep}[\sigma, \tau|_{\eta,j}] \cdot \mathsf{srep}[\tau|_{\eta,j}(\sigma), \tau|_{\eta,\mathbb{N}\setminus\{j\}}].$$

Again, by Proposition 4.17, $\tau|_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{j\}}$ is a steady schedule applicable to σ, and $\tau|_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{j\}}(\sigma) = \tau(\sigma)$. Therefore, we can apply Proposition 4.32 to obtain *a*) and *b*). By Proposition 4.29, there exist $\ell_1, \ell_2 \in \mathit{Locs}$ such that $\mathsf{Cfgs}(\sigma, \tau|_{\eta,j}) \models \boldsymbol{\kappa}[\ell_1] \neq 0$ and $\mathsf{Cfgs}(\tau|_{\eta,j}(\sigma), \tau|_{\eta,\mathbb{N}\setminus\{j\}}) \models \boldsymbol{\kappa}[\ell_2] \neq 0$. Thus, *c*) holds by Proposition 4.33.

(3) For the last case we assume there exist $i, j \in \Theta(\sigma, \tau, \eta)$ such that $\tau|_{\eta,i}$ is of *Locs*-type E, and $\tau|_{\eta,j}$ is of *Locs*-type D. We represent $\tau|_{\eta,j}$ as $\tau|^1_{\eta,j} \cdot \tau|^2_{\eta,j}$, where $\mathrm{last}(\tau|^1_{\eta,j}) \subseteq \mathit{Locs}$. With a similar idea as in the previous cases, we show that the required schedule is

$$\begin{aligned}\mathsf{repr}_{\vee}[\psi, \sigma, \tau] \quad = \quad & \mathsf{srep}[\sigma, \tau|^1_{\eta,j}] \cdot \\ & \cdot \mathsf{srep}[\tau|^1_{\eta,j}(\sigma), \tau|_{\eta,i}] \cdot \\ & \cdot \mathsf{srep}[\tau|^1_{\eta,j} \cdot \tau|_{\eta,i}(\sigma), \tau|^2_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i,j\}}].\end{aligned}$$

Again, statements *a*) and *b*) follow from Proposition 4.18 and Proposition 4.32. By Proposition 4.30, there exist $\ell_1, \ell_2, \ell_3 \in \mathit{Locs}$ such that

– $\mathsf{Cfgs}(\sigma, \tau|^1_{\eta,j}) \models \boldsymbol{\kappa}[\ell_1] \neq 0$,
– $\mathsf{Cfgs}(\tau|^1_{\eta,j}(\sigma), \tau|_{\eta,i}) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, and
– $\mathsf{Cfgs}(\tau|^1_{\eta,j} \cdot \tau|_{\eta,i}(\sigma), \tau|^2_{\eta,j} \cdot \tau|_{\eta,\mathbb{N}\setminus\{i,j\}}) \models \boldsymbol{\kappa}[\ell_3] \neq 0$.

Using these three facts and Proposition 4.33, we obtain *c*).

□

4.6.2 Detailed Proofs for Section 4.3.2

Theorem 4.4. *Fix a threshold automaton $\mathit{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$ that has a finite multiplier μ, and a configuration σ. For an $n \in \mathbb{N}$, fix sets of locations $\mathit{Locs}_m \subseteq \mathcal{L}$ for $1 \leq m \leq n$. If we have*

$$\psi = \bigwedge_{1 \leq m \leq n} \bigvee_{\ell \in \mathit{Locs}_m} \boldsymbol{\kappa}[\ell] \neq 0,$$

then for every steady conventional schedule τ, applicable to σ, with $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, there exists a schedule $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ with the following properties:

a) The representative is applicable and ends in the same final state: $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ is a steady schedule applicable to $\mu\sigma$, and $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau](\mu\sigma) = \mu\tau(\mu\sigma)$,

b) The representative has bounded length: $|\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]| \leq 4 \cdot |\mathcal{R}|$,

c) The representative maintains the formula ψ, i.e., $\mathsf{Cfgs}(\mu\sigma, \mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]) \models \psi$,

d) The representative is a concatenation of two representative schedules srep *from Proposition 4.2:*

$$\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau] = \mathsf{srep}[\mu\sigma, \tau] \cdot \mathsf{srep}[\tau(\mu\sigma), (\mu - 1)\tau].$$

Proof. We show that the required schedule is

$$\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau] = \mathsf{srep}[\mu\sigma, \tau] \cdot \mathsf{srep}[\tau(\mu\sigma), (\mu - 1)\tau].$$

By the properties of $\mathsf{srep}[\mu\sigma, \tau]$ and $\mathsf{srep}[\tau(\mu\sigma), (\mu - 1)\tau]$ from Proposition 4.2, we see that $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ is a steady schedule applicable to $\mu\sigma$, that $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau](\mu\sigma) = \mu\tau(\mu\sigma)$, and finally $|\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]| \leq 4 \cdot |\mathcal{R}|$. Now it remains just to show that *c*) holds.

For every $m \leq n$, we denote $\bigvee_{\ell \in \mathit{Locs}_m} \boldsymbol{\kappa}[\ell] \neq 0$ by ψ_m. Since $\psi = \bigwedge_{1 \leq m \leq n} \psi_m$, we prove that for every $m \leq n$, holds $\mathsf{Cfgs}(\mu\sigma, \mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]) \models \psi_m$. Let us fix an $m \leq n$. Since $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, it is also true that $\mathsf{Cfgs}(\sigma, \tau) \models \psi_m$. Therefore, we have that

- $\sigma \models \psi_m$, which implies that there exist an $\ell_m^1 \in \mathit{Locs}_m$ with $\sigma.\boldsymbol{\kappa}[\ell_m^1] \geq 1$, and
- $\tau(\sigma) \models \psi_m$, which implies that there is an $\ell_m^2 \in \mathit{Locs}_m$ with $\tau(\sigma).\boldsymbol{\kappa}[\ell_m^2] \geq 1$.

Now we show that:
i) $\mathsf{Cfgs}(\mu\sigma, \tau) \models \boldsymbol{\kappa}[\ell_m^1] \geq 1$, and
ii) $\mathsf{Cfgs}(\tau(\mu\sigma), (\mu-1)\tau) \models \boldsymbol{\kappa}[\ell_m^2] \geq 1$.

i) Let τ' be an arbitrary prefix of τ. From the assumption and Proposition 4.35 (4) we have that $1 \leq \sigma.\boldsymbol{\kappa}[\ell_m^1] < (f\sigma).\boldsymbol{\kappa}[\ell_m^1]$. Then, from Proposition 4.35 (3) we see that $\tau'(\sigma).\boldsymbol{\kappa}[\ell_m^1] < \tau'(\mu\sigma).\boldsymbol{\kappa}[\ell_m^1]$. Now, since τ' is applicable to σ, and therefore it is $\tau'(\sigma).\boldsymbol{\kappa}[\ell_m^1] \geq 0$, we obtain that $\tau'(\mu\sigma).\boldsymbol{\kappa}[\ell_m^1] \geq 1$. Hence, we have $\mathsf{Cfgs}(\mu\sigma, \tau) \models \boldsymbol{\kappa}[\ell_m^1] \geq 1$.

ii) Let us denote $\{i \colon i \in \Theta(\mu\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.\mathit{to} = \ell_m^2\}$ by T, and $\{i \colon i \in \Theta(\mu\sigma, \tau, \eta) \wedge \tau|_{\eta,i}.\mathit{from} = \ell_m^2\}$ by F. By Proposition 4.11 we have that $0 < \tau(\sigma).\boldsymbol{\kappa}[\ell_m^2] = \sigma.\boldsymbol{\kappa}[\ell_m^2] + |T| - |F|$. This implies that $\sigma.\boldsymbol{\kappa}[\ell_m^2] > |F| - |T|$. By Proposition 4.11, we also obtain

$$\begin{aligned} \tau(\mu\sigma).\boldsymbol{\kappa}[\ell_m^2] &= \mu\sigma.\boldsymbol{\kappa}[\ell_m^2] + |T| - |F| = \\ &= (\mu-1)\sigma.\boldsymbol{\kappa}[\ell_m^2] + \sigma.\boldsymbol{\kappa}[\ell_m^2] - (|F| - |T|), \end{aligned}$$

which combined with $\sigma.\boldsymbol{\kappa}[\ell_m^2] > |F| - |T|$ yields

$$(\mu-1)\sigma.\boldsymbol{\kappa}[\ell_m^2] < \tau(\mu\sigma).\boldsymbol{\kappa}[\ell_m^2].$$

Let now τ' be an arbitrary prefix of $(\mu-1)\tau$. Using the fact that τ' is applicable to $(\mu-1)\sigma$, and Proposition 4.35 (3), we obtain that

$$0 \leq \tau'((\mu-1)\sigma).\boldsymbol{\kappa}[\ell_m^2] < \tau'(\tau(\mu\sigma)).\boldsymbol{\kappa}[\ell_m^2].$$

Therefore, we obtain that $\tau'(\tau(\mu\sigma)).\boldsymbol{\kappa}[\ell_m^2] \geq 1$, and hence $\mathsf{Cfgs}(\tau(\mu\sigma), (\mu-1)\tau) \models \boldsymbol{\kappa}[\ell_m^2] \geq 1$.

Now, when the statements i) and ii) are proved, we can apply Proposition 4.33 This gives us that $\mathsf{Cfgs}(\mu\sigma, \mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]) \models \psi_m$, for an arbitrary $m \leq n$, which implies that $c)$ is true, and concludes the proof. □

4.7 Discussion

Although the verification literature predominantly deals with safety, parameterized verification for liveness properties is of growing interest, and has been addressed mostly in the context of programs that solve mutual exclusion or dining philosophers [ABEL12, FKP16, PXZ02, FPPZ06]. These techniques do not apply to fault-tolerant distributed algorithms that have arithmetic conditions on the fraction of faults, threshold guards, and typical specifications that evaluate a global system state.

Our main technical contribution is to combine and extend several important techniques: First, we extend the ideas by Etessami et al. [EVW02] to reason about shapes of infinite executions of lasso shape. These executions are counterexample candidates. Then we extend reductions introduced in [Lip75, EF82] and Chapter 3, to deal with $\mathsf{ELTL}_{\mathsf{FT}}$ formulas. (Techniques that extend Lipton's in other directions can be found in [CL98, Doe77, LS89, EQT09, FFQ05, KVW17].) Our reduction is specific to threshold guards which are typical for fault-tolerant distributed algorithms and are found in domain-specific languages. Besides using this particular reduction, we apply acceleration [BFLP08, KVW17] in order to arrive at our short counterexample property.

Our short counterexample property implies a completeness threshold, that is, a bound b that ensures that if no lasso of length up to b is satisfying an $\mathsf{ELTL}_{\mathsf{FT}}$ formula, then there is no infinite path satisfying this formula. For linear temporal logic with the $\mathbf{F}$ and $\mathbf{G}$ operators, Kroening et al. [KOS$^+$11] prove bounds on the completeness thresholds on the level of Büchi automata. Their bound involves the recurrence diameter of the transition systems, which is prohibitively large for counter systems. Similarly, the general method to transfer liveness with fairness to safety checking by Biere et al. [BAS02] leads to an exponential growth of the diameter, and thus to too large values of b. Hence, we decided to conduct an analysis on the level of threshold automata, accelerated counter systems, and a fragment of linear temporal logic, which allows us to exploit specifics of the domain, and get bounds that can be used in practice.

Acceleration has been applied for parameterized verification by means of regular model checking [PS00, BHV04, ABJ98, SB06]. As noted by Fisman et al. [FKL08], to verify fault-tolerant distributed algorithms, one would have to intersect the regular languages that describe sets of states with context-free languages that enforce the resilience condition (e.g., $n > 3t$). Our approach of reducing to SMT handles resilience conditions naturally in linear integer arithmetic.

The restrictions we put on threshold automata are justified from a practical viewpoint of our application domain, namely, threshold-guarded fault-tolerant algorithms. We assumed in Section 2.1 that all the cycles in threshold automata are simple (while the benchmarks have only self-loops or cycles of length 2). As our analysis already is quite involved, these restrictions allow us to concentrate on our central results without obfuscating the notation and theoretical results. Still, from a theoretical viewpoint it might be interesting to relax the restrictions on cycles in the future.

CHAPTER 5

Synthesis of Parameterized Threshold Guards

Synthesis is a technique for automated construction of a system, given its input-output relation in a form of a temporal formula [Chu63, PR89]. Verification and synthesis both assure correctness of systems, but while verification requires manual code writing, synthesis automatically derives systems that are correct by construction. The drawback of synthesis is its complexity [PR89]. For fault-tolerant distributed algorithms, this problem is particularly hard [DF09]. Moreover, if we require construction of a system that satisfies the given formula for any system size, that is, in the parameterized case, this problem is in general undecidable [JB14].

In the previous two chapters we have developed a technique for automatically checking safety and liveness properties of threshold-based fault-tolerant distributed algorithms, for any number of processes. As input it receives a threshold automaton as a model for an algorithm, an $\mathsf{ELTL}_{\mathsf{FT}}$ formula describing specifications, and a resilience condition defining the maximum of faulty processes, e.g., less than a third of all processes. The method either confirms that the algorithm satisfies the specifications under the resilience condition, or it produces a counterexample.

In this chapter we use our verification results as a building stone for synthesis. The user just provides required properties, a sketch of an asynchronous algorithm, and a resilience condition, and our tool automatically finds a correct distributed algorithm, or it reports that it does not exist for the given input. In this way we generate new fault-tolerant algorithms that are *correct by construction*. In our experiments we first focus on existing specifications [ST87b, CT96, WS07, SvR08] from the literature, in order to be able to compare the output of our tool with known algorithms. We then give new variations of safety and liveness specifications, and our tool generates new distributed algorithms for them.

```
Code of a correct process $i$:
var $myval_i \in \{0,1\}$
var $accept_i \in \{false, true\} \leftarrow false$

while $true$ do (in one step)
  if $myval_i = 1$ and not sent ECHO before
  then send ECHO to all

  if received ECHO from $\geq \tau_{\mathsf{0toSE}}$ distinct processes
    and not sent ECHO before
  then send ECHO to all

  if received ECHO from $\geq \tau_{\mathsf{AC}}$ distinct processes
  then $accept_i \leftarrow true$
od
```

Figure 5.1: A single-round version of the reliable broadcast algorithm [ST87b] with holes.

More precisely, we address the following challenge:

Challenge 5.1 (Parameterized synthesis of threshold guards)**.** *Given a skeleton of a threshold automaton (that is, a threshold automaton with undefined thresholds) and an* $\mathsf{ELTL}_{\mathsf{FT}}$ *formula* $\neg\psi$ *(that is,* ψ *is a specification), find the thresholds that together with the given skeleton form a threshold automaton* TA *with* $\mathsf{Sys}(\mathsf{TA}) \models \psi$*, for all values of parameters under the given resilience condition.*

We start by demonstrating the essence of our method for solving this challenge in Section 5.1, including a concrete example. Next, in Section 5.2 we precisely define new notions, like sketch threshold automata. We accurately describe out synthesis method in Section 5.3, but the extensive technical proofs are left for Section 5.5. Finally, our case studies and experimental evaluation of our results are presented in Section 5.4.

5.1 Our approach at a glance

Similar to the verification approaches in the previous chapters, we are interested in the parameterized version of the problem: Rather than synthesizing a distributed algorithm that consists of, say, four processes and tolerates one fault, our goal is to synthesize an algorithm that works for n processes, out of which t may fail, for all values of n and t that satisfy a resilience condition, e.g., $n > 3t$. However, the parameterized synthesis problem is in general undecidable [JB14]. As in the parameterized verification approach in the previous two chapters, we will therefore limit ourselves to a specific class of distributed algorithms, namely, to threshold-guarded distributed algorithms. Recall that thresholds are arithmetic expressions over parameters, e.g., $n/2$, and determine how many messages processes should wait for (a majority in the example of $n/2$).

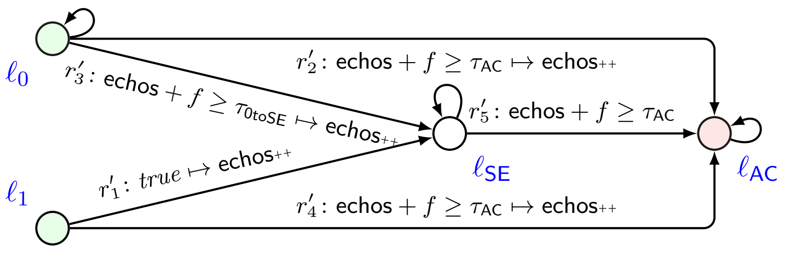

r'_2: echos + $f \geq \tau_{AC} \mapsto$ echos++
r'_3: echos + $f \geq \tau_{0toSE} \mapsto$ echos++
r'_5: echos + $f \geq \tau_{AC}$
r'_1: $true \mapsto$ echos++
r'_4: echos + $f \geq \tau_{AC} \mapsto$ echos++
ℓ_0
ℓ_1
ℓ_{SE}
ℓ_{AC}

Figure 5.2: A sketch threshold automaton

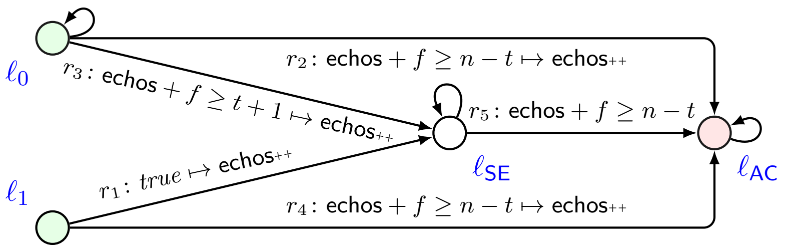

r_2: echos + $f \geq n - t \mapsto$ echos++
r_3: echos + $f \geq t + 1 \mapsto$ echos++
r_5: echos + $f \geq n - t$
r_1: $true \mapsto$ echos++
r_4: echos + $f \geq n - t \mapsto$ echos++
ℓ_0
ℓ_1
ℓ_{SE}
ℓ_{AC}

Figure 5.3: A synthesized threshold automaton

More specifically, the user provides as input a distributed algorithm with holes as in Figure 5.1: The user defines the control flow, and keeps the threshold expressions — noted as τ_{0toSE} and τ_{AC} in the figure — unspecified. Similarly as before, as pseudo code has no formal semantics, it cannot be used as a tool input. Rather, our tool takes as input a sketch threshold automaton.

Example 5.1. Figure 5.1 is a pseudo code representation of the input, and Figure 5.2 shows the corresponding sketch threshold automaton. Relation between them follows the relation between a fully defined algorithm, e.g., the one from Figure 1.1, and its threshold automaton from Figure 2.1. One can even notice that the sketch algorithm from Figure 5.1 is inspired by the algorithm from Figure 1.1.

The "holes" τ_{0toSE} and τ_{AC} in Figure 5.2 are the missing thresholds, which should be linear combination of the parameters n and t. Therefore τ_{0toSE} has the form $\mathbf{?}_1 \cdot n + \mathbf{?}_2 \cdot t + \mathbf{?}_3$, and τ_{AC} has the form $\mathbf{?}_4 \cdot n + \mathbf{?}_5 \cdot t + \mathbf{?}_6$. The unknown coefficients $\mathbf{?}_i$, for $1 \leq i \leq 6$, have to be found by the synthesis tool. Given a sketch threshold automaton from Figure 5.2, the three specifications from Section 1.1, and a resilience condition $n > 3t$, one synthesized solution is $\mathbf{?}_1 = 0$, $\mathbf{?}_2 = 1$, $\mathbf{?}_3 = 1$, $\mathbf{?}_4 = 1$, $\mathbf{?}_5 = -1$, and $\mathbf{?}_6 = 0$, that is, $\tau_{0toSE} = t + 1$ and $\tau_{AC} = n - t$. This solution is depicted in Figure 5.3, and it coincides with the threshold automaton from Figure 2.1. ◁

In addition to a sketch threshold automaton, the user has to provide a specification, that is, safety and liveness properties the distributed algorithm should satisfy (see Section 2.4). Based on these inputs, our tool generates the required coefficients, that is, a threshold automaton as in Figure 5.3. The synthesis approach of this chapter is enabled by the

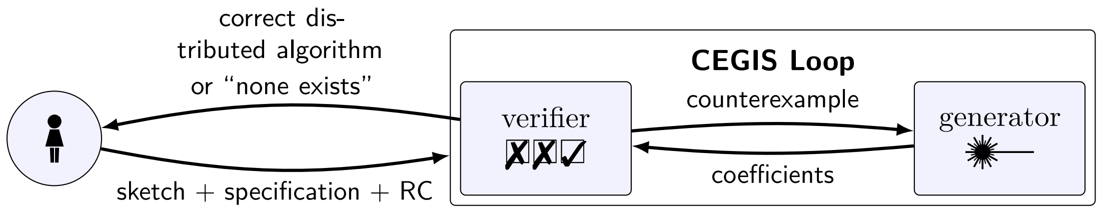

correct distributed algorithm or "none exists"
sketch + specification + RC
CEGIS Loop
verifier
✗✗✓
counterexample
coefficients
generator

Figure 5.4: The synthesis loop describing our approach.

advance in parameterized model checking of *safety and liveness* properties of distributed algorithms, presented in Chapter 4.

As shown in the previous two chapters, resilience conditions, executions of threshold-guarded distributed algorithms, and specifications can be encoded as logical formulas, whose satisfiability can be checked by solvers such as Z3 [MB08] and CVC4 [BCD+11]. In particular, the queries used in Chapter 3 and Chapter 4 correspond to counterexamples to a specification: If the SMT solver finds all queries to be unsatisfiable, the distributed algorithm is correct. Otherwise, if a query is satisfiable, the SMT solver outputs a satisfying assignment, that is an error trace, i.e., a counterexample.

Our synthesis approach. Figure 5.4 gives an overview of our method that takes as input (i) a sketch of a distributed algorithm, (ii) a set of safety and liveness specifications, and (iii) a resilience condition, e.g., $n > 3t$, and produces as output a correct distributed algorithm, or informs the user that none exist.

We follow the CEGIS approach to synthesis [ABJ+13], which proceeds in a refinement loop. Roughly speaking, the verifier starts by picking default values for the missing coefficients — e.g., a vector of zeroes — and checks whether the algorithm is correct with these coefficients. Typically this is not the case and the *verifier* produces a counterexample. By automatically analyzing this counterexample, the *generator* learns constraints on the coefficients that are known to produce counterexamples. The generator gives these constraints to an SMT solver that generates new values for the coefficients, which are used in a new verifier run. If the verifier eventually reports that the current coefficients induce a correct distributed algorithm, we output this algorithm. The theory from Chapter 4 then implies correctness of the algorithm.

Termination of synthesis. The central theoretical problem that we address in this chapter is termination of the refinement loop: In principle, the generator can produce infinitely many vectors of coefficients. In case there is no solution (which is typically the case in Byzantine fault tolerance if $n \leq 3t$), the naïve approach from the previous paragraph does not terminate, unless we restrict the guards to "reasonable" values. In this work, we require the guards to lie in the interval $[0, n]$. We call such guards *sane*. For instance, although syntactically the expressions $\mathsf{echos} \leq -42n$ and $\mathsf{echos} > 2n$ are

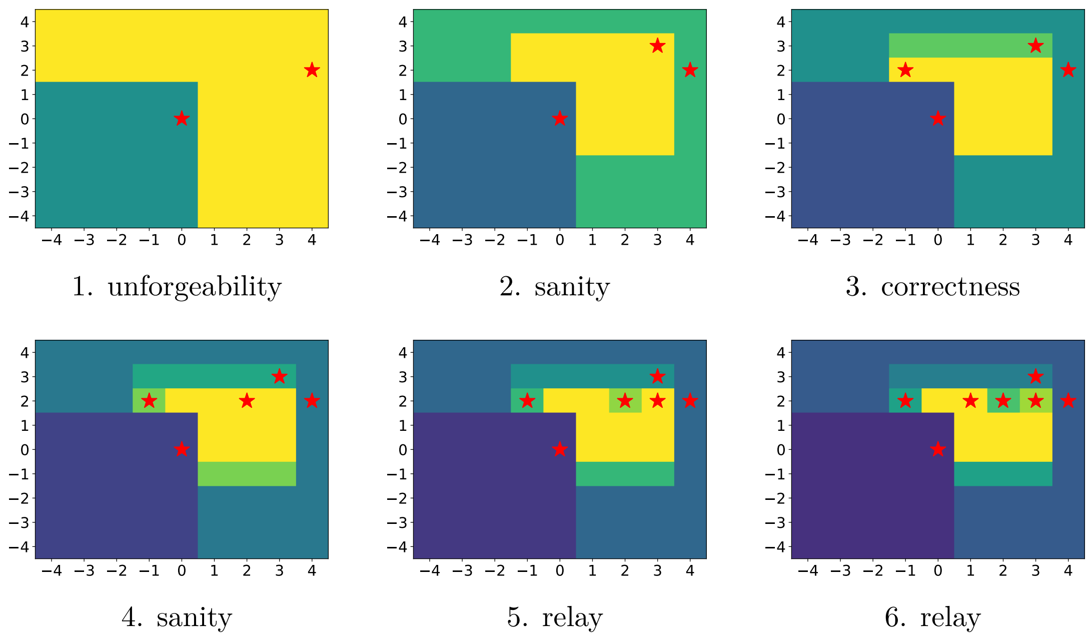


Figure 5.5: Reducing the search space for every loop iteration in Example 5.2. Bellow each plot we indicate the property whose violation led to the search space reduction.

threshold guards, they are not sane, while echos $\geq t+1$ is sane. We mathematically prove that all sane guards of a specific structure have coefficients within a hyperrectangle. We call this hypperrectangle a *sanity box*, and prove that its boundaries depend only on the resilience condition. Within the sanity box, there is only a finite number of coefficients, if we restrict them to integers or rationals with a fixed denominator. We thus obtain a finite search space and a completeness result for the synthesis loop.

Safety, liveness, and the fraction of faults. We consider the conjunction of *safety* and *liveness* specifications, as these specifications in isolation typically have trivial solutions; e.g., "do nothing" is always safe. If just given a safety specification, our tool generates thresholds like n for all guards, which leads to all guards evaluating to false initially. Hence, no action can ever be taken, which is a valid solution if liveness is not required.

Besides, our tool treats resilience conditions precisely. On one hand, given the sketch from Figure 5.2, and the resilience condition $n > 3t$, in a few seconds our tool generates the threshold automaton in Figure 5.3. On the other hand, in the case of $n \geq 3t$, our tool reports (also within seconds) that no such algorithm exists, which in fact constitutes an automatically generated impossibility result for sane thresholds and a fixed sketch.

Example 5.2. Let us consider the synthesis problem with the following input: (i) the sketch threshold automaton from Figure 5.2, with simplified sketch threshold guards

$$\tau_{\mathsf{0toSE}} = \mathbf{?}_2 \cdot t + 1 \text{ and } \tau_{\mathsf{AC}} = \mathbf{?}_5 \cdot t + 1,$$

(ii) specifications unforgeability, correctness and relay, as defined in Section 1.1 or recalled in Section 5.4.1, and (iii) resilience condition $n > 3t$. In other words, for simplicity, we already assume that $\mathbf{?}_1 = \mathbf{?}_4 = 0$ and $\mathbf{?}_3 = \mathbf{?}_6 = 1$. This allows us to graphically represent this reasoning in Figure 5.5.

A way to compute an over-approximation of the sanity box will be discussed and given in Theorem 5.1. In this case, we have that both $\mathbf{?}_2$ and $\mathbf{?}_5$ belong to the interval $[-4, 4]$. As the cardinality of this interval is $|[-4, 4]| = 9$, we have $9 \cdot 9 = 81$ possibilities for the values of the vector $(\mathbf{?}_2, \mathbf{?}_5)$. In order to check whether each of these options yields a correct algorithm, we would need to call the verification procedure 81 times. For more sophisticated benchmarks this number is much larger, and verification can be costly. In this example we show how to reduce the number of calls to the verifier, and explore all possibilities by calling the verification procedure only 6 times instead of 81.

We start the procedure by guessing that $\mathbf{?}_2 = \mathbf{?}_5 = 0$, and check if this is a correct guess. As expected, our model checker finds an error trace that contradicts unforgeability, when parameter values are $(n, t, f) = (4, 1, 1)$. Thus, there is a path in the system with 4 processes and one of them is faulty, where all correct processes start in ℓ_0, but eventually a correct process enters ℓ_{AC}. Recall that in the case of Byzantine faults, we model (correct) $n - f = 3$ processes.

Namely, the path that violates unforgeability is

$$\pi = \sigma_0, r_3^1, \sigma_1, r_5^1, \sigma_2,$$

and here we discuss configurations σ_0, σ_1 and σ_2. The path starts in a configuration σ_0 with all 3 processes in ℓ_0, and all other locations are empty. In this configuration, the threshold guard of the rule r_3, that is, $\mathsf{echos} + f \geq \mathbf{?}_2 \cdot t + 1$, is true as $0 + 1 \geq 0 \cdot 1 + 1$. This allows us to move one process along the r_3 edge. The obtained configuration σ_1 has $\boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 1$ and $\mathsf{echos} = 1$. This makes the rule r_5 applicable in σ_1, as its threshold $\mathsf{echos} + f \geq \mathbf{?}_5 \cdot t + 1$ is true by $1 + 1 \geq 0 \cdot 1 + 1$. Hence, after applying r_5 to σ_1, one correct process enters location ℓ_{AC} in σ_2, which violates unforgeability.

We want the generator to learn from this counterexample π. The questions we ask are the following: Which other guesses could lead to the same counterexample? For which values of $\mathbf{?}_2$ and $\mathbf{?}_5$ would r_3 still be applicable to σ_0, and r_5 applicable to σ_1? For such values, the same path $\pi = \sigma_0, r_3^1, \sigma_1, r_5^1, \sigma_2$ would be a legal path in the system with $(n, t, f) = (4, 1, 1)$. Thus, the system would again violate unforgeability.

Therefore, we want to exclude all the values of $\mathbf{?}_2$ and $\mathbf{?}_5$ for which the guard of r_3 is true in σ_0, and the guard of r_5 is true in σ_1, as they do not yield a correct parameterized system. Formally, we check for which $\mathbf{?}_2$ and $\mathbf{?}_5$ we have that $0 + 1 \geq \mathbf{?}_2 \cdot 1 + 1$ and $1 + 1 \geq \mathbf{?}_5 \cdot 1 + 1$. This excludes all the values from the region $[-4, 0] \times [-4, 1]$, that is represented in a turquoise color on the top left plot from Figure 5.5.

The unexplored region now is the yellow region, and we pick another guess from it. For example, let $\mathbf{?}_2 = 4$ and $\mathbf{?}_5 = 2$. These values yield thresholds that are not sane. For instance, τ_{0toSE} becomes $4 \cdot t + 1$. For $n = 4$ and $t = 1$ (the resilience condition is satisfies), we have that the value of τ_{0toSE} does not lie in the interval $[0, n]$, since $4 \cdot t + 1 = 5 > 4 = n$. We exclude all the vectors $(\mathbf{?}_2, \mathbf{?}_5)$ for which $0 \le \mathbf{?}_2 \cdot t + 1 \le n$ or $0 \le \mathbf{?}_5 \cdot t + 1 \le n$ can be violated, under the resilience condition $n > 3t$ and $t \ge f$. This is the green area in the top middle plot of Figure 5.5.

The yellow area is not explored, so we pick another guess and call the verification procedure. In this way, after 6 loop iterations in which we restrict the search space (colored in yellow), we discover a correct solution, namely $\mathbf{?}_2 = 1$ and $\mathbf{?}_5 = 2$. $\triangleleft$

5.2 Sketch Threshold Automata

Threshold-guarded algorithms are formalized by threshold automata. We recall the notions of threshold automata from Section 2.1 and introduce the new concept of sketches. As usual, $\mathbb{N}_0$ is the set of natural numbers including 0, and $\mathbb{Q}$ is the set of rational numbers. The set Π is a finite set of *parameter variables* that range over $\mathbb{N}_0$. Typically, Π consists of three variables: n for the total number of processes, f for the number of actual faults in a run, and t for an upper bound on f. The parameter variables from Π are restricted to admissible combinations by a resilience condition, e.g., $n > 3t \wedge t \ge f \ge 0$. The set Γ is a finite set that contains shared variables that store the number of distinct messages sent by distinct (correct) processes, the variables in Γ also range over $\mathbb{N}_0$. In the example in Figure 5.3, $\Gamma = \{\mathsf{echos}\}$. For the variables from Γ, we will use names echos, x, y, etc.

For a set of variables V, a function $\nu : V \to \mathbb{Q}$ is called *an assignment*; its domain V is denoted with $dom(\nu)$. In this chapter, we use Φ, Ψ, and Θ for first-order logic (FOL) formulas; e.g., when encoding linear integer constraints in SMT. For a FOL formula Φ, we write $free(\Phi)$ for the set of free variables of Φ, that is, the variables not bounded with a quantifier. (For convenience, we assume that quantified variables have unique names and they are different from the names of the free variables.) Given an assignment $\nu : V \to \mathbb{Q}$ and a FOL formula Φ, we define a *substitution* $\Phi[\nu]$ as a FOL formula that is obtained from Φ by replacing all the variables from $V \cap free(\Phi)$ with their values in ν.

To introduce sketches of threshold automata — such as in Figure 5.2 — we define unknowns such as $\mathbf{?}_1$. The set U is a finite set of *unknowns* that range over $\mathbb{Q}$. We denote the variables from U with $\mathbf{?}_1$, $\mathbf{?}_2$, etc. The rational values of unknowns are denoted by a, b, c, etc.

Sketch threshold guards. In Section 2.1 we have introduced threshold guards, but here we need to extend this notion in order to be able to express sketches of the guards. *Generalized threshold guards* — in this chapter for convenience often called just *guards* — are defined according to the grammar:

$\mathit{Guard} ::= \mathit{Shared} \geq \mathit{LinForm} \mid \mathit{Shared} < \mathit{LinForm}$	$\mathit{Shared} ::= \langle\text{variable from } \Gamma\rangle$
$\mathit{LinForm} ::= \mathit{FreeCoeff} \mid \mathit{Prod} \mid \mathit{Prod} + \mathit{LinForm}$	$\mathit{Param} ::= \langle\text{a variable from } \Pi\rangle$
$\mathit{FreeCoeff} ::= \mathit{Rat} \mid \mathit{Unknown}$	$\mathit{Unknown} ::= \langle\text{a variable from } U\rangle$
$\mathit{Prod} ::= \mathit{Rat} \times \mathit{Param} \mid \mathit{Unknown} \times \mathit{Param}$	$\mathit{Rat} ::= \langle\text{a rational from } \mathbb{Q}\rangle$

For convenience, we assume that every parameter appears in *LinForm* at most once. Let $\bar{\pi}$ denote the vector $(\pi_1, \ldots, \pi_{|\Pi|}, 1)$ that contains all the parameter variables from Π in a fixed order as well as number 1 as the last element. Then, every generalized guard can be written in one of the two following forms

$$x \geq \bar{u} \cdot \bar{\pi}^{\mathsf{T}} \quad \text{or} \quad x < \bar{u} \cdot \bar{\pi}^{\mathsf{T}},$$

where x is a shared variable from Γ, and $\bar{u}$ is a vector of elements from $U \cup \mathbb{Q}$. When a parameter does not appear in a generalized guard, its corresponding component in $\bar{u}$ equals zero. We say that a guard is a *sketch guard* if its vector $\bar{u}$ contains a variable from U. A guard that is not a sketch guard is called a *fixed guard*, and it corresponds to what is called just a guard in all the other chapters of this thesis. In other words, the work described in the other chapters is only concerned with fixed guards.

Since threshold guards are a special case of FOL formulas, we can apply substitutions to them. For instance, given an assignment $\nu : U \to \mathbb{Q}$ and a threshold guard g, the substitution $g[\nu]$ replaces every occurence of an unknown $\mathbf{?}_i \in U$ in g with the rational $\nu(\mathbf{?}_i)$.

Sketch threshold automata. In Section 2.1 we have defined a threshold automaton, and denoted it by TA. They are edge-labeled graphs, where vertices are called *locations*, and edges are called *rules*. By the new terminology, rules are labeled by $g \mapsto \texttt{act}$, where expression g is a fixed threshold guard, and the action `act` may increment a shared variable. We define *generalized threshold automata* GTA, in the same way as threshold automata, with the only difference that expressions g in the edge labeling are generalized threshold guards. If all generalized guards in a GTA are fixed, then that GTA is a TA. If at least one of the edges of a GTA is labeled by a sketch guard, then we call this automaton a *sketch threshold automaton*, and we denote it by STA. Given an STA and an assignment $\nu : U \to \mathbb{Q}$, we obtain a (fixed) threshold automaton $\mathsf{STA}[\nu]$ by applying substitution $g[\nu]$ to every sketch guard g in STA.

Counter systems. The definition of counter systems for generalized threshold automata coincides with the definitions from Section 2.2.

With a TA we associate a set of predicates $\mathcal{P}_{\mathsf{TA}}$ that track properties of the system states. The set $\mathcal{P}_{\mathsf{TA}}$ consists of the TA's threshold guards and a test $\boldsymbol{\kappa}[\ell] = 0$ for every location ℓ in TA. For every configuration σ, one can compute the set $\rho(\sigma) \subseteq \mathcal{P}_{\mathsf{TA}}$ of the predicates that hold true in σ. As was demonstrated in Section 2.3, the predicates from

$\mathcal{P}_{\mathsf{TA}}$ and linear temporal logic are sufficient to express the safety and liveness properties of threshold-guarded distributed algorithms found in the literature.

A system execution is expressed as a path in the counter system. Formally, recall that a path is an infinite alternating sequence of configurations and transitions, that is, $\sigma_0, t_1, \sigma_1, \dots, t_i, \sigma_i, \dots$, where σ_0 is an initial configuration, and σ_{i+1} is the result of applying t_{i+1} to σ_i for $i \geq 0$. The infinite sequence $\rho(\sigma_0), \rho(\sigma_1), \dots$ is called the path *trace*. With $\mathsf{Traces}_{\mathsf{TA}}$ we denote the set of all path traces in the TA's counter system. Correctness of a distributed algorithm then means that all traces in $\mathsf{Traces}_{\mathsf{TA}}$ satisfy a specification expressed in linear temporal logic [CGP99]. The verification approach from Chapter 4 discussed in Section 5.3.1 specifically looks for traces that violate the specification. Such traces are characterized by the temporal logic $\mathsf{ELTL}_{\mathsf{FT}}$ that allows one to express *negations of specifications* relevant for fault-tolerant distributed algorithms.

5.3 Synthesizing Thresholds

5.3.1 Verification machinery

In Chapter 3 and Chapter 4 we have introduced a technique for parameterized verification of threshold-based distributed algorithms. Here we recall the core of this technique, with the simplified notation, as it will be necessary for this chapter.

Given a fixed threshold automaton TA, a resilience condition RC, and a set $\{\neg\varphi_1, \dots, \neg\varphi_k\}$ of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas representing negation of specifications, we check whether there is an execution violating the specification $(\varphi_1 \wedge \dots \wedge \varphi_k)$. Thus, as an output, the algorithm from Chapter 4 either confirms correctness, or gives a counterexample. In this chapter, we use this technique as a black box, that is, we assume that there is a function

$$\mathsf{verify}_{\mathsf{ByMC}}(\mathsf{TA}, RC, \{\neg\varphi_1, \dots, \neg\varphi_k\})$$

that either reports a counterexample, or it confirms that the system $\mathsf{Sys}(\mathsf{TA})$ is correct.

5.3.2 Synthesis problem

A temporal logic formula φ in $\mathsf{ELTL}_{\mathsf{FT}}$ describes an (infinite) set of bad traces that the synthesized algorithm must avoid. Therefore, we consider the following formulation of the *synthesis problem.* Given a sketch threshold automaton STA and an (infinite) set of bad traces $\mathsf{Traces}_{\mathsf{Bad}}$, either:

- find an assignment $\mu : U \to \mathbb{Q}$, in order to obtain the fixed threshold automaton $\mathsf{STA}[\mu]$ whose traces $\mathsf{Traces}_{\mathsf{STA}[\mu]}$ *do not intersect* with $\mathsf{Traces}_{\mathsf{Bad}}$, or
- report that no such assignment exists.

Our approach is to find values for the unknowns in a synthesis refinement loop and test them with the verification technique from Section 5.3.1.

```
1  procedure synt_ByMC(STA, RC, $\{\neg\varphi_1, \dots, \neg\varphi_k\}$)
2    $\Theta_0$ := bound$_U$(RC) and i := 0
3    while (true)
4      call check_SMT($\Theta_i$)
5      case unsat ⇒ print 'no more solutions' and exit()
6      case sat($\mu$) ⇒ /* $\mu$ assigns rationals to the variables in $U$ */
7        call verify_ByMC(STA[$\mu$], RC, $\{\neg\varphi_1, \dots, \neg\varphi_k\}$)
8        case correct ⇒
9          print 'solution $\mu$' /* exclude this solution and continue */
10         $\Theta_{i+1} := \Theta_i \wedge \bigvee_{?_j \in U} ?_j \neq \mu[?_j]$ and i := i+1
11       case counterexample($S$, $\nu$) ⇒ /* $dom(\nu) \cap U = \emptyset$ */
12         $S_U$ := generalize($S$, STA)
13         $\Psi$ := formula_SMT($S_U$)
14         $\Theta_{i+1} := \Theta_i \wedge \neg\Psi[\nu]$ and i := i+1
```

Figure 5.6: Pseudo-code of the synthesis loop

5.3.3 Synthesis loop

In Figure 5.6 we show the pseudo-code of the synthesis procedure $\mathsf{synt}_{\mathsf{ByMC}}$. At its input the procedure receives a sketch threshold automaton STA, a resilience condition, and a set of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas $\{\neg\varphi_1, \dots, \neg\varphi_k\}$, which capture the bad traces $\mathsf{Traces}_{\mathsf{Bad}}$. In line 2, formula Θ_0, which captures constraints on the unknowns from U, is initialized using a function $\mathtt{bound}_U$. In principle, $\mathtt{bound}_U$ can be initialized to true (no constraints). However, to ensure termination, we will discuss later in this section, how we obtain constraints that bound the coefficients of sane guards. After initialization we enter the synthesis loop.

The SMT solver checks whether Θ_i has a satisfying assignment to the unknowns in U (line 4). If Θ_i is unsatisfiable, the loop terminates with a *negative outcome* in line 5. Otherwise, the SMT solver gives us an assignment $\mu : U \to \mathbb{Q}$ that is a solution candidate. To check feasibility of μ, the verifier is called for the fixed threshold automaton STA[μ] in line 7. The verifier generates multiple schemas, each being one SMT query, which are checked either sequentially or in parallel. If the verifier reports that a schema that produces a counterexample does not exist, then the candidate assignment μ and threshold automaton STA[μ] give us a solution to the synthesis problem. If we were interested in just one solution, the loop would terminate here with a *positive outcome.* However, because we want to enumerate all solutions, our function does a complete search, such that we exclude the solution μ for the future search in line 10, and continue.

If the verifier finds a counterexample, the loop proceeds with the branch in line 11. A counterexample is a schema S of STA[μ] and a satisfying assignment $\nu : V \to \mathbb{Q}$ to the free variables V of the SMT formula $\mathsf{formula}_{\mathsf{SMT}}$, which include the parameters Π, shared variables x^j for $x \in \Gamma$, and counters $\boldsymbol{\kappa}^j[\ell]$ for each local state $\ell \in \mathcal{L}$ and every configuration j. In principle, we could exclude μ from consideration similar to line 10.

For efficiency, we want to exclude a larger set of evaluations, namely all that lead to the same counterexample: We produce a *generalized* schema S_U, by replacing the rules and guards in S, which belong to the threshold automaton $\mathsf{STA}[\mu]$, with the rules and guards of the sketch threshold automaton STA (line 12). In line 13, we generate a generalized counterexample Ψ. As Ψ is derived from a counterexample with valuations μ and ν, we know that $\Psi[\nu][\mu]$ is true. Further, for every evaluation of the unknowns μ', if $\Psi[\nu][\mu']$ is true, then $\Psi[\nu][\mu']$ is a counterexample. To exclude all these evaluations μ' at once, we conjoin $\neg\Psi[\nu]$ with Θ_i in line 14, which gives us new constraints on the unknowns, before entering the next loop iteration.

The synthesis loop terminates only in line 5, that is, if Θ_i is unsatisfiable. As, in this case, Θ_i is equivalent to false, the following observation guarantees that all satisfying assignments of Θ_0 have been explored and all solutions (if any exists) have been reported.

Observation 3. At the beginning of every iteration $i \geq 0$ of the synthesis loop in lines 3–14, the following invariant holds: if $\mu : U \to \mathbb{Q}$ is a satisfying assignment of formula $\Theta_0 \wedge \neg\Theta_i$, then either: (1) μ was previously reported as a solution in line 9, or (2) μ was previously excluded in line 14 and thus is not a solution.

5.3.4 Completeness and termination for sane guards

Without restricting Θ_0, the search space for coefficients is infinite. In the following, we exploit domain-knowledge on distributed algorithms, and show that restricting the synthesis problem to sane guards bounds the search space.

The role of threshold guards is typically to check whether the number of distinct senders, from which messages are received, reaches a threshold. We also use threshold guards in our models to bound the number of processes that go into a special crash state. In both cases, one counts distinct processes and it is therefore natural to consider only those thresholds whose value is in $[0, n]$. More precisely, if the guard has a form $x \geq \bar{u} \cdot \bar{\pi}^\intercal$ or $x < \bar{u} \cdot \bar{\pi}^\intercal$, then for all parameter values that satisfy resilience condition it holds that $0 \leq \bar{u} \cdot \bar{\pi}^\intercal \leq n$. We call such guards *sane* for a given resilience condition.

Theorem 5.1 considers a general case of hybrid failure models [WS07] where different failure bounds exist for different failure models (e.g., t_1 Byzantine faults and t_2 crash faults), and these failure bounds are related to the number of processes n by a resilience condition of the form $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.\ t_i \geq 0$. We bound the values of the coefficients of sane guards.

Theorem 5.1. *Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, for $1 \leq i \leq k$, and $n, t_1, \dots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq an + (b_1 t_1 + \ldots + b_k t_k) + c \quad \textit{or} \quad x < an + (b_1 t_1 + \ldots + b_k t_k) + c,$$

where $x \in \Gamma$, and $a, b_1, \ldots, b_k, c \in \mathbb{Q}$. If the guard is sane for the resilience condition, then

$$\begin{aligned} 0 \leq\ & a \leq 1, && (5.1)\\ -\delta_i - 1 <\ & b_i < \delta_i + 1, \text{ for all } i = 1, \ldots, k && (5.2)\\ -2(\delta_1 + \ldots + \delta_k) - k - 1 \leq\ & c \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. && (5.3) \end{aligned}$$

The proof of this theorem uses the basics of linear algebra, and it is presented in Section 5.5 in details.

The case when $k = 1$ gives us the classical resilience condition where the system model assumes *one* type of faults (e.g., crash), and the assumed number of faults t is related to the total number of processes n, by a condition $n > \delta t \geq 0$ for some $\delta > 0$. If the guard that compares a shared variable and $an + bt + c$ is sane for the resilience condition, then we obtain that $0 \leq a \leq 1$, $-\delta - 1 < b < \delta + 1$, and $-2\delta - 2 \leq c \leq 2\delta + 2$. Any restriction of the intervals from Theorem 5.1 to finite sets gives us completeness: If we reduce the domain of variables from U to integers, or to rationals with fixed denominator (e.g., $\frac{z}{10}$ for $z \in \mathbb{Z}$), one reduces the search space to a finite set of valuations. All threshold-based distributed algorithms we are aware of, use guards with coefficients that are either integers or rationals with a denominator not greater than 3. Thus, we restrict our intervals by intersecting them with the set of rational numbers whose denominator is at most D, for a given $D \in \mathbb{N}$.

The following corollary is a direct consequence of Theorem 5.1, and it tells us how to modify intervals if the coefficients are rational numbers with a fixed denominator.

Using the fact that $x \leq \frac{\tilde{d}}{D} \leq y$ implies that $Dx \leq \tilde{d} \leq Dy$, for a $D \in \mathbb{N}$, the following statement follows directly from Theorem 5.1.

Corollary 5.2. *Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.\ t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, $1 \leq i \leq k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq \frac{\tilde{a}}{D} n + \left(\frac{\tilde{b_1}}{D} t_1 + \ldots + \frac{\tilde{b_k}}{D} t_k \right) + \frac{\tilde{c}}{D} \quad \text{or} \quad x < \frac{\tilde{a}}{D} n + \left(\frac{\tilde{b_1}}{D} t_1 + \ldots + \frac{\tilde{b_k}}{D} t_k \right) + \frac{\tilde{c}}{D},$$

where $x \in \Gamma$, $\tilde{a}, \tilde{b_1}, \ldots, \tilde{b_k}, \tilde{c} \in \mathbb{Z}$, $D \in \mathbb{N}$. If the guard is sane for the resilience condition then

$$\begin{aligned} 0 \leq\ & \tilde{a} \leq D, && (5.4)\\ D(-\delta_i - 1) <\ & \tilde{b_i} < D(\delta_i + 1), \text{ for all } i = 1, \ldots, k, && (5.5)\\ D(-2(\delta_1 + \ldots + \delta_k) - k - 1) \leq\ & \tilde{c} \leq D(2(\delta_1 + \ldots + \delta_k) + k + 1). && (5.6) \end{aligned}$$

Constraints (5.4)–(5.6) constitute the sanity box that function bound_U computes in Figure 5.6. By fixing D, we restrict Θ_0 to have finitely many satisfying assignments (integers). Hence, the loop terminates.

Because a guard that is sane for a weaker resilience condition, is also sane for a stronger one, Theorem 5.1 and Corollary 5.2 also hold for any resilience condition that follows from this one, e.g., $n > \max\{\delta_1 t_1, \ldots, \delta_k t_k\} \wedge \forall i.\ t_i \geq 0$. We can use the same intervals, confirmed by the same proofs as in Section 5.5. However, our benchmarks use the form of resilience conditions of Theorem 5.1.

Moreover, the statements similar to Theorem 5.1 and Corollary 5.2 can be derived for other forms of threshold guards, e.g., for thresholds with floor or ceiling functions. In Section 5.6 we introduce Theorem 5.4 and Corollary 5.5, that consider floor and ceiling functions. Despite the theoretical value of these statements, our benchmarks do not make use of such thresholds.

5.4 Case Studies and Experiments

Our team has extended the ByMC tool with the synthesis technique presented in this chapter. A virtual machine with the tool and the benchmarks is available from: `http://forsyte.at/software/bymc`.[1] The experiments reported in [LKWB17] are conducted on two systems: a laptop and the Vienna Scientific Cluster (VSC-3). The laptop is equipped with 16 GB of RAM and Intel® Core™ i5-6300U processor with 4 cores, 2.4 GHz. The cluster VSC-3 consists of 2020 nodes, each equipped with 64 GB of RAM and 2 processors (Intel® Xeon™ E5-2650v2, 2.6 GHz, 8 cores) and is internally connected with an Intel QDR-80 dual-link high-speed InfiniBand fabric: `http://vsc.ac.at`.

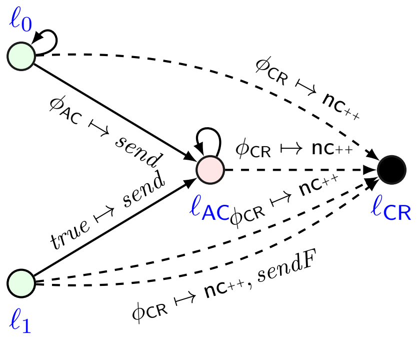
We synthesize thresholds for asynchronous fault-tolerant distributed algorithms. We consider *reliable broadcast* and *fast decision* for a consensus algorithm. In the case of reliable broadcast we consider different fault models, namely, crashes [CT96] and Byzantine faults [ST87b], as well as a hybrid fault model [WS07] with both, Byzantine and crash failures. For fast decision, we consider the one-step consensus algorithm BOSCO for Byzantine faults [SvR08].

5.4.1 Reliable broadcast for crash and/or Byzantine failures

Figure 5.8 shows a sketch threshold automaton of a reliable broadcast that should tolerate $f_c \leq t_c$ crash and $f_b \leq t_b$ Byzantine faults under the resilience condition $n > 3t_b + 2t_c$. For our experiments under simpler failure models — only Byzantine and crash faults — we use the sketch threshold automata from Figures 5.2 and 5.7. However, the same thresholds can be obtained by setting $t_c = f_c = 0$ and $t_b = f_b = 0$ in the automaton from Figure 5.8, respectively. In Figure 5.2, we do not need a dedicated crash state, as we only model correct processes explicitly, while Byzantine faults are modeled via the guards (cf. Example 5.1). The automaton from Figure 5.7 can be obtained from Figure 5.8 by removing the location ℓ_{SE}.

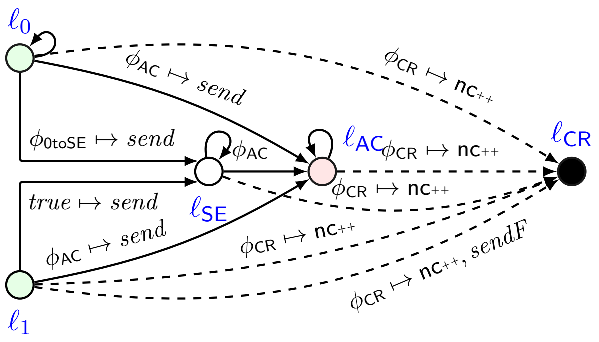
The algorithms we consider are the core of broadcasting algorithms, and establish agreement on whether to accept the message by the broadcaster. Similar to Example 5.1,

[1]See `http://forsyte.at/opodis17-artifact/` for detailed instructions on using the tool.



ℓ_0
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}$
$\phi_{\mathsf{AC}} \mapsto send$
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}$
$true \mapsto send$
ℓ_{AC}
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}$
ℓ_{CR}
ℓ_1
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}, sendF$

Figure 5.7: A sketch threshold automaton for folklore reliable broadcast



ℓ_0
$\phi_{\mathsf{AC}} \mapsto send$
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}$
ℓ_{CR}
$\phi_{\mathsf{0toSE}} \mapsto send$
ϕ_{AC}
ℓ_{AC}
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}$
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}$
$true \mapsto send$
ℓ_{SE}
$\phi_{\mathsf{AC}} \mapsto send$
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}$
$\phi_{\mathsf{CR}} \mapsto \mathsf{nc}_{++}, sendF$
ℓ_1

Figure 5.8: A sketch threshold automaton for reliable broadcast with Byzantine and crash faults

processes start in locations ℓ_1 and ℓ_0, which capture that the process has received and has not received a message by the broadcaster, respectively. A correctly designed algorithm should satisfy the following properties [ST87b], also discussed in Section 1.1:

(U) *Unforgeability:* If no correct process starts in ℓ_1, then no correct process ever enters ℓ_{AC}.

(C) *Correctness:* If all correct processes start in ℓ_1, then there exists a correct process that eventually enters ℓ_{AC}.

(R) *Relay:* Whenever a correct process enters ℓ_{AC}, all correct processes eventually enter ℓ_{AC}.

In the following discussion we use Figure 5.8 as example. We have to sketch the guards ϕ_{CR}, ϕ_{0toSE}, and ϕ_{AC}. At most f_c processes can move to the crashed state ℓ_{CR}. The algorithm designer does not have control over the crashes, and thus we fix the guard ϕ_{CR} to be $\mathsf{nc} < f_c$: The shared variable nc maintains the actual number of crashes (initially zero), which is used only to model crashes and thus cannot be used in guards other than ϕ_{CR}. To properly model that a process can crash during the "send to all" operation (*non-clean crash*), we introduce two shared variables: the variable echos stores the number of echo messages that are sent by the correct processes (some of them may crash later), and the variable $\mathsf{echosCF}$ stores the number of echo messages that are sent by the correct processes and the faulty processes when crashing. Hence, the action *send* increases both echos and $\mathsf{echosCF}$, whereas the action *sendF* increases only $\mathsf{echosCF}$.

We define the thresholds τ_{0toSE} and τ_{AC} as $(\mathbf{?}_a^{\mathsf{SE}} \cdot n + \mathbf{?}_b^{\mathsf{SE}} \cdot t_b + \mathbf{?}_c^{\mathsf{SE}} \cdot t_c + \mathbf{?}_d^{\mathsf{SE}})$ and $(\mathbf{?}_a^{\mathsf{AC}} \cdot n + \mathbf{?}_b^{\mathsf{AC}} \cdot t_b + \mathbf{?}_c^{\mathsf{AC}} \cdot t_c + \mathbf{?}_d^{\mathsf{AC}})$ respectively. Hence, ϕ_{0toSE} and ϕ_{AC} are defined as $\mathsf{echosCF} + f_b \geq \tau_{\mathsf{0toSE}}$ and $\mathsf{echosCF} + f_b \geq \tau_{\mathsf{AC}}$. As discussed in Section 2.1, we add f_b to $\mathsf{echosCF}$ to reflect that the correct processes may — although do not have to — receive

Table 5.1: Synthesized solutions for reliable broadcast that tolerates: crashes (Figure 5.7), Byzantine faults (Figure 5.2), and Byzantine & crash faults (Figure 5.8). We used the laptop in the experiments.

Resilience condition	**Specs**	**#Solutions**	**Threshold τ_{0toSE}**	**Threshold τ_{AC}**	**Calls to verifier**	**Time, seconds**
$n > t_c,\ t_b = 0$	U, C, R	1	*true*	1	12	6
$n > 3t_b,\ t_c = 0$	U, C, R	3	$n - 2t_b$ $t_b + 1$ $t_b + 1$	$n - t_b$ $2t_b + 1$ $n - t_b$	31	16
$n \geq 3t_b,\ t_c = 0$	U, C, R	None	—	—	25	7
$n > 3t_b + 2t_c$	U, C, R	3	$n - 2t_b - 2t_c$ $t_b + 1$ $t_b + 1$	$n - t_b - t_c$ $2t_b + t_c$ $n - t_b - t_c$	34	50
$n \geq 3t_b + 2t_c$	U, C, R	None	—	—	21	12
$n > 3t_b + t_c$	U, C, R	None	—	—	29	24

messages from Byzantine processes. For *reliable communication*, we have to enforce:

Every correct process eventually receives at least echos *number of messages.* (RelComm)

As threshold automata do not explicitly store the number of received messages, we transform (RelComm) into a fairness constraint, which forces processes to eventually leave a location if the messages by correct processes alone enable a guard of an edge that is outgoing from this location. That is, *there is a time after which the following holds forever:*

$$\boldsymbol{\kappa}[\ell_1] = 0 \wedge (\mathsf{echos} < \tau_{\mathsf{0toSE}} \vee \boldsymbol{\kappa}[\ell_0] = 0) \wedge (\mathsf{echos} < \tau_{\mathsf{AC}} \vee (\boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 0)). \quad \text{(Fair)}$$

Table 5.1 summarizes the experimental results from [LKWB17] for reliable broadcast, when looking for integer solutions only. The cases $t_b = 0$ and $t_c = 0$ correspond to the algorithms that tolerate only crashes (Figure 5.7) and only Byzantine faults (Figure 5.2) respectively. For these cases, we obtained the solutions known from the literature [ST87b, CT96] and some variations. Moreover, when the resilience condition is changed from $n > 3t_b$ to $n \geq 3t_b$, our tool reports no solution, which also complies with the literature [ST87b]. In the case of f_c crashes and f_b Byzantine faults, the tool reports three solutions. Moreover, when we tried to relax the resilience condition to $n \geq 3t_b + 2t_c$ and $n > 3t_b + t_c$, the tool reported that there is no solution, as expected.

Table 5.2: Synthesized solutions for variations of reliable broadcast and specifications (X)–(Z).

Resilience condition	Specs	#Solutions	Threshold τ_{0toSE}	Threshold τ_{AC}	Calls to verifier	Time, seconds
$n > 3t_b,\ t_c = 0$	X, C, R	None	—	—	15	2
$n > 3t_b + 2,\ t_c = 0$	X, C, R	3	$n - 2t_b$ $t_b + 3$ $t_b + 3$	$n - t_b$ $2t_b + 3$ $n - t_b$	35	12
$n > 3t_b,\ t_c = 0$	Y, C, R	None	—	—	28	6
$n > 4t_b,\ t_c = 0$	Y, C, R	3	$n - 2t_b$ $2t_b + 1$ $2t_b + 1$	$n - t_b$ $3t_b + 1$ $n - t_b$	33	12
$n > 3t_b + 2t_c$	U, Z, R	2	$t_b + 1$ $t_b + 1$	$n - t_b - t_c$ $2t_b + t_c + 1$	41	31

Variations of the specification. Our logic allows us to easily change the specifications. For instance, we can replace the precondition of unforgeability "if *no* correct process starts in ℓ_1" by giving an upper bound (number or parameter) on correct processes starting in ℓ_1 that still prevents entering ℓ_{AC}, in specifications (X) and (Y). We also changed the precondition of correctnesss "if *all* correct processes start in ℓ_1" in specification (Z):

(X) If *at most two* correct processes start in ℓ_1, then no correct process ever enters ℓ_{AC}.

(Y) If *at most* t_b correct processes start in ℓ_1, then no correct process ever enters ℓ_{AC}.

(Z) If *at least* $t_b + t_c + 1$ non-Byzantine processes (correct or crash faulty) start in ℓ_1, then there exists a correct process that eventually enters ℓ_{AC}.

Interestingly, we obtain new distributed computing problems that put quantitative conditions on the initial state. These specifications are related to the specifications of condition-based consensus [MMPR03]. Our tool automatically generates solutions, or shows their absence in the case resilience conditions are too strong. Table 5.2 summarizes these results.

5.4.2 Byzantine one-step consensus

Figure 5.9 shows a sketch threshold automaton of a one-step Byzantine consensus algorithm that should tolerate $f \leq t$ Byzantine faults under the assumption $n > 3t$. It

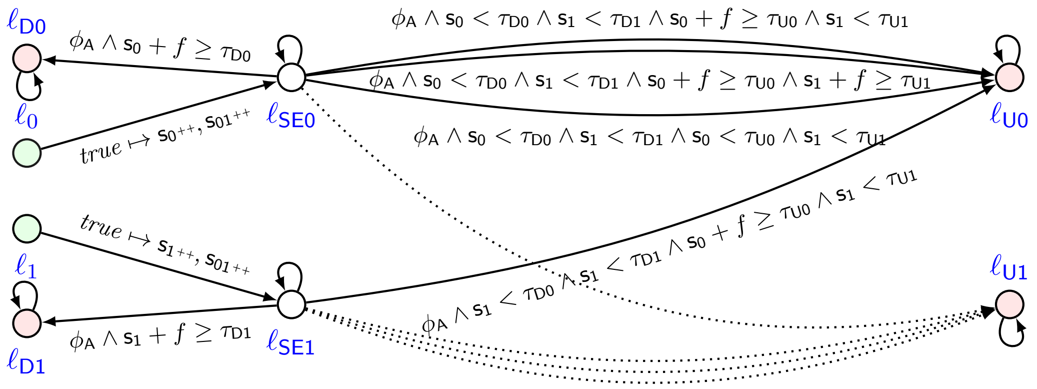

ℓ_{D0}
$\phi_{\mathsf{A}} \wedge \mathsf{s}_0 + f \geq \tau_{\mathsf{D0}}$
ℓ_0
$true \mapsto \mathsf{s}_0{}^{++}, \mathsf{s}_{01}{}^{++}$
ℓ_{SE0}
$\phi_{\mathsf{A}} \wedge \mathsf{s}_0 < \tau_{\mathsf{D0}} \wedge \mathsf{s}_1 < \tau_{\mathsf{D1}} \wedge \mathsf{s}_0 + f \geq \tau_{\mathsf{U0}} \wedge \mathsf{s}_1 < \tau_{\mathsf{U1}}$
$\phi_{\mathsf{A}} \wedge \mathsf{s}_0 < \tau_{\mathsf{D0}} \wedge \mathsf{s}_1 < \tau_{\mathsf{D1}} \wedge \mathsf{s}_0 + f \geq \tau_{\mathsf{U0}} \wedge \mathsf{s}_1 + f \geq \tau_{\mathsf{U1}}$
$\phi_{\mathsf{A}} \wedge \mathsf{s}_0 < \tau_{\mathsf{D0}} \wedge \mathsf{s}_1 < \tau_{\mathsf{D1}} \wedge \mathsf{s}_0 < \tau_{\mathsf{U0}} \wedge \mathsf{s}_1 < \tau_{\mathsf{U1}}$
ℓ_{U0}
$true \mapsto \mathsf{s}_{1++}, \mathsf{s}_{01++}$
ℓ_1
$\phi_{\mathsf{A}} \wedge \mathsf{s}_1 < \tau_{\mathsf{D0}} \wedge \mathsf{s}_1 < \tau_{\mathsf{D1}} \wedge \mathsf{s}_0 + f \geq \tau_{\mathsf{U0}} \wedge \mathsf{s}_1 < \tau_{\mathsf{U1}}$
ℓ_{U1}
ℓ_{D1}
$\phi_{\mathsf{A}} \wedge \mathsf{s}_1 + f \geq \tau_{\mathsf{D1}}$
ℓ_{SE1}

Figure 5.9: A sketch threshold automaton for one-step Byzantine consensus. Labels of dashed edges are omitted; they can be obtained from the respective solid edges by swapping 0 and 1.

is a formalization of the BOSCO algorithm [SvR08]. The purpose of the algorithm is to quickly reach consensus if (a) $n > 5t$ and $f = 0$, or (b) $n > 7t$. In this encoding, correct processes make a "fast" decision on 0 or 1 by going in the locations ℓ_{D0} and ℓ_{D1}, respectively. When neither (a) nor (b) holds, the processes precompute their votes in the first step and then go to the locations ℓ_{U0} and ℓ_{U1}, from which an *underlying consensus* algorithm is taking over. In this sense, BOSCO can be seen as an asynchronous preprocessing step for general consensus algorithms, and the properties given below contain preconditions for calling consensus in a safe way (see Fast Agreement below). Every run of a synthesized threshold automaton must satisfy the following properties (for $i \in \{0, 1\}$ and $j = 1 - i$):

(A) *Fast agreement [SvR08, Lemmas 3–4]:* Condition $\boldsymbol{\kappa}[\ell_{\mathsf{Di}}] \neq 0$ implies $\boldsymbol{\kappa}[\ell_{\mathsf{Dj}}] = \boldsymbol{\kappa}[\ell_{\mathsf{Uj}}] = 0$.

(O) *One step:* If $n > 5t \wedge f = 0$ or $n > 7t$, and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it always holds that $\boldsymbol{\kappa}[\ell_{\mathsf{Dj}}] = 0$ and $\boldsymbol{\kappa}[\ell_{\mathsf{U0}}] = \boldsymbol{\kappa}[\ell_{\mathsf{U1}}] = 0$. That is, the underlying consensus is never called.

(F) *Fast termination:* If $n > 5t \wedge f = 0$ or $n > 7t$, and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

(T) *Termination:* It eventually holds that $\boldsymbol{\kappa}[\ell_0] = \boldsymbol{\kappa}[\ell_1] = 0$ and $\boldsymbol{\kappa}[\ell_{\mathsf{SE0}}] = \boldsymbol{\kappa}[\ell_{\mathsf{SE1}}] = 0$.

We define thresholds τ_{A}, τ_{D0}, τ_{D1}, τ_{U0}, τ_{U1} as $\mathbf{?}^x_a \cdot n + \mathbf{?}^x_b \cdot t + \mathbf{?}^x_c$ for $x \in \{\mathsf{A}, \mathsf{D0}, \mathsf{D1}, \mathsf{U0}, \mathsf{U1}\}$. Then, the guard ϕ_{A} is defined as: $\mathsf{s}_{01} + f \geq \tau_{\mathsf{A}}$. Interestingly, the thresholds appear in different roles in the guards, e.g., $\mathsf{s}_0 + f \geq \tau_{\mathsf{D0}}$ and $\mathsf{s}_0 < \tau_{\mathsf{D0}}$. These cases correspond to

Specs	Nr. of solu-tions	Calls to verifier	Nr. of cores	Time min.
AOFT	4	516	128	39
AOFT	4	432	96	25
AOFT	4	425	64	24
AOFT	4	502	16	44
AOFT	4	440	8	51
AOUT	0	376	8	40
AOVT	0	337	8	33

Table 5.3: Experiments for one-step Byzantine consensus for $n > 3t$ running the parallel verifier at VSC-3

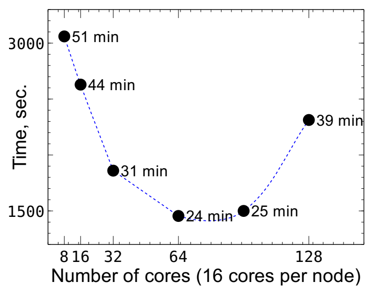


Figure 5.10: Synthesis times for BOSCO at Vienna Scientific Cluster (VSC-3)

BOSCO's decisions on how many messages *have been received* and how many messages *have not been received* "modulo Byzantine faults."

As with reliable broadcast, we model reliable communication with the following fairness constraint: For $i \in \{0, 1\}$, *from some point on*, the following holds: $\boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_1] = 0 \wedge (\mathsf{s}_{01} < \tau_{\mathsf{A}} \vee \mathsf{s}_i < \tau_{\mathsf{Di}} \vee \boldsymbol{\kappa}[\ell_{\mathsf{SEi}}] = 0)$.

We bound *denominators of rationals with two* and use the sanity box provided by Corollary 5.2. To reduce the search space, we assume that the guards for 0 and 1 are *symmetric*, that is $\mathbf{?}_a^{\mathsf{D0}} = \mathbf{?}_a^{\mathsf{D1}}$ and $\mathbf{?}_a^{\mathsf{U0}} = \mathbf{?}_a^{\mathsf{U1}}$. Still, BOSCO is a challenging benchmark both for verification and synthesis. Since the verification procedure from Section 5.3.1 independently checks schemas with SMT, we *parallelized* schema checking with OpenMPI, and ran the experiments at Vienna Scientific Cluster (VSC-3) using 8–128 cores; Table 5.3 summarizes the results. The tool has found four solutions for the guards: $\tau_{\mathsf{A}} = n - t\ [-\frac{1}{2}]$, $\tau_{\mathsf{D0}} = \tau_{\mathsf{D1}} = \frac{n+3t+1}{2}$, and $\tau_{\mathsf{U0}} = \tau_{\mathsf{U1}} = \frac{n-t}{2}\ [+\frac{1}{2}]$. In addition to the guards from [SvR08], the tool also reported that one can add or subtract ½ from several guards. Figure 5.10 demonstrates that increasing the number of cores above 64 slows down synthesis times for this benchmark.

Variations of the BOSCO specifications. We relaxed the precondition for fast termination:

(U) If $n \geq 5t \wedge f = 0$ and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

(V) If $n \geq 7t$ and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

As can be seen from Table 5.3, specifications (U) and (V) have no solutions.

5.5 Detailed Proofs for Section 5.3

In order to prove Theorem 5.1, we first prove its mathematical background.

Lemma 5.3. *Fix a $k \in \mathbb{N}$, and for every $i \in \{1, \ldots, k\}$ fix $\delta_i > 0$. Let $a, b_1, \ldots, b_k, c$ be rationals for which the following holds: for every $n, t_1, \ldots, t_k \in \mathbb{N}$ such that $n > \sum_{i=1}^k \delta_i t_i \geq 0$, it holds that $0 \leq an + \sum_{i=1}^k b_i t_i + c \leq n$. Then it is the case that*

$$0 \leq \ a \ \leq 1, \tag{5.7}$$

$$-\delta_i - 1 < \ b_i \ < \delta_i + 1, \ \textit{for all } i = 1, \ldots, k \tag{5.8}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 1 \leq \ c \ \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. \tag{5.9}$$

Proof. Let $\mathbf{P}_{RC}$ be the set of all tuples $(n, t_1, \ldots, t_k) \in \mathbb{N}^{k+1}$ that satisfy $n > \sum_{i=1}^k \delta_i t_i \geq 0$. Thus, we assume that for $a, b_1, \ldots, b_k, c \in \mathbb{Q}$ the following holds:

$$0 \leq an + \sum_{i=1}^k b_i t_i + c \leq n, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{5.10}$$

We show that if any of the conditions (5.7)–(5.9) is violated, we obtain a contradiction by finding $(n^0, t_1^0, \ldots, t_k^0) \in \mathbf{P}_{RC}$ such that $0 \leq an^0 + \sum_{i=1}^k b_i t_i^0 + c \leq n^0$ does not hold.

Proof of (5.7). Let us first show that $0 \leq a \leq 1$.

Assume by contradiction that $a > 1$. From (5.10) we know that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $n \geq an + \sum_{i=1}^k b_i t_i + c$, that is, $(1-a)n \geq \sum_{i=1}^k b_i t_i + c$. Since $1 - a < 0$, we obtain

$$n \leq \frac{\sum_{i=1}^k b_i t_i + c}{1-a}, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{5.11}$$

Consider any tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ where $n^0 > \max\left\{\sum_{i=1}^k \delta_i t_i^0, \frac{\sum_{i=1}^k b_i t_i^0 + c}{1-a}\right\}$. By construction, we obtain: (i) the tuple is in $\mathbf{P}_{RC}$ because $n^0 > \sum_{i=1}^k \delta_i t_i^0$, and (ii) we have $n^0 > \frac{\sum_{i=1}^k b_i t_i^0 + c}{1-a}$, such that we arrive at the required contradiction to (5.11).

Assume now that $a < 0$. Again from (5.10) we have that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $an + \sum_{i=1}^k b_i t_i + c \geq 0$, or in other words $an \geq -\sum_{i=1}^k b_i t_i - c$. As $a < 0$, this means that

$$n \leq \frac{-\sum_{i=1}^k b_i t_i - c}{a}, \text{ for every } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{5.12}$$

Consider a tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ with $n^0 > \max\left\{\sum_{i=1}^k \delta_i t_i^0, \frac{-\sum_{i=1}^k b_i t_i^0 - c}{a}\right\}$. By construction it holds that $n^0 > \sum_{i=1}^k \delta_i t_i^0$, and thus the tuple is in $\mathbf{P}_{RC}$. Also by construction it holds that $n^0 > \frac{-\sum_{i=1}^k b_i t_i^0 - c}{a}$ which is a contradiction with (5.12).

Proof of (5.8). Let us now prove that $-\delta_i - 1 < b_i < \delta_i + 1$, for an arbitrary $i \in \{1, \ldots, k\}$.

Assume by contradiction that $b_i \geq \delta_i + 1$. Recall from (5.10) that for all $(n, t_1, \dots, t_k) \in \mathbf{P}_{RC}$ holds $an + \sum_{j=1}^{k} b_j t_j + c \leq n$, or in other words $(1-a)n \geq \sum_{j=1}^{k} b_j t_j + c$. Since $a \in [0,1]$, then $(1-a)n \leq n$, for every $n \geq 0$. Since $b_i \geq \delta_i + 1$, and $t_i \geq 0$, it holds that $b_i t_i \geq (\delta_i + 1) t_i$. Thus, we have that for every $(n, t_1, \dots, t_k) \in \mathbf{P}_{RC}$ holds that

$$n \geq (1-a)n \geq \sum_{j=1}^{k} b_j t_j + c \geq (\delta_i + 1)t_i + \sum_{j \neq i} b_j t_j + c.$$

In other words, we have that

$$(n - \delta_i t_i) - \sum_{j \neq i} b_j t_j - c \geq t_i, \text{ for all } (n, t_1, \dots, t_k) \in \mathbf{P}_{RC}. \tag{5.13}$$

Consider the tuple $(n^0, t_1^0, \dots, t_k^0) \in \mathbb{N}^{k+1}$ such that $t_i^0 = \max\{1, \sum_{j \neq i}(\delta_j - b_j) - c + 2\}$, $t_j^0 = 1$ for $j \neq i$, and $n^0 = \sum_{j=1}^{k} \delta_j t_j^0 + 1 = \sum_{j \neq i} \delta_j + \delta_i t_i^0 + 1$. This tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{j=1}^{k} \delta_j t_j^0$. Let us check the inequality from (5.13). By construction we have $(n^0 - \delta_i t_i^0) - \sum_{j \neq i} b_j t_j^0 - c = \sum_{j \neq i} \delta_j + \delta_i t_i^0 + 1 - \delta_i t_i^0 - \sum_{j \neq i} b_j - c$, that is, $\sum_{j \neq i}(\delta_j - b_j) - c + 1$, which is strictly smaller than t_i^0 by construction. Thus, we obtained a contradiction with (5.13).

Let us now assume $b_i \leq -\delta_i - 1$. Recall from (5.10) that for all $(n, t_1, \dots, t_k) \in \mathbf{P}_{RC}$ holds $0 \leq an + \sum_{j=1}^{k} b_j t_j + c$. Since $a \in [0,1]$, for every $n \in \mathbb{N}$ holds $an \leq n$, and since $b_i \leq -\delta_i - 1$, we have $b_i t_i \leq -\delta_i t_i - t_i$, for every $t_i \geq 0$. Thus, for every $(n, t_1, \dots, t_k) \in \mathbf{P}_{RC}$ we have

$$0 \leq an + \sum_{j=1}^{k} b_j t_j + c \leq n + (-\delta_i t_i - t_i) + \sum_{j \neq i} b_j t_j + c.$$

In other words, we have that

$$t_i \leq (n - \delta_i t_i) + \sum_{j \neq i} b_j t_j + c, \text{ for all } (n, t_1, \dots, t_k) \in \mathbf{P}_{RC}. \tag{5.14}$$

Consider the tuple $(n^0, t_1^0, \dots, t_k^0) \in \mathbb{N}^{k+1}$ where $t_i^0 = \max\{\sum_{j \neq i}(\delta_j + b_j) + c + 2, 1\}$, $t_j^0 = 1$, for every $j \neq i$, and $n^0 = \sum_{j=1}^{k} \delta_j t_j^0 + 1 = \sum_{j \neq i} \delta_j + \delta_i t_i^0 + 1$. This tuple is in $\mathbf{P}_{RC}$, since $n^0 > \sum_{i=1}^{k} \delta_i t_i^0$. Let us check the inequality from (5.14). By construction we have $(n^0 - \delta_i t_i^0) + \sum_{j \neq i} b_j t_j^0 + c = \sum_{j \neq i} \delta_j + \delta_i t_i^0 + 1 - \delta_i t_i^0 + \sum_{j \neq i} b_j + c$, that is, $\sum_{j \neq i}(\delta_j + b_j) + c + 1$, which is strictly smaller than t_i^0 by construction. This gives us a contradiction with (5.14).

Proof of (5.9). And finally, let us prove that $-2\sum_{i=1}^{k} \delta_i - k - 1 \leq c \leq 2\sum_{i=1}^{k} \delta_i + k + 1$.

Assume by contradiction that $c > 2\sum_{i=1}^{k} \delta_i + k + 1$. Recall that for every $(n, t_1, \dots, t_k) \in \mathbf{P}_{RC}$ holds $n \geq an + \sum_{i=1}^{k} b_i t_i + c$, by (5.10). Since $a \geq 0$, $b_i > -\delta_i - 1$, for every $i = 1, \dots, k$, and $c > 2\sum_{i=1}^{k} \delta_i + k + 1$, then we have that

$$n \geq an + \sum_{i=1}^{k} b_i t_i + c > \sum_{i=1}^{k} (-\delta_i - 1) t_i + 2\sum_{i=1}^{k} \delta_i + k + 1, \text{ for all } (n, t_1, \dots, t_k) \in \mathbf{P}_{RC}. \tag{5.15}$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0)$ where $t_1^0 = \ldots = t_k^0 = 1$, and $n^0 = \sum_{i=1}^k \delta_i t_i^0 + 1 = \sum_{i=1}^k \delta_i + 1$. The tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{i=1}^k \delta_i t_i^0$, but by construction it holds that $\sum_{i=1}^k (-\delta_i - 1)t_i^0 + 2\sum_{i=1}^k \delta_i + k + 1 = \sum_{i=1}^k \delta_i + 1 = n^0$, which is a contradiction with (5.15).

Assume by contradiction that $c < -2\sum_{i=1}^k \delta_i - k - 1$. Recall that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $0 \leq an + \sum_{i=1}^k b_i t_i + c$, by (5.10). Since $a \leq 1$, $b_i < \delta_i + 1$, for every $i = 1, \ldots, k$, and $c < -2\sum_{i=1}^k \delta_i - k - 1$, then we have that

$$0 \leq an + \sum_{i=1}^{k} b_i t_i + c < n + \sum_{i=1}^{k} (\delta_i + 1) t_i - 2\sum_{i=1}^{k} \delta_i - k - 1, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \quad (5.16)$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0)$ where $t_1^0 = \ldots = t_k^0 = 1$, and $n^0 = \sum_{i=1}^k \delta_i t_i^0 + 1 = \sum_{i=1}^k \delta_i + 1$. This tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{i=1}^k \delta_i t_i^0$, but by construction it holds that $n^0 + \sum_{i=1}^k (\delta_i + 1)t_i^0 - 2\sum_{i=1}^k \delta_i - k - 1 = 0$, which is a contradiction with (5.16). □

Now it is straightforward to prove the main theorem.

Theorem 5.1. *Let $n > \sum_{i=1}^k \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, for $1 \leq i \leq k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq an + (b_1 t_1 + \ldots + b_k t_k) + c \quad \text{or} \quad x < an + (b_1 t_1 + \ldots + b_k t_k) + c,$$

where $x \in \Gamma$, and $a, b_1, \ldots, b_k, c \in \mathbb{Q}$. If the guard is sane for the resilience condition, then

$$0 \leq \ a \ \leq 1, \quad (5.17)$$

$$-\delta_i - 1 < \ b_i \ < \delta_i + 1, \ \text{for all } i = 1, \ldots, k \quad (5.18)$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 1 \leq \ c \ \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. \quad (5.19)$$

Proof. As the given guard is sane for the resilience condition, the number compared against a shared variable should have a value from 0 to n. For every tuple $(n, t_1, \ldots, t_k)$ of parameter values satisfying the resilience condition, it should hold that $0 \leq an + \sum_{i=1}^k b_i t_i + c \leq n$. We may thus apply Lemma 5.3 and the theorem follows. □

5.6 Thresholds with floor and ceiling functions

The following theorem considers threshold guards that use the ceiling or the floor function. It uses the same reasoning as in Theorem 5.1, combined with the properties of these functions. Namely, for every $x \in \mathbb{R}$ it holds that $x \leq \lceil x \rceil < x + 1$ and $x - 1 < \lfloor x \rfloor \leq x$.

Theorem 5.4. *Fix a* $k \in \mathbb{N}$. *Let* $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.t_i \geq 0$ *be a resilience condition, where* $\delta_i > 0$, $i = 1, \ldots, k$, *and* $n, t_1, \ldots, t_k \in \Pi$ *are parameters. Fix a threshold guard of the form*

$$x \geq f\left(an + (b_1 t_1 + \ldots + b_k t_k) + c\right) \qquad \textit{or} \qquad x < f\left(an + (b_1 t_1 + \ldots + b_k t_k) + c\right),$$

where $x \in \Gamma$ *is a shared variable,* $a, b_1, \ldots, b_k, c \in \mathbb{Q}$ *are rationals, and* f *is either the ceiling or the floor function. If the guard is sane for the resilience condition, then it holds that*

$$\begin{aligned} 0 \leq & a \leq 1, && (5.20)\\ -\delta_i - 1 < & b_i < \delta_i + 1, \ \textit{for all } i = 1, \ldots, k, && (5.21)\\ -2(\delta_1 + \ldots + \delta_k) - k - 2 \leq & c \leq 2(\delta_1 + \ldots + \delta_k) + k, \ \textit{if } f \textit{ is floor, or} && (5.22)\\ -2(\delta_1 + \ldots + \delta_k) - k \leq & c \leq 2(\delta_1 + \ldots + \delta_k) + k + 2, \ \textit{if } f \textit{ is ceiling.} && (5.23) \end{aligned}$$

Proof sketch. The proof largely follows the arguments of the proof of Lemma 5.3 with fixed denominators as in Corollary 5.2. The only remaining issue is that instead of constraints of the form $0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n$, that are considered in Lemma 5.3, here we have to argue about constraints of the form $0 \leq f\left(an + \sum_{i=1}^{k} b_i t_i + c\right) \leq n$, where f is the ceiling or the floor function.

Let us first discuss the case when f is the ceiling function. As for every $x \in \mathbb{R}$ holds that $x \leq \lceil x \rceil < x + 1$, we have that

$$an + (b_1 t_1 + \ldots + b_k t_k) + c \leq \lceil an + (b_1 t_1 + \ldots + b_k t_k) + c \rceil < an + (b_1 t_1 + \ldots + b_k t_k) + c + 1.$$

Still, as the guard is sane, we have that $0 \leq \lceil an + (b_1 t_1 + \ldots + b_k t_k) + c \rceil \leq n$. Combining these two constraints, we obtain that

$$0 < an + (b_1 t_1 + \ldots + b_k t_k) + (c + 1) \qquad \text{and} \qquad an + (b_1 t_1 + \ldots + b_k t_k) + c \leq n.$$

With these constraints, we can derive a contradiction following the proof of Lemma 5.3.

Similarly, if f is the floor function, we use the fact that for every $x \in \mathbb{R}$ holds that $x - 1 < \lfloor x \rfloor \leq x$. Therefore, we have that

$$an + (b_1 t_1 + \ldots + b_k t_k) + c - 1 < \lfloor an + (b_1 t_1 + \ldots + b_k t_k) + c \rfloor \leq an + (b_1 t_1 + \ldots + b_k t_k) + c.$$

As $0 \leq \lfloor an + (b_1 t_1 + \ldots + b_k t_k) + c \rfloor \leq n$, we obtain that

$$0 \leq an + (b_1 t_1 + \ldots + b_k t_k) + c \qquad \text{and} \qquad an + (b_1 t_1 + \ldots + b_k t_k) + (c - 1) < n.$$

And again, the rest of the proof follows the line of the proof of Lemma 5.3. □

If coefficients in guards have a fixed denominator, we can obtain intervals for numerators as a direct consequence of Theorem 5.4.

Corollary 5.5. *Fix a $k \in \mathbb{N}$. Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.t_i \geq 0$ be a resilience condition, where $\delta_i > 0$, $i = 1, \dots, k$, and $n, t_1, \dots, t_k \in \Pi$ are parameters. Fix a threshold guard of the form*

$$x \geq f\left(\frac{\tilde{a}}{D}n + \sum_{i=1}^{k}\frac{\tilde{b_i}}{D}t_i + \frac{\tilde{c}}{D}\right) \quad \textit{or} \quad x < f\left(\frac{\tilde{a}}{D}n + \sum_{i=1}^{k}\frac{\tilde{b_i}}{D}t_i + \frac{\tilde{c}}{D}\right),$$

where $x \in \Gamma$ is a shared variable, $\tilde{a}, \tilde{b_1}, \dots, \tilde{b_k}, \tilde{c} \in \mathbb{Z}$ are integers, $D \in \mathbb{N}$, and f is either the ceiling or the floor function. If the guard is sane for the resilience condition, then it holds

$$0 \leq a \leq D, \tag{5.24}$$

$$D(-\delta_i - 1) < b_i < D(\delta_i + 1), \textit{ for all } i = 1, \dots, k, \tag{5.25}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k - 2) \leq c \leq D(2(\delta_1 + \ldots + \delta_k) + k), \textit{ if } f \textit{ is floor, or} \tag{5.26}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k) \leq c \leq D(2(\delta_1 + \ldots + \delta_k) + k + 2), \textit{ if } f \textit{ is ceiling.} \tag{5.27}$$

5.7 Discussion

The classic approach to establish correctness of a distributed algorithm is to start with a system model, a specification, and pseudo code, all given in natural language and mathematical definitions, and then write a manual proof that confirms that “all fits together.” Manual correctness proofs mix code inspection, system assumptions, and reasoning about events in the past and the future. Slight modifications to the system assumptions or the code require us to redo the proof. Thus, the proofs often just establish correctness of the algorithm, rather than deriving details of the algorithm — like the threshold guards — from the system assumption or the specification.

We introduced an automated method that synthesizes a correct distributed algorithm from the specifications and the basic assumptions. Our tool computes threshold expressions from the resilience condition and the specification, by learning the constraints that are derived from counterexamples. Learning dramatically reduces the number of verifier calls. In case of BOSCO, the sanity box contains 2^{36} vectors of unknowns, which makes exhaustive search impractical, while our technique only needs to check approximately 500 vectors.

In addition to synthesizing known algorithms from the literature, we considered several modified specifications. For some of them, our tool synthesizes thresholds, while for others it reports that no algorithm of a specific form exists. The latter results are indeed impossibility results (lower bounds on the fraction of correct processes) for fixed sketch threshold automata.

Despite the difficulty of the synthesis of fault-tolerant distributed algorithms [DF09], and in particular undecidability of parameterized synthesis [JB14], there is still an increasing interest in different restrictions of these problems.

Recent work on the synthesis of fault-tolerant distributed algorithms [GT14, FBTK16, FB15] often only applies to a small fixed size of the system, typically for $n < 10$. Bounded synthesis [FS13] deals with the system size by setting SMT constraints for a fixed small size, and then systematically increases it. This idea gave rise to many research paths toward synthesis of distributed systems. Bounded synthesis is an important building block in the development of the semi-decision procedures for token-passing systems with different topologies, inspired by the undecidability results for parameterized synthesis [JB14]. This work uses cutoff results to reduce parameterized synthesis to the synthesis problem for a small number of processes. In [JB14], a framework for exploiting the existing cutoff results for parameterized verification, has been developed in order to attack the parameterized synthesis problem. Other approaches using cutoffs can be found in [BBJ16b, DHJ$^+$16, MPST14]. A recent technique [MFJB18] that combines cutoffs with an SMT based CEGIS loop, successfully synthesizes self-stabilizing protocols in symmetric rings for any system size.

A non-standard synthesis of parameterized systems, that does not take a temporal formula as input, is studied in [KE17] for self-stabilizing rings. This approach allowed the authors to prove a rather surprising claim, namely that the parameterized synthesis problem for unidirectional self-stabilizing rings is decidable, while its parameterized verification problem is undecidable.

The major difference between above mentioned parameterized synthesis approaches and our approach is that we focus on threshold-based FTDAs, that contain parameters as a part of the process code through the thresholds. In other words, we consider parameterized process code, apart from the system size parameterization.

CHAPTER 6

Model Checking of Randomized Distributed Algorithms

Theoretical results [FLP85] show that consensus in asynchronous setting is impossible even in the presence of one faulty process. The attempts to circumvent this result often consist of restricting asynchrony to partial synchrony [DLS88] or relaxing the termination requirement to probabilistic termination [Ben83]. Many systems, including Blockchain [Nak08], provide probabilistic guarantees along unboundedly many rounds. To check their correctness, one has to exploit probabilistic reasoning. We take a step in this direction and reason about randomized distributed algorithms. These algorithms extend asynchronous threshold-guarded distributed algorithms with two features: (i) a random choice (coin toss), and (ii) repeated execution of a single round until it converges (with probability 1).

To this end, we extend the results of Chapter 4 in two ways: (i) we extend threshold automata and introduce *probabilistic threshold automata*, and (ii) we extend the $\mathsf{ELTL}_{\mathsf{FT}}$ logic to the *multi-round* $\mathit{ELTL}_{\mathit{FT}}$, to be able to formulate specifications that include round numbers. More precisely, in this chapter we address the following challenge:

Challenge 6.1. *Given a probabilistic threshold automaton* PTA *and a specification* ψ *(or its negation in multi-round* $\mathit{ELTL}_{\mathit{FT}}$*), check whether it holds that* $\mathit{Sys}(\mathit{PTA}) \models \psi$*, possibly with probability 1.*

We give an overview of our method in Section 6.1. In Section 6.2 we describe properties of probabilistic consensus and intuitive strategies for their verification. The important concepts are defined in Section 6.3. In Sections 6.4–6.6 we explain the techniques for verification of consensus properties for an arbitrary probabilistic threshold automaton. Namely, for non-probabilistic properties we need to reduce the multiple-round specifications to one-round specifications (Section 6.4), as well as to reduce infinite systems

```
 1  Boolean v := input_value
 2  Integer r := 1
 3
 4  while (true) do
 5    send (R,r,v) to all;
 6
 7    wait till received n − t messages (R,r,∗);
 8    if received (n + t) / 2 messages (R,r,w) then
 9      send (P,r,w,D) to all
10    else
11      send (P,r,?) to all;
12
13    wait till received n − t messages (P,r,∗);
14    if received at least t + 1 messages (P,r,w,D) then {
15      v := w;
16      if received at least (n + t) / 2 messages (P,r,w,D) then
17        decide w
18    }
19    else v := 0 or 1 randomly
20    r := r + 1
```

Figure 6.1: Pseudo code of Ben-Or's algorithm for Byzantine faults

to the one-round systems (Section 6.5). For probabilistic properties, the more involved reduction is described in Section 6.6. Experimental evaluation of our results is presented in Section 6.7.

6.1 Overview of the Method

A prominent example of a consensus algorithm is introduced by Ben-Or in [Ben83]. It circumvents the impossibility of asynchronous consensus [FLP85] by relaxing the termination requirement to almost-sure termination, *i.e.*, termination with probability 1. This is achieved by a multi-round fault-tolerant distributed algorithm given in Figure 6.1. Here processes execute the while-loop, and the i-th iteration of this loop is called the i-th *round*. Each round consists of two stages where processes first exchange messages tagged R, wait until the number of received messages reaches a certain threshold (given as expression over parameters in line 7) and then exchange messages tagged P. If n is the number of processes in the system, among which at most t are faulty, then all the thresholds, that is, $n - t$ and $(n + t)/2$ and $t + 1$, should ensure that, for example, no two correct processes ever decide on different values, even if up to t Byzantine faulty processes send conflicting information. At the end of a round, if there is no "strong majority" for a value, that is, $(n + t)/2$ received messages, a process picks a new value randomly in line 19.

While in principle these complex threshold expressions can be dealt with the method

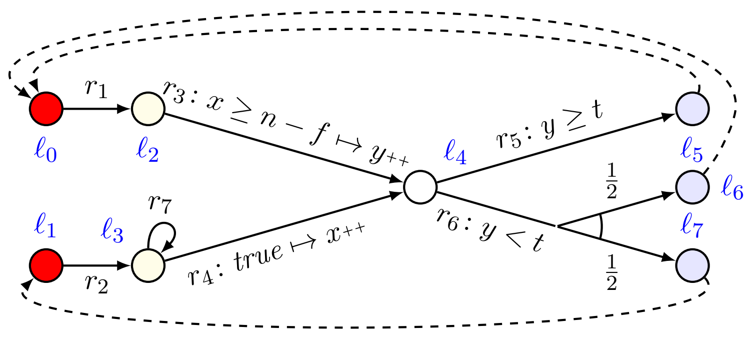


Figure 6.2: Example of a probabilistic threshold automaton.

from Chapter 4, several challenges remain. Basically, the technique from Chapter 4 can be used to verify a fixed number of loop iterations from Figure 6.1. However, consensus algorithms should ensure that there are no two rounds r and r' such that a process decides 0 in r and another decides 1 in r'. This calls for a compositional approach that allows one to compose verification results for individual rounds. A challenge in the composition is that distributed algorithms implement “asynchronous rounds”, that is, during the run processes may be in different rounds in different times.

In addition, the combination of distributed aspects and probabilities makes reasoning difficult. Quoting Lehmann and Rabin [LR81], “proofs of correctness for probabilistic distributed systems are extremely slippery”. This advocates the development of automated verification techniques for probabilistic properties of randomized distributed algorithms in the parameterized setting.

We introduce probabilistic threshold automata to model randomized threshold-based algorithms. An example of such an automaton is given in Figure 6.2. Nodes represent local states (or locations) of processes, which move along the labeled edges or forks. Edges and forks are called rules. Labels have the form $\varphi \mapsto u$, meaning that a process can move along the edge only if φ evaluates to true, and this is followed by the update u of shared variables. Additionally, each tine of a fork is labeled with a number in the $[0, 1]$ interval, representing the probability of a process moving along the fork to end up at the target location of the tine.

If we ignore the dashed arrows in Figure 6.2, a threshold automaton captures the behavior of a process in one round. The dashed edges, called round switch rules, encode how a process, after finishing a round, starts the next round. A naïve unfolding of rounds would produce an infinite number of copies of a threshold automaton, and thus it would produce infinitely many variables.

Threshold automata without probabilistic forks and round switching rules can be automatically checked for safety and liveness (Chapter 4). However, adding forks and round switches is required to adequately model randomized distributed algorithms.

In order to overcome the issue of infinitely many rounds, we prove in Section 6.4 and Section 6.5 that we can verify probabilistic threshold automata by analyzing a one-round automaton, that fits in the framework of Chapter 4. We prove that we can reorder transitions of any fair execution such that their round numbers are in an increasing order. The obtained ordered execution is stutter equivalent to the original one, and thus, they satisfy the same $\mathsf{LTL}_{\text{-}\mathsf{X}}$ properties over the atomic propositions describing only one round. In other words, a system of probabilistic threshold automata can be transformed to a sequential composition of one-round systems.

The main problem with isolating a one-round system is that our specifications often involve at least two different rounds. In this case we need to come up with round invariants that imply the specifications. For example, if we want to verify agreement, we have to check whether two processes can decide different values, possibly in different rounds. We do this in two steps: (i) we check the round invariant that no process changes its decision from round to round, and (ii) we check that within a round no two processes disagree on their decisions.

Finally, we introduce round-rigid adversaries, that respect the natural order of rounds. Verifying almost-sure termination under round-rigid adversaries is in nature very different from proving safety specification which are properties that should hold on every path, and involve no probabilities. It thus calls for special arguments. Our methodology generalizes the ideas of the manual proof of Ben Or's consensus algorithm by Aguilera and Toueg [AT12]. However, our arguments are not specific to Ben Or's algorithm, and we apply it to other randomized distributed algorithms (see Section 6.7). Compared to their paper-and-pencil proof, the threshold automata framework required us to provide a more formal setting and a more informative proof, also pinpointing the needed hypothesis. The crucial parts of our proof are automatically checked by the model checker ByMC. Hence the established correctness stands on more formal ground, which addresses the mentioned concerns of Lehmann and Rabin.

6.2 Consensus Properties and their Verification

Probabilistic consensus consists of safety specifications and an almost-sure termination requirement. We discuss here the specifications of Ben-Or's algorithm shown in Figure 6.1. Every correct process has an initial value from $\{0, 1\}$, and must decide a value, either 0 or 1, such that:

Agreement: No two correct processes decide differently.

Validity: If all correct processes have v as the initial value, then no process decides $1 - v$.

Probabilistic wait-free termination: Under every round-rigid adversary, with probability 1 every correct process eventually decides.

Note that the $\mathsf{ELTL_{FT}}$ logic, presented in Section 2.3, is insufficient for expressing consensus properties, as it does not allow reasoning about different rounds. In Section 6.3.3 we present the *multi-round $\mathsf{ELTL_{FT}}$ logic* for specifying such properties.

Our verification strategies. Note that Agreement and Validity are non-probabilistic properties, while Probabilistic wait-free termination requires probability 1. We have two different strategies for verification of probabilistic and non-probabilistic properties. Both strategies are based on the verification technique presented in Chapter 4. Given a threshold automaton TA and an $\mathsf{ELTL_{FT}}$ formula $\neg\varphi$ (negation of specification φ), the technique from Chapter 4 checks if $\mathsf{Sys(TA)} \models \varphi$.

Note that a threshold automaton can be seen as a degenerate case of a probabilistic threshold automaton, where we have only one round and no probabilistic choices (no forks). Similarly, a formula in $\mathsf{ELTL_{FT}}$ can be seen as a multi-round $\mathsf{ELTL_{FT}}$ formula, with only one round. Thus, we need to reduce the probabilistic setting to the form that fits to Chapter 4.

The main reasons why the technique from Chapter 4 cannot be applied directly to randomized FTDAs, and our strategies how to overcome them, are as follows:

- Verification of non-probabilistic properties, like Agreement and Validity:

  Specifications Non-probabilistic specifications involve multiple rounds. In Section 6.4 we reduce them to formulas from multi-round $\mathsf{ELTL_{FT}}$ but with only one round quantifier, that is, we reduce them to $\mathsf{ELTL_{FT}}$ from Section 2.3.

  System For checking non-probabilistic properties, we replace probabilistic choices with non-deterministic choices, and introduce infinite non-probabilistic counter systems in Section 6.3.2. In Section 6.5 we reduce them to one-round non-probabilistic counter systems, that is, exactly to those counter systems defined in Section 2.2.

- Verification of probabilistic properties, e.g., Probabilistic wait-free termination:

  Specifications Reasoning about properties that hold with probability 1 is out of the scope of Chapter 4. In Section 6.6 we reduce such requirements to multi-round $\mathsf{ELTL_{FT}}$ formulas with one-round quantifier, that is, again to $\mathsf{ELTL_{FT}}$.

  System For checking probabilistic properties, we already assume that we have round-rigid adversaries, which impose the natural order on rounds. This makes the isolation of the one round system trivial.

6.3 Framework of Probabilistic Threshold Automata

For modeling randomized distributed algorithms, we introduce *probabilistic threshold automata.* What distinguishes them from threshold automata defined in Section 2.1, are differently defined rules (edges and forks), and locations with special roles (border and final locations).

A *probabilistic threshold automaton* PTA is a tuple $(\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$, where

- $\mathcal{L}$ is a finite set of locations, that contains the following disjoint subsets: *initial locations* $\mathcal{I}$, *final locations* $\mathcal{F}$, and *border locations* $\mathcal{B}$, with $|\mathcal{B}| = |\mathcal{I}|$;
- $\mathcal{V}$ is a set of variables. It is partitioned in two sets: Π contains *parameter variables*, and Γ contains *shared variables*;
- $\mathcal{R}$ is a finite set of *rules*; and
- RC, the *resilience condition*, is a formula in linear integer arithmetic over variables that encode parameters.

A rule r is a tuple $(\mathit{from}, \delta_{to}, \varphi, \mathbf{u})$ where $\mathit{from} \in \mathcal{L}$ is the *source* location, $\delta_{to} \in \mathsf{Dist}(\mathcal{L})$ is a probability distribution over the *destination* locations, $\mathbf{u} \in \mathbb{N}_0^{|\Gamma|}$ is the *update vector*, and φ is a *guard* of the form $x \geq \bar{a} \cdot \mathbf{p}^\intercal + a_0$ or $x < \bar{a} \cdot \mathbf{p}^\intercal + a_0$, where $x \in \Gamma$ is a shared variable, $(\bar{a}, a_0) \in \mathbb{Q}^{|\Pi|+1}$ is a vector of rational numbers, and $\mathbf{p}$ is the vector or all parameters. If $r.\delta_{to}$ is a Dirac distribution, i.e., there exists $\ell \in \mathcal{L}$ such that $\delta_{to}(\ell) = 1$, we call r a *Dirac rule*, and sometimes write it as $(\mathit{from}, \ell, \varphi, \mathbf{u})$. We introduce a restriction that all destination locations of non-Dirac rules are final locations.

Probabilistic threshold automata allow one to model algorithms with successive identical rounds. Informally, a round happens between border locations and the succeeding final locations, then round switch rules let processes move from final locations of a given round to border locations of the next round. From each border location there is exactly one Dirac rule to an initial location, and it has a form $(\ell, \ell', \mathtt{true}, \mathbf{0})$ where $\ell \in \mathcal{B}$ and $\ell' \in \mathcal{I}$. As $|\mathcal{B}| = |\mathcal{I}|$, one can think of border locations as copies of initial locations. It remains to model from which final locations to which border locations (that is, initial for the next round) processes move. This is done by *round switch rules*. These rules are deterministic, and can be described by a function $\rho \colon \mathcal{F} \to \mathcal{B}$, or equivalently as Dirac rules $(\ell, \ell', \mathtt{true}, \mathbf{0})$ with $\ell \in \mathcal{F}$ and $\ell' \in \mathcal{B}$. The set of round switch rules is denoted by $\mathcal{S} \subseteq \mathcal{R}$.

We assume the following structure for probabilistic threshold automata. A location is a border location if and only if all its incoming edges are round switch edges. Similarly, a location is final if and only if there is only one outgoing edge and this edge is a round switch rule.

Example 6.1. In Figure 6.2 we have a PTA with border locations $\mathcal{B} = \{\ell_0, \ell_1\}$, initial locations $\mathcal{I} = \{\ell_2, \ell_3\}$, and final locations $\mathcal{F} = \{\ell_5, \ell_6, \ell_7\}$. The only rule that is not Dirac is r_6. Round switch rules are represented by dashed arrows. ◁

6.3.1 Probabilistic Counter Systems

Given a probabilistic threshold automaton PTA, we define its semantics, called the *probabilistic counter system* $\mathsf{Sys}(\mathsf{PTA})$, to be the infinite-state Markov Decision Process (MDP) $(\Sigma, I, \mathsf{Act}, R)$, where Σ is the set of configurations for PTA among which $I \subseteq \Sigma$ are initial, the set of actions is $\mathsf{Act} = \mathcal{R} \times \mathbb{N}_0$ and $R\colon \Sigma \times \mathsf{Act} \to \mathsf{Dist}(\Sigma)$ is the probabilistic transition function.

As in the non-probabilistic case, every resilience condition RC defines the set of *admissible parameters* $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}_0^{|\Pi|} : \mathbf{p} \models RC\}$. A function $N\colon \mathbf{P}_{RC} \to \mathbb{N}_0$ maps a vector of admissible parameters to a number of modeled processes in the system. For example, if we want to tolerate Byzantine faults, we model only $N(n,t,f) = n - f$ processes; if we have crash faults, we model all $N(n,t,f) = n$ processes.

Configurations. As we need to capture the values of variables for every round, every configuration $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$ is determined by the following components:

- a function $\sigma.\boldsymbol{\kappa}\colon \mathcal{L} \times \mathbb{N}_0 \to \mathbb{N}_0$ that defines values of local state counters per round. That is, $\sigma.\boldsymbol{\kappa}[\ell, k]$ stores the counter value for location ℓ and round k in configuration σ.
- a function $\sigma.\mathbf{g}\colon \Gamma \times \mathbb{N}_0 \to \mathbb{N}_0$ defining shared variable values per round. Similarly, $\sigma.\mathbf{g}[x, k]$ stores the value of the shared variable x in round k in configuration σ.
- a vector $\sigma.\mathbf{p} \in \mathbb{N}_0^{|\Pi|}$ of parameter values, which remains the same for every round.

By $\mathbf{g}[k]$ we denote the vector $(\mathbf{g}[x,k])_{x\in\Gamma}$ of shared variables in a round k, and similarly by $\boldsymbol{\kappa}[k]$ we denote the vector $(\boldsymbol{\kappa}[\ell,k])_{\ell\in\mathcal{L}}$ of local state counters in a round k.

A configuration $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$ is *initial* if for every $x \in \Gamma$ and $k \in \mathbb{N}_0$ we have $\sigma.\mathbf{g}[x,k] = 0$, if $\sum_{\ell\in\mathcal{B}} \sigma.\boldsymbol{\kappa}[\ell, 0] = N(\mathbf{p})$, and finally if $\sigma.\boldsymbol{\kappa}[\ell,k] = 0$, for every $(\ell, k) \in (\mathcal{L}\setminus\mathcal{B}) \times \{0\} \cup \mathcal{L} \times \mathbb{N}$.

We say that a threshold guard $\varphi : x \geq \bar{a} \cdot \mathbf{p}^\intercal + a_0$ evaluates to true in a configuration σ for a round k, and write $\sigma, k \models \varphi$, if $\sigma.\mathbf{g}[x,k] \geq \bar{a} \cdot \sigma.\mathbf{p}^\intercal + a_0$. Similarly we define when a guard of the other form, that is, $x < \bar{a} \cdot \mathbf{p}^\intercal + a_0$, evaluates to true in σ for a round k.

Actions. Actions are induced by the rules of a PTA, and have the form $\alpha = (r, k) \in \mathcal{R} \times \mathbb{N}_0$. Intuitively, an action (r, k) stands for a process moving along the rule r in the round k (either along the edge if r is a Dirac rule, or otherwise, moving along the fork and ending up in a destination location). We use notation $\alpha.\mathit{from}$ for $r.\mathit{from}$, $\alpha.\varphi$ for $r.\varphi$, etc. If r is a Dirac rule, we say α is a Dirac action.

An action $\alpha = (r, k)$ is *unlocked* in configuration σ, if its guard evaluates to true in its round, that is $\sigma, k \models \varphi$. An action $\alpha = (r, k)$ is *applicable* to a configuration σ if α is unlocked in σ, and $\sigma.\boldsymbol{\kappa}[r.\mathit{from}, k] \geq 1$.

Remark 6.1. Note that in the probabilistic case we do not allow acceleration, as in the previous chapters. Namely, if several processes simultaneously move along a fork, we cannot guarantee that they will all arrive at the same destination location. Later we reduce our reasoning to the known non-probabilistic case, and then we are again allowed to use the ByMC tool that is based on acceleration.

Similarly as when executing a transition in the non-probabilistic case, we show how each component of a configuration changes when an action is executed in the probabilistic counter system.

Definition 6.1. *We introduce a partial function apply*: $\mathsf{Act} \times \mathcal{L} \times \Sigma \nrightarrow \Sigma$ *such that given an action* $\alpha = (r, k) \in \mathsf{Act}$*, a location* $\ell \in \mathcal{L}$*, and a configuration* σ*, the result of* $apply(\alpha, \ell, \sigma)$ *is defined if and only if* α *is applicable to* σ *and* $\alpha.\delta_{to}(\ell) > 0$*. We have that* $apply(\alpha, \ell, \sigma) = \sigma'$ *if and only if* $apply(\alpha, \ell, \sigma)$ *is defined and the following holds:*

- $\sigma'.\mathbf{g}[k] = \sigma.\mathbf{g}[k] + \alpha.\mathbf{u}$*, and* $\sigma'.\mathbf{g}[k'] = \sigma.\mathbf{g}[k']$*, for every round* $k' \neq k$*,*
- $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$*,*
- *if* $r \in \mathcal{R} \setminus \mathcal{S}$ *and* $\alpha.from \neq \ell$*, that is, if* r *is not a round switch rule and not a self-loop, then*
  - $\sigma'.\boldsymbol{\kappa}[\alpha.from, k] = \sigma.\boldsymbol{\kappa}[\alpha.from, k] - 1$*,*
  - $\sigma'.\boldsymbol{\kappa}[\ell, k] = \sigma.\boldsymbol{\kappa}[\ell, k] + 1$*,*
  - $\forall \ell \in \mathcal{L} \setminus \{\alpha.from, \ell\}$*,* $\sigma'.\boldsymbol{\kappa}[\ell, k] = \sigma.\boldsymbol{\kappa}[\ell, k]$*, and*
  - $\sigma'.\boldsymbol{\kappa}[k'] = \sigma.\boldsymbol{\kappa}[k']$*, for all rounds* $k' \neq k$

  if $r \in \mathcal{R} \setminus \mathcal{S}$ *and* $\alpha.from = \ell$*, that is, if* r *is a self-loop (and thus not a round switch rule), then* $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa}$*,*
  if $r \in \mathcal{S}$ *is a round switch rule, then*
  - $\sigma'.\boldsymbol{\kappa}[\alpha.from, k] = \sigma.\boldsymbol{\kappa}[\alpha.from, k] - 1$*,*
  - $\sigma'.\boldsymbol{\kappa}[\ell, k+1] = \sigma.\boldsymbol{\kappa}[\ell, k+1] + 1$*, and*
  - $\sigma'.\boldsymbol{\kappa}[\ell', k'] = \sigma.\boldsymbol{\kappa}[\ell', k']$*, for all* $(\ell', k') \in \mathcal{L} \times \mathbb{N}_0 \setminus \{(\alpha.from, k), (\ell, k+1)\}$*.*

A probabilistic transition function R is defined such that for every two configurations σ and σ' and for every action α applicable to σ, we have

$$R(\sigma, \alpha)(\sigma') = \begin{cases} \alpha.\delta_{to}(\ell) > 0, & \text{if } apply(\alpha, \ell, \sigma) = \sigma', \\ 0, & \text{otherwise.} \end{cases}$$

6.3.2 Infinite Non-probabilistic Counter Systems

Specifications of randomized distributed algorithms often include non-probabilistic properties, like Agreement and Validity in the case of consensus (Section 6.2). For verifying such properties, concrete values of probability distributions on forks do not play a role. Thus, we can replace probabilistic choices with non-deterministic choices. In this way, with any PTA we naturally associate a non-probabilistic threshold automaton, as defined in Section 2.1.

Definition 6.2. *Given a* $\mathsf{PTA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$*, its* underlying (non-probabilistic) threshold automaton *is* $\mathsf{TA}_{\mathsf{PTA}} = (\mathcal{L}, \mathcal{V}, \mathcal{R}_{np}, RC)$ *where the set of rules* $\mathcal{R}_{np}$ *is defined as*

$$\mathcal{R}_{np} = \{r_\ell = (\mathit{from}, \ell, \varphi, \mathbf{u}) \colon r = (\mathit{from}, \delta_{to}, \varphi, \mathbf{u}) \in \mathcal{R} \wedge \ell \in \mathcal{L} \wedge \delta_{to}(\ell) > 0\}.$$

We write TA instead of $\mathsf{TA}_{\mathsf{PTA}}$ when it is clear which PTA we refer to. Note that every rule from $\mathcal{R}_{np}$ corresponds to exactly one rule in $\mathcal{R}$, and for every rule in $\mathcal{R}$ there is at least one corresponding rule in $\mathcal{R}_{np}$ (and exactly one for Dirac rules).

We present two equivalent definitions of an infinite non-probabilistic counter system of a PTA. We can either consider the probabilistic counter system Sys(PTA), and then reduce every action to a transition (Definition 6.3), or we can replace every fork in a PTA with nondeterminism (multiple edges), and obtain a TA, and then consider its counter system defined analogously as in Section 2.2 but with multiple rounds (Definition 6.4). This is illustrated in Figure 6.3.

Definition 6.3. *Given a probabilistic counter system* $\mathsf{Sys}(\mathsf{PTA}) = (\Sigma, I, \mathsf{Act}, R_P)$*, we define its non-probabilistic version* $\mathsf{Sys}_{np}(\mathsf{PTA})$ *to be the tuple* (Σ, I, R)*, where* R *is a transition relation defined below.*

If $\mathsf{Act} = \mathcal{R} \times \mathbb{N}_0$ *and if* $\mathcal{R}_{np}$ *is defined from* $\mathcal{R}$ *as in Definition 6.2, then every transition is a tuple* $t = (r_\ell, k) \in \mathcal{R}_{np} \times \mathbb{N}_0$ *such that* $\alpha = (r, k)$ *is an action from* Act *and for* $\ell \in \mathcal{L}$ *holds that* $\alpha.\delta_{to}(\ell) > 0$*. Transition* t *is unlocked in a configuration* σ *from* $\mathsf{Sys}_{np}(\mathsf{PTA})$ *if* α *is unlocked in* σ *in* $\mathsf{Sys}(\mathsf{PTA})$*. Similarly we define when* t *is applicable to* σ*. We obtain* σ' *by applying an applicable transition* t *to* σ*, written* $t(\sigma) = \sigma'$*, if and only if there exists a location* $\ell \in \mathcal{L}$ *such that* $\mathrm{apply}(\alpha, \ell, \sigma) = \sigma'$*.*

Two configurations σ *and* σ' *are in the transition relation* R*, i.e.,* $(\sigma, \sigma') \in R$*, if and only if there exists a transition* t *such that* $\sigma' = t(\sigma)$*.*

Definition 6.4. *Given an arbitrary* $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}_{np}, RC)$*, with border, initial, and final location sets* $\mathcal{B}$*,* $\mathcal{I}$*, and* $\mathcal{F}$*, respectively, we define its* infinite counter system $\mathsf{Sys}_\infty(\mathsf{TA})$ *to be the tuple* (Σ, I, R)*. Configurations from* Σ *and* I *are defined as in Section 6.3.1. A transition* t *is a tuple* $(r_\ell, k) \in \mathcal{R}_{np} \times \mathbb{N}_0$*. Since it coincides with Dirac actions, we define when a transition is unlocked in a configuration and when it is applicable to a configuration, in the same way as for a Dirac action in Section 6.3.1. A configuration* σ' *is obtained by applying an applicable transition* $t = (r_\ell, k)$ *to* σ*, written* $\sigma' = t(\sigma)$*, if and*

$$\begin{tikzcd}
\mathsf{PTA} \arrow[rr, "\text{Definition 6.2}"] \arrow[dd, "\text{Section 6.3.1}"] & & \mathsf{TA} \arrow[d, "\text{Definition 6.4}"] \\
& & \mathsf{Sys}_{\infty}(\mathsf{TA}) \\
\mathsf{Sys}(\mathsf{PTA}) \arrow[r, "\text{Definition 6.3}"] & \mathsf{Sys}_{np}(\mathsf{PTA}) \arrow[ur, leftrightarrow, "="] &
\end{tikzcd}$$

Figure 6.3: Diagram following Proposition 6.1 presents two ways to obtain an infinite non-probabilistic counter system, given a PTA.

only if apply$(\alpha, \ell, \sigma) = \sigma'$, *for a Dirac action* $\alpha = (r_\ell, k)$ *and the destination location* ℓ *of* r.

Now we have $(\sigma, \sigma') \in R$ *if and only if there exists a transition* t *such that* $\sigma' = t(\sigma)$.

Proposition 6.1. *Given a PTA, the non-probabilistic version* $\mathsf{Sys}_{np}(\mathsf{PTA})$ *of its counter system coincides with the infinite counter system* $\mathsf{Sys}_{\infty}(\mathsf{TA})$ *of its threshold automaton.*

It is easy to see that the diagram from Figure 6.3 commutes, and thus every PTA yields the unique non-probabilistic counter system. The two constructions give us possibility to remove probabilistic reasoning either on the level of a PTA (using Definition 6.2) or on the level of a counter system Sys(PTA) (using Definition 6.3). As the definitions are equivalent, we can use both interchangeably.

Since $\mathsf{Sys}_{\infty}(\mathsf{TA})$ is a non-probabilistic counter system, we use notions as defined in Section 2.2. Here we recall those that are needed for this chapter.

A (finite or infinite) sequence of transitions is called *schedule*, and it is often denoted by τ. A schedule $\tau = t_1, t_2, \ldots, t_{|\tau|}$ is applicable to a configuration σ if there exists a sequence of configurations $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_{|\tau|}$ such that for every $1 \leq i \leq |\tau|$ we have that t_i is applicable to σ_{i-1} and $\sigma_i = t_i(\sigma_{i-1})$. A *path* is an alternating sequence of configurations and transitions, for example $\sigma_0, t_1, \sigma_1, \ldots, t_{|\tau|}, \sigma_{|\tau|}$, such that for every t_i, $1 \leq i \leq |\tau|$, in the sequence, we have that t_i is applicable to σ_{i-1} and $\sigma_i = t_i(\sigma_{i-1})$. Given a configuration σ_0 and a schedule $\tau = t_1, t_2, \ldots, t_{|\tau|}$, a path $\sigma_0, t_1, \sigma_1, \ldots, t_{|\tau|}, \sigma_{|\tau|}$ where $t_i(\sigma_{i-1}) = \sigma_i$, $1 \leq i \leq |\tau|$, we denote by $\mathsf{path}(\sigma_0, \tau)$. Similarly we define an infinite schedule $\tau = t_1, t_2, \ldots$, and an infinite path $\sigma_0, t_1, \sigma_1, \ldots$, also denoted by $\mathsf{path}(\sigma_0, \tau)$. An infinite path is *fair* if whenever a transition is applicable, it will eventually be performed.

Since every transition in $\mathsf{Sys}_{\infty}(\mathsf{TA})$ comes from an action in Sys(PTA), note that every path in $\mathsf{Sys}_{\infty}(\mathsf{TA})$ is a valid path in Sys(PTA).

6.3.3 Multi-round $\mathsf{ELTL}_{\mathsf{FT}}$ Logic

In order to formally express the negations of consensus specifications, we extend the $\mathsf{ELTL}_{\mathsf{FT}}$ logic in the following two aspects:

- Atomic propositions have form $\boldsymbol{\kappa}[\ell, k] > 0$, instead of $\boldsymbol{\kappa}[\ell] > 0$, and $\boldsymbol{\kappa}[\ell, k] = 0$, instead of $\boldsymbol{\kappa}[\ell] = 0$, where ℓ is a location, and k is a round.
- We allow existential quantification over rounds, that is, formulas have form
$$(\exists \bar{k} \in \mathbb{N}_0^m)\, \mathsf{E}\, \varphi(\bar{k}), \tag{6.1}$$
where $\bar{k}$ is a vector of round variables $(k_1, \ldots, k_m)$. Thus, specifications have the form $(\forall \bar{k} \in \mathbb{N}_0^m) \mathsf{A}\, \neg\varphi(\bar{k})$.

We call this logic *multi-round $\mathsf{ELTL}_{\mathsf{FT}}$*. Here we express consensus specifications, that is, formulas whose negations are in multi-round $\mathsf{ELTL}_{\mathsf{FT}}$.

Formalization. In order to formulate and analyze specifications, we partition sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{F}$, each into two subsets, e.g., $\mathcal{I}_0$ and $\mathcal{I}_1$, and an analogous notation for the subsets of $\mathcal{B}$ and $\mathcal{F}$. Here are the restrictions for every $v \in \{0, 1\}$:

- Processes that are initially in a location $\ell \in \mathcal{I}_v$ have the initial value v.
- Rules connecting locations from $\mathcal{B}$ and $\mathcal{I}$ respect the partitioning, i.e., they connect $\mathcal{B}_v$ and $\mathcal{I}_v$.
- Similarly, rules connecting locations from $\mathcal{F}$ and $\mathcal{B}$ respect the partitioning.

We also introduce two subsets $\mathcal{D}_v \subseteq \mathcal{F}_v$, for $v \in \{0, 1\}$. Intuitively, a process is in $\mathcal{D}_v$ in a round k if and only if it decides v in that round.

Now we can express consensus specifications, such that their negations are in multi-round $\mathsf{ELTL}_{\mathsf{FT}}$, as follows:

Agreement: For both values $v \in \{0, 1\}$ holds the following:
$$(\forall k \in \mathbb{N}_0)(\forall k' \in \mathbb{N}_0)\, \mathsf{A}\, (\mathsf{F} \bigvee_{\ell \in \mathcal{D}_v} \boldsymbol{\kappa}[\ell, k] > 0 \;\rightarrow\; \mathsf{G} \bigwedge_{\ell' \in \mathcal{D}_{1-v}} \boldsymbol{\kappa}[\ell', k'] = 0) \tag{6.2}$$

Validity: For both $v \in \{0, 1\}$ it holds
$$(\forall k \in \mathbb{N}_0)\, \mathsf{A}\, (\bigwedge_{\ell \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell, 0] = 0 \;\rightarrow\; \mathsf{G} \bigwedge_{\ell' \in \mathcal{D}_v} \boldsymbol{\kappa}[\ell', k] = 0) \tag{6.3}$$

Probabilistic wait-free termination: For every round-rigid adversary s
$$\mathbb{P}_{\mathsf{s}}(\bigvee_{k \in \mathbb{N}_0} \bigvee_{v \in \{0,1\}} \mathsf{G} \bigwedge_{\ell \in \mathcal{F} \setminus \mathcal{D}_v} \boldsymbol{\kappa}[\ell, k] = 0) \quad = \quad 1 \tag{6.4}$$

6.4 Reduction to Specifications with one Round Quantifier

While Agreement contains two round variables, Validity talks about round 0 and a round $k \in \mathbb{N}_0$. Thus, both of these specifications involve two round numbers. Our goal is to reduce reasoning from unboundedly many rounds to one round, so we can use the ByMC tool, that can only analyze one round systems. Therefore, properties are only allowed to talk about one round number. In this Section we show how to check formulas (6.2) and (6.3) by checking properties that describe one round. Namely, we introduce two properties, round invariants (6.5) and (6.6), and prove that they imply our two specifications.

The first round invariant claims that in every round and in every path, if a process decides v in a round, no process ever enters a location from $\mathcal{F}_{1-v}$ in that round. Formally written, we have the following:

$$(\forall k \in \mathbb{N}_0)\ \mathsf{A}\,(\mathsf{F} \bigvee_{\ell \in \mathcal{D}_v} \boldsymbol{\kappa}[\ell, k] > 0 \quad \rightarrow \quad \mathsf{G} \bigwedge_{\ell' \in \mathcal{F}_{1-v}} \boldsymbol{\kappa}[\ell', k] = 0). \tag{6.5}$$

The second round invariant claims that in every round in every path, if no process starts a round with a value v, then no process terminates that round with value v. Formally, the following formula holds:

$$(\forall k \in \mathbb{N}_0)\ \mathsf{A}\,(\mathsf{G} \bigwedge_{\ell \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell, k] = 0 \quad \rightarrow \quad \mathsf{G} \bigwedge_{\ell' \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell', k] = 0). \tag{6.6}$$

The benefit of analyzing these two formulas instead of (6.2) and (6.3) lies in the fact that formulas (6.5) and (6.6) describe properties of only one round in a path. Next we want to prove that formulas (6.5) and (6.6) imply formulas (6.2) and (6.3).

Let us first give some useful properties of $\mathsf{Sys}_\infty(\mathsf{TA})$.

Lemma 6.2 (Round Switch)**.** *For every $\mathsf{Sys}_\infty(\mathsf{TA})$ and every $v \in \{0,1\}$ the following holds:*

$$(\forall k \in \mathbb{N}_0)\ \boldsymbol{\mathsf{A}}\,(\boldsymbol{\mathsf{G}} \bigwedge_{\ell \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell, k] = 0 \quad \rightarrow \quad \boldsymbol{\mathsf{G}} \bigwedge_{\ell' \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell', k+1] = 0). \tag{6.7}$$

Proof. By definitions of $\mathcal{F}_v$, $\mathcal{B}_v$ and $\mathcal{I}_v$, we have that

$$(\forall k \in \mathbb{N}_0)\ \mathsf{A}\,(\mathsf{G} \bigwedge_{\ell \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell, k] = 0 \rightarrow \mathsf{G} \bigwedge_{\ell'' \in \mathcal{B}_v} \boldsymbol{\kappa}[\ell'', k+1] = 0), \text{ and}$$

$$(\forall k \in \mathbb{N}_0)\ \mathsf{A}\,(\mathsf{G} \bigwedge_{\ell'' \in \mathcal{B}_v} \boldsymbol{\kappa}[\ell'', k+1] = 0 \rightarrow \mathsf{G} \bigwedge_{\ell' \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell', k+1] = 0).$$

The two formulas together yield the required one for both values of v. □

Lemma 6.3. *For every $\mathsf{Sys}_\infty(\mathsf{TA})$ such that $\mathsf{Sys}_\infty(\mathsf{TA}) \models (6.6)$, and for every $v \in \{0,1\}$, the following holds:*

$$(\forall k \in \mathbb{N}_0)(\forall k' \in \mathbb{N}_0)(k \leq k' \to \boldsymbol{A}(\boldsymbol{G} \bigwedge_{\ell \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell, k] = 0 \to \boldsymbol{G} \bigwedge_{\ell' \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell', k'] = 0)), \tag{6.8}$$

$$(\forall k \in \mathbb{N}_0)(\forall k' \in \mathbb{N}_0)(k \leq k' \to \boldsymbol{A}(\boldsymbol{G} \bigwedge_{\ell \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell, k] = 0 \to \boldsymbol{G} \bigwedge_{\ell' \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell', k'] = 0)). \tag{6.9}$$

Proof. Assume formula (6.6) holds. Note that Lemma 6.2 together with formula (6.6) gives us that globally empty initial location is a round invariant. Formally, by transitivity we have that

$$(\forall k \in \mathbb{N}_0)\ \mathbf{A}\ (\mathbf{G} \bigwedge_{\ell \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell, k] = 0\ \to\ \mathbf{G} \bigwedge_{\ell' \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell', k+1] = 0). \tag{6.10}$$

By induction we obtain the required formula (6.8). Finally, by combining formulas (6.7), (6.6) and (6.8) we obtain formula (6.9). □

Now we are ready to prove the main claim, that is, formulas (6.5) and (6.6) imply formulas (6.2) and (6.3).

Proposition 6.4. *If $\mathsf{Sys}_\infty(\mathsf{TA}) \models (6.5) \land (6.6)$, then $\mathsf{Sys}_\infty(\mathsf{TA}) \models (6.2) \land (6.3)$.*

Proof. Assume $\mathsf{Sys}_\infty(\mathsf{TA}) \models (6.5) \land (6.6)$.

Let us first focus on formula (6.2), and prove that $\mathsf{Sys}_\infty(\mathsf{TA}) \models (6.2)$. Assume by contradiction that the formula does not hold on $\mathsf{Sys}_\infty(\mathsf{TA})$, that is, there exist rounds $k, k' \in \mathbb{N}_0$ and a path π such that:

$$\pi \models \mathbf{F} \bigvee_{\ell_0 \in \mathcal{D}_0} \boldsymbol{\kappa}[\ell_0, k] > 0\ \land\ \mathbf{F} \bigvee_{\ell_1 \in \mathcal{D}_1} \boldsymbol{\kappa}[\ell_1, k'] > 0. \tag{6.11}$$

Since by formula (6.11) we have $\pi \models \mathbf{F} \bigvee_{\ell_0 \in \mathcal{D}_0} \boldsymbol{\kappa}[\ell_0, k] > 0$, then from formula (6.5) with $v = 0$ we obtain that it also holds $\pi \models \mathbf{G} \bigwedge_{\ell \in \mathcal{F}_1} \boldsymbol{\kappa}[\ell, k] = 0$. As $\mathcal{D}_1 \subseteq \mathcal{F}_1$, we know that no process decides 1 in round k. Now formula (6.9) from Lemma 6.3 for $v = 1$ yields that $\pi \models \mathbf{G} \bigwedge_{\ell \in \mathcal{F}_1} \boldsymbol{\kappa}[\ell, k_1] = 0$ for every $k_1 \geq k$, i.e., in any round greater than k no process will ever decide 1. As by (6.11) we have that $\pi \models \mathbf{F} \bigvee_{\ell_1 \in \mathcal{D}_1} \boldsymbol{\kappa}[\ell_1, k'] > 0$, i.e., a process decides 1 in a round k', thus it must be that $k' < k$.

Now we consider the other part of formula (6.11), i.e., $\pi \models \mathbf{F} \bigvee_{\ell_1 \in \mathcal{D}_1} \boldsymbol{\kappa}[\ell_1, k'] > 0$. By following the analogous analysis we conclude that it must be that $k < k'$. This brings us to the contradiction with $k' < k$, which proves the first part of the statement, that violation of (6.5) and (6.6) implies violation of (6.2).

Next we focus on formula (6.3), and prove by contradiction that it must hold. We start by assuming that the formula does not hold, that is, there exists a round k and a path π

such that no process starts the first round of π with value v and eventually in a round k a process decides v. Formally,

$$\pi \models \bigwedge_{\ell \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell, 0] = 0 \ \wedge\ \mathsf{F} \bigvee_{\ell' \in \mathcal{D}_v} \boldsymbol{\kappa}[\ell', k] > 0. \tag{6.12}$$

Note first that $\pi \models \bigwedge_{\ell \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell, 0] = 0$ implies $\pi \models \mathsf{G} \bigwedge_{\ell \in \mathcal{I}_v} \boldsymbol{\kappa}[\ell, 0] = 0$. Then, since $\mathsf{Sys}_\infty(\mathsf{TA}) \models (6.6)$, that is, since formula (6.6) holds, we have that $\pi \models \mathsf{G} \bigwedge_{\ell' \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell', 0] = 0$. Then by formula (6.9) we have that for every $k \in \mathbb{N}_0$ it holds $\pi \models \mathsf{G} \bigwedge_{\ell' \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell', k] = 0$. Since $\mathcal{D}_v \subseteq \mathcal{F}_v$, we also have that $\pi \models \mathsf{G} \bigwedge_{\ell' \in \mathcal{D}_v} \boldsymbol{\kappa}[\ell', k] = 0$. As this contradicts our assumption from (6.12) that $\pi \models \mathsf{F} \bigvee_{\ell' \in \mathcal{D}_v} \boldsymbol{\kappa}[\ell', k] > 0$, it proves the second part of the statement, that violation of (6.5) and (6.6) implies violation of (6.3). $\square$

6.5 Reduction to One-Round Counter System

Given a property describing one round, our goal is to prove that there is a counterexample to the property in the infinite system if and only if there is a counterexample in a one-round system. This is formulated in Theorem 6.11, and it allows us to use the existing technique from Chapter 4 on a one-round system.

The proof idea contains two parts. Firstly, in Section 6.5.2 we prove that one can exchange an arbitrary finite schedule with a round-rigid one, while preserving atomic propositions of a fixed round. We show that swapping two neighboring transitions that do not respect the order in an execution, gives us a legal stutter equivalent execution, i.e., an execution satisfying the same $\mathsf{LTL}_{\text{-}\mathsf{X}}$ properties.

Secondly, in Section 6.5.3 we extend this reasoning to infinite schedules, and lift it from schedules to transition systems. The main idea is to do inductive and compositional reasoning over the rounds. In order to do so, we require that round boundaries are well-defined, which is the case if every round that is started is also finished; a property we can automatically check for fair schedules. In more detail, regarding propositions for one round, we show in Lemma 6.10 that our targeted infinite transition system is stutter equivalent to a one-round transition system. This holds under the assumption that all fair executions of a one-round transition system terminate, and this can be checked using the technique from Chapter 4. As stutter equivalence of systems implies preserving $\mathsf{LTL}_{\text{-}\mathsf{X}}$ properties, this is enough to prove the main goal of the section.

6.5.1 Stutter Equivalence

We restrict our attention to the atomic propositions that describe non-emptiness of the locations different from border locations, that is, from $\mathcal{L} \setminus \mathcal{B}$, in a specific round. Note that these atomic propositions are sufficient for expressing specifications of randomized distributed algorithms[1]. Formally, the set of all such propositions for a round $k \in \mathbb{N}_0$ is

[1]Border locations are redundant copies of initial locations, so they do not play a role in specifications. We introduce them, roughly, in order to model a process that has finished round k, but it has not yet

denoted by

$$\mathrm{AP}_k = \{\mathrm{p}(\ell, k) \colon \ell \in \mathcal{L} \setminus \mathcal{B}\}.$$

For every round k we define a labeling function $\lambda_k \colon \Sigma \to 2^{\mathrm{AP}_k}$ such that

$$\mathrm{p}(\ell, k) \in \lambda_k(\sigma) \quad \text{if and only if} \quad \sigma.\boldsymbol{\kappa}[\ell, k] > 0,$$

i.e., if and only if the location ℓ is nonempty in round k in σ.

For a path $\pi = \sigma_0, t_1, \sigma_1, \ldots, t_n, \sigma_n$, $n \in \mathbb{N}$, and a round k, a trace $\mathrm{trace}_k(\pi)$ w.r.t. the labeling function λ_k is the sequence $\lambda_k(\sigma_0)\lambda_k(\sigma_1)\ldots\lambda_k(\sigma_n)$. Similarly, if a path is infinite $\pi = \sigma_0, t_1, \sigma_1, t_2, \sigma_2, \ldots$, then $\mathrm{trace}_k(\pi) = \lambda_k(\sigma_0)\lambda_k(\sigma_1)\ldots$.

We say that two finite traces are stutter equivalent w.r.t. AP_k, denoted $\mathrm{trace}_k(\pi_1) \triangleq \mathrm{trace}_k(\pi_2)$, if there is a finite sequence $A_0A_1\ldots A_n \in (2^{\mathrm{AP}_k})^+$, $n \in \mathbb{N}_0$, such that both $\mathrm{trace}_k(\pi_1)$ and $\mathrm{trace}_k(\pi_2)$ are contained in the language given by the regular expression $A_0^+A_1^+\ldots A_n^+$. If traces of π_1 and π_2 are infinite, then stutter equivalence $\mathrm{trace}_k(\pi_1) \triangleq \mathrm{trace}_k(\pi_2)$ is defined in the standard way [BK08], as we recall it here. For infinite π_1 and π_2, we have $\mathrm{trace}_k(\pi_1) \triangleq \mathrm{trace}_k(\pi_2)$, if there is an infinite sequence $A_0A_1\ldots$ with $A_i \subseteq \mathrm{AP}_k$, and natural numbers $n_0, n_1, n_2, \ldots, m_0, m_1, m_2 \ldots \geq 1$ such that

$$\mathrm{trace}_k(\pi_1) = \underbrace{A_0\ldots A_0}_{n_0\text{-times}}\underbrace{A_1\ldots A_1}_{n_1\text{-times}}\underbrace{A_2\ldots A_2}_{n_2\text{-times}}\ldots$$

$$\mathrm{trace}_k(\pi_2) = \underbrace{A_0\ldots A_0}_{m_0\text{-times}}\underbrace{A_1\ldots A_1}_{m_1\text{-times}}\underbrace{A_2\ldots A_2}_{m_2\text{-times}}\ldots$$

To simplify notation, we say that paths π_1 and π_2 are stutter equivalent w.r.t. AP_k, and write $\pi_1 \triangleq_k \pi_2$, instead of referring to specific path traces.

Two counter systems C_0 and C_1 are stutter equivalent w.r.t. AP_k, written $C_0 \triangleq_k C_1$, if for every path π from C_i there is a path π' from C_{1-i} such that $\pi \triangleq_k \pi'$, for $i \in \{0, 1\}$.

6.5.2 Reduction to round-rigid schedules

In order to isolate one round, we need to reduce the reasoning to paths in which rounds are ordered in the natural way.

Definition 6.5. *A schedule* $\tau = (r_1, k_1) \cdot (r_2, k_2) \cdot \ldots \cdot (r_m, k_m)$*,* $m \in \mathbb{N}_0$*, is called* round-rigid *if for every* $1 \leq i < j \leq m$*, we have* $k_i \leq k_j$*.*

The following reduction theorem shows that every schedule can be re-ordered into a round-rigid schedule that is, for all rounds k, stutter equivalent regarding $\mathsf{LTL}_{\text{-}\mathsf{X}}$ formulas over propositions of round k.

started round $k + 1$. Nonetheless, the fact that we do not observe their atomic propositions will be crucial for our reduction technique.

Proposition 6.5. *For every configuration σ and every finite schedule τ applicable to σ, there is a round-rigid schedule τ' such that the following holds:*

(a) Schedule τ' is applicable to σ.

(b) τ' and τ reach the same configuration when applied to σ, i.e., $\tau'(\sigma) = \tau(\sigma)$.

(c) For every $k \in \mathbb{N}_0$ we have $\mathsf{path}(\sigma, \tau) \triangleq_k \mathsf{path}(\sigma, \tau')$.

Proof Sketch: Section 6.8.1 is dedicated to the detailed proof of this claim. The idea lies in swapping two adjacent transitions every time the earlier one has the bigger round number. After swapping all such adjacent transitions according to this rule, we obtain a round-rigid schedule, i.e., we order transitions by their round numbers. Note that we do not change the order of transitions that belong to the same round.

We prove the claim by first focusing on two transitions t_1 and t_2. We show that if $t_1.k > t_2.k$, and if $t_1 t_2$ is applicable to a configuration σ, then $t_2 t_1$ is also applicable to σ, and they reach the same configuration, that is, $t_1 t_2(\sigma) = t_2 t_1(\sigma)$. Moreover, we prove that $\mathsf{path}(\sigma, t_1 t_2) \triangleq_k \mathsf{path}(\sigma, t_2 t_1)$. Iteratively applying this result, we obtain the required. □

The following proposition follows from the well-known result that stutter equivalent traces satisfy the same $\mathsf{LTL}_{\text{-}X}$ specifications [BK08, Thm. 7.92].

Proposition 6.6. *Fix a $k \in \mathbb{N}_0$. If π_1 and π_2 are paths such that $\pi_1 \triangleq_k \pi_2$, then for every formula φ from $\mathsf{LTL}_{\text{-}X}$ over AP_k we have $\pi_1 \models \varphi$ if and only if $\pi_2 \models \varphi$.*

We conclude that instead of reasoning about all schedules of $\mathsf{Sys}_\infty(\mathsf{TA})$, it is thus sufficient to reason about its round-rigid schedules. In the following section we will use this to simplify the verification further, namely to a one-round counter system.

6.5.3 From round-rigid schedules to one-round counter system

For each PTA, we define a *round* threshold automaton that can be analyzed with the tools of Chapter 4. Roughly speaking, we focus on one round, but also keep the border locations of the next round, where we add self-loops. We show that for specific fairness constraints, this automaton shows the same behavior as a round in $\mathsf{Sys}_\infty(\mathsf{TA})$. In Theorem 6.11 we prove that we can use it for analysis of non-probabilistic properties of $\mathsf{Sys}_\infty(\mathsf{TA})$.

Restrictions. In the proof we restrict ourselves to fair schedules, that is, those where every transition that is applicable will eventually be performed. We also assume that every fair schedule of a one-round algorithm terminates. Under the fairness assumption we check the latter assumption with ByMC [KW18]. Moreover, we restrict ourselves to deadlock-free threshold automata, that is, we require that in each configuration each

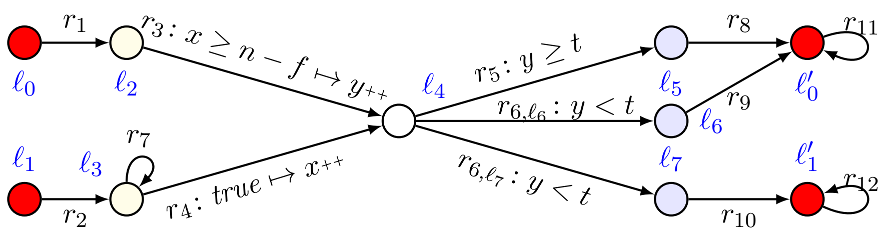


Figure 6.4: The $\mathsf{TA}^{\mathrm{rd}}$ obtained from PTA in Figure 6.2

location has at least one outgoing edge unlocked. As we use TAs to model distributed algorithms, this is no restriction: locations in which no progress should be made unless certain thresholds are reached, typically have self-loops that are guarded with `true`. Thus for our benchmarks one can easily check whether they are deadlock-free using SMT.

Definition 6.6. *Given a $\mathsf{PTA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}, RC)$ or its $\mathsf{TA} = (\mathcal{L}, \mathcal{V}, \mathcal{R}_{np}, RC)$, we define a* round threshold automaton *TA^{rd} to be the tuple $(\mathcal{L} \cup \mathcal{B}', \mathcal{V}, \mathcal{R}^{rd}, RC)$, where $\mathcal{B}' = \{\ell' : \ell \in \mathcal{B}\}$ are copies of border locations, and $\mathcal{R}^{rd}$ is defined as follows. We have $\mathcal{R}^{rd} = (\mathcal{R}_{np} \setminus \mathcal{S}) \cup \mathcal{S}' \cup \mathcal{R}^{loop}$, where modifications $\mathcal{S}'$ of round switch rules are*

$$\mathcal{S}' = \{(from, \ell', \texttt{true}, \mathbf{0}) : (from, \ell, \texttt{true}, \mathbf{0}) \in \mathcal{S} \text{ with } \ell' \in \mathcal{B}'\},$$

and $\mathcal{R}^{loop} = \{(\ell', \ell', \texttt{true}, \mathbf{0}) : \ell' \in \mathcal{B}'\}$ are self-looping rules at locations from $\mathcal{B}'$. Initial locations of TA^{rd} are locations from $\mathcal{B} \subseteq \mathcal{L}$.

For a $\mathsf{TA}^{\mathrm{rd}}$ and a $k \in \mathbb{N}_0$ we define a counter system $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ as the tuple (Σ^k, I^k, R^k): A configuration is a tuple $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p}) \in \Sigma^k$, where $\sigma.\boldsymbol{\kappa} : \mathcal{D} \to \mathbb{N}_0$ defines values of the counters, for $\mathcal{D} = (\mathcal{L} \times \{k\}) \cup (\mathcal{B}' \times \{k+1\})$; and $\sigma.\mathbf{g} : \Gamma \times \{k\} \to \mathbb{N}_0$ defines shared variable values; and $\sigma.\mathbf{p} \in \mathbb{N}_0^{|\Pi|}$ is a vector of parameter values.

Note that by the definition of $\sigma.\boldsymbol{\kappa}$ using $\mathcal{D}$, every configuration $\sigma \in \mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ can be extended to a valid configuration of $\mathsf{Sys}_\infty(\mathsf{TA})$, by assigning values of all other counters and global variables to zero. In the following, we identify a configuration in $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ with its extension in $\mathsf{Sys}_\infty(\mathsf{TA})$, since they have the same labeling function λ_k, for every $k \in \mathbb{N}_0$.

We define $\Sigma^k_{\mathcal{B}} \subseteq \Sigma^k$, for a $k \in \mathbb{N}_0$, to be the set of all configurations σ in which all processes are in border locations of the round k. Formally, we have that $\sigma.\mathbf{g}[x, k] = 0$ for all $x \in \Gamma$, then $\sum_{\ell \in \mathcal{B}} \sigma.\boldsymbol{\kappa}[\ell, k] = N(\mathbf{p})$, and $\sigma.\boldsymbol{\kappa}[\ell, i] = 0$ for all $(\ell, i) \in \mathcal{D} \setminus (\mathcal{B} \times \{k\})$. We call these configurations *border configurations for the round k*. The set of initial states I^k is a subset of $\Sigma^k_{\mathcal{B}}$.

The transition relation R in defined as in $\mathsf{Sys}_\infty(\mathsf{TA})$, i.e., two configurations are in the relation R^k if and only if they (or more precisely, their above described extensions) are in R.

If we do not restrict initial configurations, all these systems are isomorphic, and this is formalized in the following lemma.

Lemma 6.7. *All systems $\mathsf{Sys}^k(\mathsf{TA}^{rd})$, $k \in \mathbb{N}_0$, are isomorphic to each other w.r.t. $\Sigma^k_{\mathcal{B}}$, i.e., for every $k \in \mathbb{N}_0$, if $I^k = \Sigma^k_{\mathcal{B}}$, then we have $\mathsf{Sys}^0(\mathsf{TA}^{rd}) \cong \mathsf{Sys}^k(\mathsf{TA}^{rd})$.*

We restrict our attention to *fair paths*, that is, those paths π such that for every configuration σ in π the following holds: if a transition is applicable, then it will eventually be fired in π. Moreover, we assume that all such paths in $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}})$ terminate, that is, they reach a configuration with all processes in $\mathcal{B}'$. Formally, we assume that for every fair path π in $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}})$ it holds that $\pi \models \mathbf{F} \bigwedge_{\ell\in\mathcal{L}} \boldsymbol{\kappa}[\ell, 0] = 0$. This can easily be checked with ByMC [KW18].

Lemma 6.8. *If all fair executions in $\mathsf{Sys}^0(\mathsf{TA}^{rd})$ terminate w.r.t. $\Sigma^0_{\mathcal{B}}$, then the same holds for $\mathsf{Sys}^k(\mathsf{TA}^{rd})$ w.r.t. $\Sigma^k_{\mathcal{B}}$, for every $k \in \mathbb{N}_0$.*

Proof. It follows directly from Lemma 6.7. □

We assume that in TA (and thus also in $\mathsf{TA}^{\mathrm{rd}}$) holds that for every configuration, every location, and every round, there is an unlocked outgoing edge from ℓ in that configuration and that round. Formally, for every σ, ℓ, and k, there is a rule r such that $r.\mathit{from} = \ell$ and $\sigma, k \models r.\varphi$. This property assures that systems $\mathsf{Sys}_\infty(\mathsf{TA})$ and $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$, $k \in \mathbb{N}_0$, are deadlock-free.

In order to relate $\mathsf{Sys}_\infty(\mathsf{TA})$ and $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$, $k \in \mathbb{N}_0$, we define the set of *initial configurations* I^0 of $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}})$ to be the set I of initial configurations of $\mathsf{Sys}_\infty(\mathsf{TA})$, and then inductively we define $I^{k+1} \subseteq \Sigma^{k+1}_{\mathcal{B}}$ to be the set of final configurations of $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$.

From now on, we fix a TA and a $\mathsf{TA}^{\mathrm{rd}}$, and if not specified differently, for every $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ we assume the above definition of I^k.

Lemma 6.9. *If all fair executions of $\mathsf{Sys}^0(\mathsf{TA}^{rd})$ w.r.t. $\Sigma^0_{\mathcal{B}}$ terminate, then for every $k \in \mathbb{N}_0$ we have that the set I^k is well-defined and all fair executions of $\mathsf{Sys}^k(\mathsf{TA}^{rd})$ terminate (w.r.t. I^k).*

Proof. We prove this claim by induction on $k \in \mathbb{N}_0$. The set $I^0 = I$ is clearly well-defined, and since $I^0 \subseteq \Sigma^0_{\mathcal{B}}$, by our assumption we have that all fair executions of $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}})$ terminate. Since for every $k \in \mathbb{N}_0$ we have $I^k \subseteq \Sigma^k_{\mathcal{B}}$, by Lemma 6.8 we have that every fair execution of $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ terminates and therefore I^{k+1} is well-defined. □

Lemma 6.10. *If $\mathsf{Sys}_\infty(\mathsf{TA})$ is deadlock-free, and if all fair executions of $\mathsf{Sys}^0(\mathsf{TA}^{rd})$ w.r.t. $\Sigma^0_{\mathcal{B}}$ terminate, then for every $k \in \mathbb{N}_0$ we have*

$$\mathsf{Sys}^k(\mathsf{TA}^{rd}) \triangleq_k \mathsf{Sys}_\infty(\mathsf{TA}),$$

i.e., the two systems are stutter equivalent w.r.t. AP_k.

Proof Sketch: The detailed proof of this statement is given in Section 6.8.2. It is based on induction on the round $k \in \mathbb{N}_0$, and every instance consists of two directions.

Given a path $\pi = \mathsf{path}(\sigma_k, \tau)$ from $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ we find a stutter equivalent path π' from $\mathsf{Sys}_\infty(\mathsf{TA})$ by (i) appending a prefix that reaches σ_k, which is possible by the definition of I^k, and (ii) extending π by an arbitrary path from $\tau(\sigma_k)$, that exists since there are no deadlocks.

For the other direction, given a path π from $\mathsf{Sys}_\infty(\mathsf{TA})$, we construct a stutter equivalent path π' from $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ by cutting out all transitions of different round. If necessary, we add selfloops to imitate an infinite path. $\square$

By Lemma 6.7, for every $k \in \mathbb{N}_0$ and every $\sigma \in \Sigma^k_{\mathcal{B}}$, there is a corresponding configuration $\sigma' \in \Sigma^0_{\mathcal{B}}$ obtained from σ by renaming the round k to 0. Let f_k be the renaming function, i.e., $\sigma' = f_k(\sigma)$. Let us define $\Sigma^u \subseteq \Sigma^0_{\mathcal{B}}$ to be the union of all renamed initial configurations $\{f_k(\sigma) \colon k \in \mathbb{N}_0, \sigma \in I^k\}$.

Theorem 6.11. *Let system $\mathsf{Sys}_\infty(\mathsf{TA})$ be deadlock-free, and let all fair executions of $\mathsf{Sys}^0(\mathsf{TA}^{rd})$ w.r.t. $\Sigma^0_{\mathcal{B}}$ terminate. Given a formula $\varphi[i]$ from $\mathsf{LTL}_{\text{-}X}$ over AP_i, for a round variable i, the following statements are equivalent:*

(a) $\mathsf{Sys}^0(\mathsf{TA}^{rd}) \models \mathbf{E}\,\varphi[0]$ w.r.t. initial configurations Σ^u

(b) there exists a $k \in \mathbb{N}_0$ such that $\mathsf{Sys}_\infty(\mathsf{TA}) \models \mathbf{E}\,\varphi[k]$.

Proof. Let us first assume that $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}}) \models \mathbf{E}\,\varphi[0]$ w.r.t. initial configurations Σ^u. This means there is a path $\pi = \mathsf{path}(\sigma, \tau)$ such that $\sigma \in \Sigma^u$ and $\pi \models \varphi[0]$. Since $\sigma \in \Sigma^u$, there is a $k \in \mathbb{N}_0$ and a $\sigma_k \in I^k$ such that $\sigma = f_k(\sigma_k)$. From Lemma 6.7 we know that $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}}) \cong \mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$, and thus there is a schedule τ_k in $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ such that $\mathsf{path}(\sigma_k, \tau_k) \models \varphi[k]$. Now Lemma 6.10 tells us that there must be a path π' from $\mathsf{Sys}_\infty(\mathsf{TA})$ such that $\mathsf{path}(\sigma_k, \tau_k) \triangleq_k \pi'$. By Proposition 6.6 we know that $\pi' \models \varphi[k]$, and thus $\mathsf{Sys}_\infty(\mathsf{TA}) \models \mathbf{E}\,\varphi[k]$. This proves one direction of the statement.

Assume now that there is a $k \in \mathbb{N}_0$ such that $\mathsf{Sys}_\infty(\mathsf{TA}) \models \mathbf{E}\,\varphi[k]$. Thus, there is a path $\pi = \mathsf{path}(\sigma, \tau)$ in $\mathsf{Sys}_\infty(\mathsf{TA})$ such that $\pi \models \varphi[k]$. By Lemma 6.10 we know that there is a path $\pi' = \mathsf{path}(\sigma', \tau')$ in $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ with $\pi \triangleq_k \pi'$, and then by Proposition 6.6 also $\pi' \models \varphi[k]$. Finally, by Lemma 6.7 there is an equivalent path π_0 in $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}})$ starting in $f_k(\sigma')$. Then we have that $\pi_0 \models \varphi[0]$, and since $f_k(\sigma') \in \Sigma^u$, we know that $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}}) \models \mathbf{E}\,\varphi[0]$ w.r.t. initial configurations Σ^u. This concludes the other direction of the proof. $\square$

Bringing it all together. In Section 6.4 we show how to reduce our specifications to formulas of the following form:

$$(\forall k \in \mathbb{N}_0)\ \mathbf{A}\,\psi[k].$$

Theorem 6.11 deals with negations of such forms, namely with existence of a round k such that formula $\mathsf{E}\,\varphi[k]$ holds. Therefore, the theorem allows us to check on the one-round system instead of on the infinite one, if there is a counterexample to formulas we want to check.

For instance, if we want to check whether two processes disagree in a round, we would need to search through all (unboundedly many) rounds. Instead, we search for a disagreement only in the first round with respect to all possible initial configurations. If there is no disagreement in the first round, Theorem 6.11 confirms that there is no disagreement in any other round.

6.6 Probabilistic Wait-Free Termination

6.6.1 Adversaries

In randomized distributed algorithms, some properties are required to hold with at least a certain probability. This in general means that they are supposed to hold for every initial configuration and failure pattern. We group executions that differ only in random choices, and analyze each group. This is done using adversaries. They are typically considered as fictitious entities that control the choice of the next transition in a system, but if the transition is probabilistic (an action), they do not have any influence on the probabilistic choice. Thus, for every adversary and every initial configuration we obtain a unique execution tree, where branching appears due to probabilistic choices.

Definition 6.7. *Let Paths be the set of all finite paths in $\mathsf{Sys}(\mathsf{PTA})$. An* adversary *is a function $\mathtt{s} : \mathsf{Paths} \to \mathsf{Act}$, that given a path π selects an action applicable to the last configuration of π.*

Given a configuration σ and an adversary $\mathtt{s}$, we generate a family of infinite paths, depending of the outcomes of non-Dirac transitions. We denote this set by $\mathsf{AdvPaths}(\sigma, \mathtt{s})$. An adversary $\mathtt{s}$ is *fair* if all paths in $\mathsf{AdvPaths}(\sigma, \mathtt{s})$ are fair.

As usual, the MDP $\mathsf{Sys}(\mathsf{PTA})$ together with an initial configuration σ and an adversary $\mathtt{s}$ induce a Markov chain, written $M^{\sigma}_{\mathtt{s}}$. In the sequel, we write $\mathbb{P}^{\sigma}_{\mathtt{s}}$ for the probability measure over infinite paths starting at σ in the latter Markov chain.

Definition 6.8. *An adversary $\mathtt{s}$ is* round-rigid *if it is fair, and if every sequence of actions it produces can be written as $\mathtt{s}_0 \cdot \mathtt{s}^p_0 \cdot \mathtt{s}_1 \cdot \mathtt{s}^p_1 \ldots$, where for every $k \in \mathbb{N}_0$, we have that $\mathtt{s}_k$ contains only Dirac transitions of round k, and $\mathtt{s}^p_k$ contains only non-Dirac transitions of round k.*

We denote the set of all round-rigid adversaries by $\mathcal{A}^R$.

6.6.2 Sufficient conditions for Probabilistic wait-free termination

Let us first recall Probabilistic wait-free termination from (6.4):

$$\mathbb{P}_{\mathsf{s}}\big(\bigvee_{k\in\mathbb{N}_0} \bigvee_{v\in\{0,1\}} \mathbf{G} \bigwedge_{\ell\in\mathcal{F}\setminus\mathcal{D}_v} \boldsymbol{\kappa}[\ell,k]=0\big) \quad = \quad 1.$$

We start by introducing two sufficient conditions for Probabilistic Wait-Free Termination under round-rigid adversaries. This is formalized in Theorem 6.12.

One condition is the existence of a positive probability lower-bound for a certain event to happen, that is, Theorem 6.12(a). In order to verify it, we reduce it in two steps to the form that ByMC is able to check. First, in Section 6.6.3 we reduce it to a non-probabilistic condition, which is not in the multi-round $\mathsf{ELTL}_{\mathsf{FT}}$ logic, that is, does not have the form from (6.1). Second, in Section 6.6.4 we modify the system to a non-probabilistic one, and modify the condition to the convenient form $(\forall k\in\mathbb{N}_0)\ \mathbf{A}\,\varphi[k]$ on the new system.

The other condition, Theorem 6.12(b), already has the correct form, that is, it is in multi-round $\mathsf{ELTL}_{\mathsf{FT}}$ with one quantifier. Thus, it can be checked with ByMC without modifying the system and the formula.

Theorem 6.12. *Let $\mathbf{p}\in\mathbf{P}_{RC}$ be a vector of admissible parameters in $\mathit{Sys}(\mathit{PTA})$.*

(a) Assume there is a bound $p\in(0,1]$, such that for every initial configuration σ with parameters $\mathbf{p}$, every round-rigid adversary s, and every $k\in\mathbb{N}_0$ holds

$$\mathbb{P}^{\sigma}_{\mathsf{s}}\big(\bigvee_{v\in\{0,1\}} \boldsymbol{G}\big(\bigwedge_{\ell\in\mathcal{F}_v} \boldsymbol{\kappa}[\ell,k]=0\big)\big) > p. \qquad (6.13)$$

(b) Assume for every $v\in\{0,1\}$, holds that

$$(\forall k\in\mathbb{N}_0)\ \boldsymbol{A}\,\big(\boldsymbol{G} \bigwedge_{\ell\in\mathcal{I}_{1-v}} \boldsymbol{\kappa}[\ell,k]=0 \ \rightarrow\ \boldsymbol{G} \bigwedge_{\ell'\in\mathcal{F}\setminus\mathcal{D}_v} \boldsymbol{\kappa}[\ell',k]=0\big). \qquad (6.14)$$

Then we have Probabilistic Wait-Free Termination, that is, formula (6.4) holds.

Proof. Fix a $\mathbf{p}\in\mathbf{P}_{RC}$, an initial configuration σ_0, and a round-rigid adversary s. Assume there is a non-zero probability p such that from any initial configuration σ over $\mathbf{p}$ and under any round-rigid adversary s formula (6.13) holds.

Two options may occur along a path $\pi\in\mathsf{AdvPaths}(\sigma_0,\mathsf{s})$: (i) either round 0 ends with a final configuration in which all processes have the same value, say v, or (ii) round 0 ends with a final configuration with both values present.

(i) In this case we have that $\pi\models\mathbf{G}\,(\bigwedge_{\ell\in\mathcal{F}_{1-v}}\boldsymbol{\kappa}[\ell,0]=0)$, and by our assumption, *i.e.*, formula (6.13) for $k=0$, the probability that this case happens is at least p. Then, by Lemma 6.2 we also have $\pi\models\mathbf{G}\,(\bigwedge_{\ell\in\mathcal{I}_{1-v}}\boldsymbol{\kappa}[\ell,1]=0)$. By formula (6.14), in this case all processes decide value v in round 1.

(ii) The probability that the second case happens is at most $1-p$. In this case, round 1 starts with an initial configuration σ_1 with both initial values 0 and 1. From σ_1 under s, by the same reasoning as from σ_0, at the end on the round 1 we have the analogous two cases, and all processes decide in round 2 with probability at least p.

Iterating this reasoning, almost surely all processes eventually decide. Let us formally explain this iteration. Let σ_0 be an initial configuration, and let s be a round-rigid adversary. For a $k \in \mathbb{N}$, consider the event $\mathcal{E}_k$: from σ_0 and under s, not every process decides in the first k rounds. In particular, at the end of every round $i < k$ it is not the case that everyone decides. By the reasoning above, namely case (ii) for round i, this happens with probability at most $(1-p)$. Therefore, for k rounds we have $\mathbb{P}_{\mathsf{s}}^{\sigma}(\mathcal{E}_k) \leq (1-p)^k$. The limit when k tends to infinity yields that the probability for not having Probabilistic Wait-Free Termination is 0. This is equivalent to the required formula (6.4). □

6.6.3 Reduction to one-round non-probabilistic specification

We have previously used the idea of decomposing an infinite counter system $\mathsf{Sys}_\infty(\mathsf{TA})$ into one-round systems $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$. In $\mathsf{Sys}(\mathsf{PTA})$, this idea is even more straightforward since we consider only round-rigid adversaries. They guarantee that rounds are ordered in all generated paths.

Since the assumption (a) in Theorem 6.12 is a property of one-round k in the whole system $\mathsf{Sys}(\mathsf{PTA})$, we introduce analogous objects as in the non-probabilistic case. Namely, we introduce a $\mathsf{PTA}^{\mathrm{rd}}$ (analogously as in Definition 6.6), and its counter system $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$, for every $k \in \mathbb{N}_0$. Initial configurations of $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$ are denoted by I^k (as they do coincide, similarly to the non-probabilistic case). Moreover, for a fixed vector of parameters $\mathbf{p}$, the set of configurations σ from I^k with $\sigma.\mathbf{p} = \mathbf{p}$ is denoted by $I^k_{\mathbf{p}}$.

Now, instead of checking that Theorem 6.12(a) holds on $\mathsf{Sys}(\mathsf{PTA})$, since the property itself refers only to one round k, it can be checked on the one-round system $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$ as in Lemma 6.13(a).

In the following lemma we explain how to use non-probabilistic reasoning in order to prove a probabilistic property.

Lemma 6.13. *Let $k \in \mathbb{N}_0$ and let $\mathbf{p} \in \mathbf{P}_{RC}$ be a parameter valuation in $\mathsf{Sys}^k(\mathsf{PTA}^{rd})$. For every LTL formula $\varphi[k]$ over atomic propositions AP_k of round k, the following three formulas are equivalent:*

(a) $(\exists p > 0)\ (\forall \sigma \in I^k_{\mathbf{p}})\ (\forall \mathsf{s} \in \mathcal{A}^R)\quad \mathbb{P}_{\mathsf{s}}^{\sigma}(\varphi[k]) > p$,

(b) $(\forall \sigma \in I^k_{\mathbf{p}})\ (\forall \mathsf{s} \in \mathcal{A}^R)\quad \mathbb{P}_{\mathsf{s}}^{\sigma}(\varphi[k]) > 0$,

(c) $(\forall \sigma \in I^k_{\mathbf{p}})\ (\forall \mathsf{s} \in \mathcal{A}^R)\ (\exists \pi \in \mathit{AdvPaths}(\sigma, \mathsf{s}))\ \pi \models \varphi[k]$.

Proof. Fix parameters $\mathbf{p} \in \mathbf{P}_{RC}$.

The two implications from top to bottom are trivial: if a probability is lower bounded by a positive constant, then it is positive; and if the probability of a given property is positive, then there must be at least a path satisfying that property. It is thus sufficient to prove that the last statement implies the first one to obtain all equivalences.

Assume that from every initial configuration σ with parameter values $\mathbf{p}$, and under every round-rigid adversary s, there exists a path $\pi \in \mathsf{AdvPaths}(\sigma, \mathsf{s})$ such that $\pi \models \varphi[k]$. Observe that, independently of σ and s, since there are no cycles in PTA, and since PTA contains a fixed number c of non-Dirac transitions, we have that for any path, the prefix corresponding to round k has at most $c \cdot N(\mathbf{p})$ non-Dirac transitions, where $N(\mathbf{p})$ is the number of modeled processes. The probability of the set of all infinite paths that have the same prefix of round k as π, is thus at least $2^{-c \cdot N(\mathbf{p})}$. Therefore, we can define a positive probability $p = 2^{-c \cdot N(\mathbf{p})}$ to be our lower bound, which only depends on PTA and $\mathbf{p}$ such that, in $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$we have that $(\exists p > 0)(\forall \sigma \in I_{\mathbf{p}}^k)(\forall \mathsf{s} \in \mathcal{A}^{\mathrm{R}})\ \mathbb{P}_{\mathsf{s}}^{\sigma}(\varphi[k]) > p$. This concludes the proof. □

The following Corollary shows how to apply Lemma 6.13 to the property we need in Theorem 6.12.

Corollary 6.14. *The assumption (a) from Theorem 6.12 is equivalent to the following: for every $k \in \mathbb{N}_0$ in $\mathsf{Sys}^k(\mathsf{PTA}^{rd})$ holds that for every vector of parameters $\mathbf{p} \in \mathbf{P}_{RC}$, for every initial configuration $\sigma \in I_{\mathbf{p}}^k$, and for every round-rigid adversary s, there is a path $\pi \in \mathsf{AdvPaths}(\sigma, \mathsf{s})$ such that*

$$\pi \models \bigvee_{v \in \{0,1\}} \mathbf{G}\Big(\bigwedge_{\ell \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell, k] = 0\Big).$$

6.6.4 Reduction to one-round non-probabilistic systems

According to Corollary 6.14, in order to prove assumption (a) from Theorem 6.12, we only need to prove that for every $k \in \mathbb{N}_0$ in the system $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$ it holds that

$$(\forall \sigma \in I_{\mathbf{p}}^k)\ (\forall \mathsf{s} \in \mathcal{A}^{\mathrm{R}})\ (\exists \pi \in \mathsf{AdvPaths}(\sigma, \mathsf{s}))\quad \pi \models \bigvee_{v \in \{0,1\}} \mathbf{G}\Big(\bigwedge_{\ell \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell, k] = 0\Big). \tag{6.15}$$

Proving this on the one-round system $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$ is not immediate. Indeed, the quantifier alternation (universal over initial configurations and adversaries vs. existential on paths) makes its automated verification out-of-reach for the technique from Chapter 4. We therefore transform $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$ to $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{m}})$ in order to avoid the quantifier alternation which is due to the presence of probabilistic rules. Intuitively, paths are stopped before the series of non-Dirac transitions happen. Doing so, from a configuration σ, an adversary s yields a unique path, written $\mathsf{path}(\sigma, \mathsf{s})$.

Given a PTA, we define a threshold automaton TA^{m} such that for every non-Dirac rule $r = (\mathit{from}, \delta_{to}, g, \mathbf{u})$ in PTA, all locations ℓ with $\delta_{to}(\ell) > 0$ are merged into a new

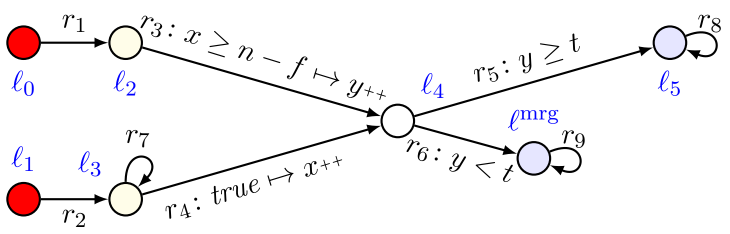


Figure 6.5: A one-round non-probabilistic threshold automaton TA^{m} obtained from the PTA from Figure 6.2.

location ℓ^{mrg} in TA^{m}. Note that this location must belong to the set of final locations $\mathcal{F}$. Naturally, instead of a non-Dirac rule r we obtain a Dirac rule $(from, \ell^{\mathrm{mrg}}, g, \mathbf{u})$. Also we add self-loops at all final locations. The counter system $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{m}})$ is defined in the standard non-probabilistic way.

Figure 6.5 illustrates the transformation on our running example from Figure 6.2. The new final location ℓ^{mrg} represents a coin toss taking place; it belongs neither to $\mathcal{F}_0$ nor $\mathcal{F}_1$ (but it is in $\mathcal{F}$).

Initial configurations in $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$ coincide with initial configurations in $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{m}})$. This exploits our definition of round-rigid adversaries, where all non-Dirac transitions are gathered at the end of a round.

Lemma 6.15. *Fix $k \in \mathbb{N}_0$, an initial configuration σ from $\mathsf{Sys}^k(\mathsf{PTA}^{rd})$, and a round-rigid adversary s. For every LTL formula $\varphi[k]$, the statements are equivalent:*

(a) there exists $\pi \in \mathsf{AdvPaths}(\sigma, \mathsf{s})$ such that $\pi \models \varphi[k]$ in $\mathsf{Sys}^k(\mathsf{PTA}^{rd})$,

(b) for every $\pi \in \mathsf{AdvPaths}(\sigma, \mathsf{s})$ holds $\pi \models \varphi[k]$ in $\mathsf{Sys}^k(\mathsf{TA}^{m})$.

Proof. Paths in $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{m}})$ are prefixes of paths in $\mathsf{Sys}^k(\mathsf{PTA}^{\mathrm{rd}})$. Moreover, as all the forks are removed from the threshold automaton, an adversary does not induce a tree, but only a single path, since there is no branching. Thus, since every set of paths $\mathsf{AdvPaths}(\sigma, \mathsf{s})$ in $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{m}})$ is a singleton, then existential and universal quantification coincide. □

Finally, let us return to Theorem 6.12, now when we have reduced the condition ((a)) to the convenient form. Now, proving Probabilistic Wait-Free Termination under all round-rigid adversaries boils down to proving formula (6.14) on $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ and also that

$$\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{m}}) \models \mathbf{A} \bigvee_{v \in \{0,1\}} \mathbf{G} \bigwedge_{\ell \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell, k] = 0.$$

Both can be checked using the technique of Chapter 4, with ByMC. Formally, from Corollary 6.14 and Lemma 6.15 we obtain the following.

Corollary 6.16. *If $\mathsf{Sys}^k(\mathsf{TA}^{rd}) \models (6.14)$, and if*

$$\mathsf{Sys}^k(\mathsf{TA}^m) \models \boldsymbol{A} \bigvee_{v \in \{0,1\}} \boldsymbol{G} \bigwedge_{\ell \in \mathcal{F}_v} \boldsymbol{\kappa}[\ell, k] = 0,$$

then Probabilistic Wait-Free Termination holds on $\mathsf{Sys}(\mathsf{PTA})$.

6.7 Experiments

We have applied the approach presented in Sections 6.4–6.6 to five randomized fault-tolerant distributed algorithms [2]:

1. Protocol 1 for randomized consensus by Ben-Or [Ben83]. We consider two kinds of crashes: clean crashes (ben-or-cc) and dirty crashes (ben-or-dc). During a dirty crash a process can send to a subset of processes, while in clean crashes a process is either sending to all proceses or none. This algorithm works correctly when $n > 2t$.

2. Protocol 2 for randomized Byzantine consensus (ben-or-byz) by Ben-Or [Ben83]. This algorithm tolerates Byzantine faults when $n > 5t$.

3. Protocol 2 for randomized consensus (rabc-c) by Bracha [Bra87]. It runs as a high-level algorithm together with a low-level broadcast that turns Byzantine faults into "little more than fail-stop (faults)". We check only the high-level algorithm for clean crashes. Our model checker produces counterexamples when Byzantine or Byzantine-symmetric faults are introduced in rabc-c. The multi-layered protocol is designed for $f < n/3$ faults. However, our tool shows that rabc-c itself tolerates $f < n/2$ clean crashes.

4. k-set agreement for crash faults (kset) by Raynal [MMR18], for $k = 2$. This algorithm works in presense of clean crashes when $n > 3t$.

5. Randomized asynchronous Byzantine one-step consensus (rs-bosco) by Song and van Renesse [SvR08]. This algorithm tolerates Byzantine faults when $n > 3t$, and it terminates fast when $n > 7t$ or $n > 5t$ and $f = 0$.

Following the reduction approach of Sections 6.4–6.6, for each benchmark, we have encoded two versions of one-round threshold automata: an N-automaton that models a coin toss by a non-deterministic choice, and a P-automaton that never leaves the coin-toss location, once it entered this location. The N-automaton is used to support the non-probabilistic reasoning, while the P-automaton is used to prove probabilistic wait-free termination. Both automata are given as the input to Byzantine Model Checker (ByMC) [KW18], which implements the parameterized model checking techniques for

[2]The benchmarks and the instructions on running the experiments are available from: `https://forsyte.at/software/bymc/artifact82/`

Label	Name	Automaton	Formula
S1	`agreement_0`	N	$\mathbf{A}\,\mathbf{G}\,(\neg\textsc{Ex}\{\mathsf{D0}\}) \vee \mathbf{G}\,(\neg\textsc{Ex}\{\mathsf{D1},\mathsf{E1}\})$
S2	`validity_0`	N	$\mathbf{A}\,\textsc{All}\{\mathsf{V0}\} \to \mathbf{G}\,(\neg\textsc{Ex}\{\mathsf{D1},\mathsf{E1}\})$
S3	`completeness_0`	N	$\mathbf{A}\,\textsc{All}\{\mathsf{V0}\} \to \mathbf{G}\,(\neg\textsc{Ex}\{\mathsf{D1},\mathsf{E1}\})$
S4	`round-term`	N	$\mathbf{A}\,\mathit{fair} \to \mathbf{F}\,\textsc{All}\{\mathsf{D0},\mathsf{D1},\mathsf{E0},\mathsf{E1},\mathsf{CT}\}$
S5	`decide-or-flip`	P	$\mathbf{A}\,\mathit{fair} \to \mathbf{F}\,(\textsc{All}\{\mathsf{D0},\mathsf{E0},\mathsf{CT}\} \vee \textsc{All}\{\mathsf{D1},\mathsf{E1},\mathsf{CT}\})$
S1'	`sim-agreement`	N	$\mathbf{A}\,\mathbf{G}\,(\neg\textsc{Ex}\{\mathsf{D0},\mathsf{E0}\} \vee \neg\textsc{Ex}\{\mathsf{D1},\mathsf{E1}\})$
S1"	`2-agreement`	N	$\mathbf{A}\,\mathbf{G}\,(\neg\textsc{Ex}\{\mathsf{D0},\mathsf{E0}\} \vee \neg\textsc{Ex}\{\mathsf{D1},\mathsf{E1}\} \vee \neg\textsc{Ex}\{\mathsf{D2},\mathsf{E2}\})$

Table 6.1: Temporal properties verified in our experiments for value 0 (the properties for value 1 can be obtained by swapping 0 and 1). We write fairness constraints as *fair* to save space.

safety and liveness of counter systems of threshold-automata (for a bounded number of rounds and no randomization), as in Chapter 4.

The automata follow the pattern shown in Figure 6.2: They start in one of the initial locations (e.g., V_0 or V_1), progress by switching locations and incrementing shared variables and end up in a final location that corresponds to a decision (e.g., D_0 or D_1), an estimate of a decision (e.g., E_0 or E_1), or a coin toss (CT).

Table 6.1 summarizes the temporal properties that were verified in our experiments. Given the set of all possible locations $\mathcal{L}$, a set $Y = \{\ell_1, \dots, \ell_m\} \subseteq \mathcal{L}$ of locations, and the designated crashed location $\mathsf{CR} \in \mathcal{L}$, we use the shorthand notation: $\textsc{Ex}\{\ell_1, \dots, \ell_\mathsf{m}\}$ for $\bigvee_{\ell \in Y} \boldsymbol{\kappa}[\ell] \neq 0$ and $\textsc{All}\{\ell_1, \dots, \ell_\mathsf{m}\}$ for $\bigwedge_{\ell \in \mathcal{L} \setminus Y} (\boldsymbol{\kappa}[\ell] = 0 \vee \ell = \mathsf{CR})$. For rs-bosco and kset, instead of checking S1, we check S1' and S1".

Table 6.2 presents the computational results of the experiments reported in [BKLW18]. The meaning of the columns is as follows: column $|\mathcal{L}|$ shows the number of automata locations, column $|\mathcal{R}|$ shows the number of automata rules, column $|\mathcal{S}|$ shows the number of enumerated schemas (which depends on the structure of the automaton and the specification), column *time* shows the computation times — either in seconds or in the format `HH:MM:SS`. For $|\mathcal{R}|$, we give the figures for the N-automata, since they have more rules in addition to the rules in P-automata. To save space, we omit the figures for memory use from the table: Benchmarks 1–5 need 30–170 MB RAM, whereas rs-bosco needs up to 1.5 GB RAM per cluster node.

Recall that the benchmark rs-bosco presents a challenge for the schema enumeration technique of Chapter 4: Its threshold automaton contains 12 threshold guards that can change their values almost in any order. Additional combinations are produced by the temporal formulas. ByMC reduces the number of combinations by analyzing dependencies between the guards. However, this benchmark requires us to enumerate between 11! and 14! schemas. To this end, we have run the verification experiments for rs-bosco on 1024 CPU cores of the computing cluster Grid5000. Table 6.2 presents the wall time results for rs-bosco, that is, the actual number of computation hours on all the

Automaton				**S1/S1'/S1"**		**S2**		**S3**		**S4**		**S5**	
#	**Name**	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{S}\rvert$	**Time**	$\lvert\mathcal{S}\rvert$	**Time**	$\lvert\mathcal{S}\rvert$	**Time**	$\lvert\mathcal{S}\rvert$	**Time**	$\lvert\mathcal{S}\rvert$	**Time**
1	**ben-or-cc**	10	27	9	1	5	0	5	0	5	0	5	0
2	**ben-or-dc**	10	32	9	1	5	1	5	0	5	0	5	1
3	**ben-or-byz**	9	18	3	1	2	0	2	0	2	0	2	1
4	**rabc-cr**	11	31	9	0	5	1	5	1	5	0	5	0
5	**kset**	13	58	65	3	65	17	65	12	65	39	65	40
6	**rs-bosco**	19	48	156M	3:21:00	156M	3:02:00	156M	3:21:00	n/a	n/a	156M	3:43:12

Table 6.2: The experiments for rows 1-5 were run on a single computer (Apple MacBook Pro 2018, 16GB). The experiments for row 6 (rs-bosco) were run in Grid5000 on 32 nodes (2 CPUs Intel Xeon Gold 6130, 16 cores/CPU, 192GB). Wall times are given.

cores is the wall time multiplied by 1024.

For all the benchmarks in Table 6.2, ByMC has reported that the specifications hold. By varying the relations between the parameters (e.g., by changing $n > 3t$ to $n > 2t$), we have found that rabc-cr can handle more faults, that is, $t < n/2$ in contrast to the original $t < n/3$ (the original was needed to implement the underlying communication structure which we assume given in the experiments). In other cases, whenever we changed the parameters, that is, increased the number of faults beyond the known bound, the tool reported a counterexample.

6.8 Detailed Proofs for Section 6.5

6.8.1 Detailed Proofs for Section 6.5.2

The following lemma follows directly from the definitions of transitions, and it gives us the most important transition invariants.

Lemma 6.17. *Let σ be a configuration and let $t = (r, k)$ be a transition. If $\sigma' = t(\sigma)$ then the following holds:*

(a) $\sigma'.\mathbf{g}[k'] = \sigma.\mathbf{g}[k']$, *for every round* $k' \neq k$,

(b) $\sigma'.\boldsymbol{\kappa}[k'] = \sigma.\boldsymbol{\kappa}[k']$, *for every round* $k' \in \mathbb{N}_0 \setminus \{k, k+1\}$,

(c) $\sigma'.\boldsymbol{\kappa}[\ell, k'] = \sigma.\boldsymbol{\kappa}[\ell, k']$, *for every round* $k' \neq k$ *and every location* $\ell \in \mathcal{L} \setminus \mathcal{B}$,

(d) $\sigma'.\boldsymbol{\kappa}[k+1] \geq \sigma.\boldsymbol{\kappa}[k+1]$,

The following lemma establishes a central argument for inductive round-based reasoning: a transition belonging to a smaller round can always be moved before a transition of the larger round.

Lemma 6.18. *Let σ be a configuration, and let $t_1 = (r_1, k_1)$ and $t_2 = (r_2, k_2)$ be transitions, such that $k_1 > k_2$. If $t_1 \cdot t_2$ is applicable to σ, then $t_2 \cdot t_1$ is also applicable to σ.*

Proof. Let us denote $t_1(\sigma)$ by σ_1. As $t_1 \cdot t_2$ is applicable to σ, this means that t_1 is applicable to σ and t_2 is applicable to σ_1. By definition of applicability, this means that

$$\sigma.\boldsymbol{\kappa}[r_1.\mathit{from}, k_1] \geq 1 \quad \text{and} \quad \sigma_1.\boldsymbol{\kappa}[r_2.\mathit{from}, k_2] \geq 1, \tag{6.16}$$

and additionally we have that $\sigma, k_1 \models t_1.\varphi$ and $\sigma_1, k_2 \models t_2.\varphi$.

We show that $t_2 \cdot t_1$ is applicable to σ by showing that: (i) t_2 is applicable to σ, and (ii) t_1 is applicable to $t_2(\sigma)$.

(i) First we need to show that $\sigma.\boldsymbol{\kappa}[r_2.\mathit{from}, k_2] \geq 1$ and $\sigma, k_2 \models t_2.\varphi$.

As $\sigma_1 = t_1(\sigma)$ and $k_2 < k_1$, by Lemma 6.17(b) we have $\sigma_1.\boldsymbol{\kappa}[r_2.\mathit{from}, k_2] = \sigma.\boldsymbol{\kappa}[r_2.\mathit{from}, k_2]$. From this and (6.16) we get that $\sigma.\boldsymbol{\kappa}[r_2.\mathit{from}, k_2] \geq 1$.

Note that evaluation of the guard $t_2.\varphi$ depends only on the values of shared variables $\sigma.\mathbf{g}[k_2]$ in round k_2 and parameter values $\sigma.\mathbf{p}$. As $\sigma_1 = t_1(\sigma)$ and $k_1 > k_2$, from Lemma 6.17(a) we have that $\sigma.\mathbf{g}[k_2] = \sigma_1.\mathbf{g}[k_2]$. Recall that $\sigma_1, k_2 \models t_2.\varphi$, and thus it must be the case that also $\sigma, k_2 \models t_2.\varphi$. This shows that t_2 is applicable to σ.

(ii) Let $\sigma_2 = t_2(\sigma)$. Next we show that t_1 is applicable to σ_2. Using the same reasoning as in (i), we prove that $\sigma_2.\boldsymbol{\kappa}[r_1.\mathit{from}, k_1] \geq 1$ and that $\sigma_2, k_1 \models t_1.\varphi$.

As $\sigma_2 = t_2(\sigma)$ and $k_2 < k_1$, Lemma 6.17(b) and 6.17(d) yield $\sigma_2.\boldsymbol{\kappa}[r_1.\mathit{from}, k_1] \geq \sigma.\boldsymbol{\kappa}[r_1.\mathit{from}, k_1]$. Together with (6.16) we obtain that $\sigma_2.\boldsymbol{\kappa}[r_1.\mathit{from}, k_1] \geq 1$.

To this end, we show that $\sigma_2, k_1 \models t_1.\varphi$. Since $\sigma_2 = t_2(\sigma)$ and $k_1 > k_2$, by Lemma 6.17(a) we know that $\sigma.\mathbf{g}[k_1] = \sigma_2.\mathbf{g}[k_1]$. Since by the initial assumption we have $\sigma, k_1 \models t_1.\varphi$, and evaluation of the guard only depends on shared variable values and parameter values, then it also holds $\sigma_2, k_1 \models t_1.\varphi$. ☐

Lemma 6.19. *Let σ be a configuration, let $t_1 = (r_1, k_1)$ and $t_2 = (r_2, k_2)$ be transitions such that $k_1 > k_2$. If $t_1 \cdot t_2$ is applicable to σ, then the following holds:*

(a) Both $t_1 \cdot t_2$ and $t_2 \cdot t_1$ reach the same configuration, i.e., $t_1 \cdot t_2(\sigma) = t_2 \cdot t_1(\sigma)$.

(b) For every $k \in \mathbb{N}_0$ we have $\mathsf{path}(\sigma, t_1 \cdot t_2) \triangleq_k \mathsf{path}(\sigma, t_2 \cdot t_1)$.

Proof. Note that since $t_1 \cdot t_2$ is applicable to σ, we also have that $t_2 \cdot t_1$ is applicable to σ by Lemma 6.18, since $k_1 > k_2$.

(a) When a transition is applied to a configuration, the obtained configuration has the same parameter values, and counters and global variables are incremented or decremented depending on the transition (and independently of the initial configuration). For any configuration $(\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$, we can write $t_i(\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p}) = (\boldsymbol{\kappa}+\mathbf{u}_i, \mathbf{g}+\mathbf{v}_i, \mathbf{p})$ for $i \in \{1,2\}$, and some vectors $\mathbf{u}_1, \mathbf{u}_2, \mathbf{v}_1, \mathbf{v}_2$ of integers. By only using commutativity of addition and subtraction, we obtain $t_1 \cdot t_2(\sigma) = (\boldsymbol{\kappa}+\mathbf{u}_1+\mathbf{u}_2, \mathbf{g}+\mathbf{v}_1+\mathbf{v}_2, \mathbf{p}) = (\boldsymbol{\kappa}+\mathbf{u}_2+\mathbf{u}_1, \mathbf{g}+\mathbf{v}_2+\mathbf{v}_1, \mathbf{p}) = t_2 \cdot t_1(\sigma)$.

(b) Let $\sigma_1 = t_1(\sigma)$, let $\sigma_2 = t_2(\sigma)$, and $\sigma_3 = t_1 \cdot t_2(\sigma)$. Then $\text{trace}_k(\mathsf{path}(\sigma, t_1 \cdot t_2)) = \lambda_k(\sigma)\lambda_k(\sigma_1)\lambda_k(\sigma_3)$, and $\text{trace}_k(\mathsf{path}(\sigma, t_2 \cdot t_1)) = \lambda_k(\sigma)\lambda_k(\sigma_2)\lambda_k(\sigma_3)$. We consider three cases: (i) $k \neq k_1$ and $k \neq k_2$, (ii) $k = k_1$, and (iii) $k = k_2$.

(i) In this case, by Lemma 6.17(c), we have $\lambda_k(\sigma) = \lambda_k(\sigma_1) = \lambda_k(\sigma_2) = \lambda_k(\sigma_3)$. Thus, both traces are $\lambda_k(\sigma)\lambda_k(\sigma)\lambda_k(\sigma)$, and they are clearly stutter equivalent.

(ii) Since $k = k_1 > k_2$, then again by Lemma 6.17(c) we have that $\lambda_k(\sigma_1) = \lambda_k(\sigma_3)$ and $\lambda_k(\sigma) = \lambda_k(\sigma_2)$. Thus, $\text{trace}_k(\mathsf{path}(\sigma, t_1 \cdot t_2)) = \lambda_k(\sigma)\lambda_k(\sigma_3)\lambda_k(\sigma_3)$, and $\text{trace}_k(\mathsf{path}(\sigma, t_2 \cdot t_1)) = \lambda_k(\sigma)\lambda_k(\sigma)\lambda_k(\sigma_3)$, and the traces are stutter equivalent.

(iii) The last case is analogous to the previous one. □

The following lemma tells us that adding or removing transitions of a round different from k results in a k-stutter equivalent path. It will be crucial only later, for the proof of Lemma 6.10.

Lemma 6.20. *Let σ be a configuration and let $t_1 = (r_1, k_1)$ and $t_2 = (r_2, k_2)$ be transitions such that $t_1 t_2$ is appllicable to σ. Then the following holds:*

(a) $\mathsf{path}(\sigma, t_1 t_2) \triangleq_k \mathsf{path}(\sigma, t_1)$, for every $k \neq k_2$, and

(b) $\mathsf{path}(\sigma, t_1 t_2) \triangleq_k \mathsf{path}(t_1(\sigma), t_2)$, for every $k \neq k_1$.

Proof. It follows directly from Lemma 6.17 (c). □

Proposition 6.5. *For every configuration σ and every finite schedule τ applicable to σ, there is a round-rigid schedule τ' such that the following holds:*

(a) Schedule τ' is applicable to σ.

(b) τ' and τ reach the same configuration when applied to σ, i.e., $\tau'(\sigma) = \tau(\sigma)$.

(c) For every $k \in \mathbb{N}_0$ we have $\mathsf{path}(\sigma, \tau) \triangleq_k \mathsf{path}(\sigma, \tau')$.

Proof. Since τ is finite, the claim (a) follows from Lemma 6.18, the second claim follows from Lemma 6.19(a), and the last one from Lemma 6.19(b). □

6.8.2 Detailed Proofs for Section 6.5.3

In order to prove Lemma 6.10, we introduce and prove a property of every $\mathsf{Sys}_\infty(\mathsf{TA})$.

Lemma 6.21. *Let $\mathsf{Sys}_\infty(\mathsf{TA})$ be deadlock-free, fix a $k \in \mathbb{N}_0$ and let σ be a configuration in $\mathsf{Sys}_\infty(\mathsf{TA})$ with a non-empty border location in round $k+1$, i.e., $\bigvee_{\ell \in \mathcal{B}} \sigma.\boldsymbol{\kappa}[\ell, k+1] \geq 1$. Then for every configuration σ' reachable from σ, there is a transition $t = (r, f, k_1)$ with $k_1 > k$ that is applicable to σ'.*

Proof. Let σ be a configuration with a non-empty border location in round $k+1$, and let σ' be a configuration reachable from σ. Assume by contradiction that there is no transition $t = (r, f, k_1)$ with $k_1 > k$ that is applicable to σ'. Recall that every location has a non-guarded outgoing rule. Thus, it must hold that for every location ℓ we have that $\sigma'.\boldsymbol{\kappa}[\ell, k_1] = 0$, for every $k_1 > k$. This is a contradiction with the assumption that σ' is reachable from σ and $\bigvee_{\ell \in \mathcal{B}} \sigma.\boldsymbol{\kappa}[\ell, k+1] \geq 1$. □

Lemma 6.10. *If $\mathsf{Sys}_\infty(\mathsf{TA})$ is deadlock-free, and if all fair executions of $\mathsf{Sys}^0(\mathsf{TA}^{rd})$ w.r.t. $\Sigma^0_{\mathcal{B}}$ terminate, then for every $k \in \mathbb{N}_0$ we have*

$$\mathsf{Sys}^k(\mathsf{TA}^{rd}) \triangleq_k \mathsf{Sys}_\infty(\mathsf{TA}),$$

i.e., the two systems are stutter equivalent w.r.t. AP_k.

Proof. We prove the statement by induction on $k \in \mathbb{N}_0$.

Base case. Let us first show that $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}}) \triangleq_0 \mathsf{Sys}_\infty(\mathsf{TA})$

($\Rightarrow$) Let $\pi = \mathsf{path}(\sigma, \tau)$ be a path in $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}})$. We need to find a path π' from $\mathsf{Sys}_\infty(\mathsf{TA})$, such that $\pi \triangleq_k \pi'$.

If $\tau = t_1 t_2 \ldots$, then every transition t_i either exists also in TA, or it is a self-loop at the copy of a border location. Using this, we construct a schedule $\tau' = t'_1 t'_2 \ldots$ in the following way.

For every $i \in \mathbb{N}$, if t_i exists in TA, then we define t'_i to be exactly t_i, and if t'_i is a self-loop at an $\ell' \in \mathcal{B}'$, then Lemma 6.21 gives us that there exists a transition $\tilde{t_i}$ from a round greater than 0 that is applicable to the current configuration, and we define $t'_i = \tilde{t_i}$. Thus, $\tau' = t'_1 t'_2 \ldots$ is obtained from τ by removing certain self-looping transitions and adding transitions of rounds greater than 0. By Lemma 6.20 we have $\mathsf{path}(\sigma, \tau') \triangleq_0 \mathsf{path}(\sigma, \tau)$.

Now we have that $\pi' = \mathsf{path}(\sigma, \tau') \triangleq_0 \mathsf{path}(\sigma, \tau) = \pi$.

($\Leftarrow$) Let now $\pi = \mathsf{path}(\sigma, \tau)$ be a path in $\mathsf{Sys}_\infty(\mathsf{TA})$. We construct a path $\pi' = \mathsf{path}(\sigma', \tau')$ from $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ such that $\pi \triangleq_k \pi'$. Since $I = I^0$, we define $\sigma' = \sigma$.

Let τ_0 be the projection of τ to round 0. There are two cases to consider. First, if τ and τ_0 are either both infinite or both finite schedules, then by Lemma 6.20 they yield stutter equivalent paths starting in σ. Observe that by Lemma 6.17 counters $\boldsymbol{\kappa}[\ell, 0]$ only change due to transitions for round 0, so that the applicability of τ_0 to σ follows from the applicability of τ.. Thus, in these cases we define τ' to be τ_0.

Second, we show the construction of τ' in the case when τ is an infinite schedule and τ_0 is finite. In this case we construct τ' as infinite extension of τ_0 as follows: Note that, since TA is deadlock-free, there must exist at least one location $\ell \in \mathcal{B}_1$ that is nonempty after executing τ_0 from σ, i.e., $\tau_0(\sigma).\boldsymbol{\kappa}[\ell, 1] \geq 1$. This must also be the case in $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}})$, with a difference that the nonempty location belongs to $\mathcal{B}'$, since $\mathcal{B}'$ plays the role of $\mathcal{B}_1$. If r is the self-looping rule at ℓ, then we obtain τ' by concatenating infinitely many transitions $(r, 1)$ to τ_0, i.e., $\tau' = \tau_0(r, 1)^\omega$. Transition $(r, 1)$ does not affect atomic propositions of round 0, and thus we have stutter equivalence by Lemma 6.20.

Induction step. Assume that $\mathsf{Sys}^i(\mathsf{TA}^{\mathrm{rd}}) \triangleq_i \mathsf{Sys}_\infty(\mathsf{TA})$ for every $0 \leq i < k$, and let us prove that the claim holds for k.

($\Rightarrow$) Let $\pi = \mathsf{path}(\sigma, \tau)$ be a path in $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$. We need to find a path π' from $\mathsf{Sys}_\infty(\mathsf{TA})$, such that $\pi \triangleq_k \pi'$.

First note that $\sigma \in I^k$. By definition of I^k, there exist a configuration $\sigma_0 \in I^0$ and schedules $\tau_1, \tau_2, \ldots, \tau_{k-1}$, such that every τ_i contains only transitions from round i, and $\tau_1\tau_2 \ldots \tau_{k-1}(\sigma_0) = \sigma$. Since no transition here is from round k, by Lemma 6.20 we have that $\mathsf{path}(\sigma_0, \tau_1\tau_2 \ldots \tau_{k-1}) \triangleq_k \mathsf{path}(\sigma, \varepsilon)$, where ε is the empty schedule. This path will be a prefix of π'.

If $\tau = t_1 t_2 \ldots$, then we use the same strategy as in the base case to define $\tau' = t_1' t_2' \ldots$ such that $\mathsf{path}(\sigma, \tau') \triangleq_k \mathsf{path}(\sigma, \tau)$.

Now we have that $\pi' = \mathsf{path}(\sigma_0, \tau_1\tau_2 \ldots \tau_{k-1}\tau') \triangleq_k \mathsf{path}(\sigma, \varepsilon\tau) = \pi$.

($\Leftarrow$) Let now $\pi = \mathsf{path}(\sigma, \tau)$ be a path in $\mathsf{Sys}_\infty(\mathsf{TA})$. We construct a path π' from $\mathsf{Sys}^k(\mathsf{TA}^{\mathrm{rd}})$ such that $\pi \triangleq_k \pi'$.

Since we assume that all fair executions of $\mathsf{Sys}^0(\mathsf{TA}^{\mathrm{rd}})$ terminate w.r.t. $\Sigma^0_{\mathcal{B}}$, then by Lemma 6.9 for every $0 \leq i < k$ the set I^i is well-defined and all fair executions of $\mathsf{Sys}^i(\mathsf{TA}^{\mathrm{rd}})$ terminate. By the induction hypothesis, we know that $\mathsf{Sys}^i(\mathsf{TA}^{\mathrm{rd}}) \triangleq_i \mathsf{Sys}_\infty(\mathsf{TA})$. Together, this gives us that all rounds i, with $0 \leq i < k$, terminate in $\mathsf{Sys}_\infty(\mathsf{TA})$. In other words, every execution of $\mathsf{Sys}_\infty(\mathsf{TA})$ has a finite prefix that contains all its transitions of rounds less than k.

Let τ_{pre} be such a prefix of $\tau = \tau_{\mathrm{pre}}\tau_{\mathrm{suf}}$. Because τ_{pre} is finite, we may invoke Proposition 6.5, from which follows that there exist schedules $\tau_0, \tau_1, \ldots, \tau_{k-1}, \tau_{\geq k}$ such that every τ_i, $0 \leq i < k$ contains only round i transitions, $\tau_{\geq k}$ contains transitions of rounds at least k, the schedule $\tau_0\tau_1 \ldots \tau_{k-1}\tau_{\geq k}$ is applicable to σ, leads to $\tau_{\mathrm{pre}}(\sigma)$ when applied to σ, and

$$\mathsf{path}(\sigma, \tau_0\tau_1 \ldots \tau_{k-1}\tau_{\geq k}\tau_{\mathrm{suf}}) \triangleq_k \mathsf{path}(\sigma, \tau_{\mathrm{pre}}\tau_{\mathrm{suf}}). \tag{6.17}$$

Since $\sigma \in I = I^0$, the existence of schedules $\tau_0, \tau_1, \ldots, \tau_{k-1}$ confirms that the configuration $\sigma' = \tau_0\tau_1 \ldots \tau_{k-1}(\sigma)$ is in I^k. Next we apply the strategy from the base case to construct τ'

from $\tau_{\geq k}\tau_{\text{suf}}$, by projecting it to round k, such that

$$\mathsf{path}(\sigma', \tau_{\geq k}\tau_{\text{suf}}) \triangleq_k \mathsf{path}(\sigma', \tau'). \tag{6.18}$$

By (6.17) and (6.18) we get $\pi' = \mathsf{path}(\sigma, \tau_0\tau_1 \dots \tau_{k-1}\tau') \triangleq_k \mathsf{path}(\sigma, \tau_{\text{pre}}\tau_{\text{suf}}) = \pi$. □

6.9 Discussion

In this chapter we lifted the threshold automata framework to multi-round randomized algorithms. We proved a reduction that allows to check $\mathsf{LTL}_{\text{-}\mathsf{X}}$ specifications over propositions for one round in a single-round automaton so that the verifications results transfer directly to the infinite counter system. We have shown, using round-based compositional reasoning, that it is sufficient to check specifications that span multiple rounds, e.g., agreement of consensus. We have applied a distinct reduction argument for almost sure termination under round-rigid adversaries.

We considered the algorithms that follow the ideas of Ben-Or [Ben83]. Interestingly, these algorithms were analyzed in [KNS01, KN02] where probabilistic reasoning was done using the probabilistic model checker PRISM [KNP11] for systems of 10-20 processes, while only safety was verified in the parameterized setting using Cadence SMV.

By experimental evaluation we showed that the verification conditions that came out of our reduction can be automatically verified for several challenging randomized consensus algorithms in the parameterized setting.

Our proof methodology for almost sure termination applies to round-rigid adversaries only. This restriction is crucial: transforming an adversary into a round-rigid one while preserving the probabilistic properties over the induced paths, comes up against the fact that, depending on the issue of a coin toss in some step at round k, different rules may be triggered later for processes in rounds less than k. As future work we shall prove that verifying almost-sure termination under round-rigid adversaries is sufficient to prove it for more general adversaries.

Regarding the structure of rounds, the authors of [NFM03] highlight problems on the notion of rounds in asynchronous distributed algorithms. The central problem is that the notion of a round provides some abstraction of time, which might not coincide with the notion of time that comes from the length of the prefix in asynchronous interleavings. In this chapter, for algorithms that can be represented as probabilistic threshold automata, we show that a reduction argument ensures that for interesting specifications we may focus on the rounds in reasoning about distributed algorithms in a sound way. We thus provide a precise relation between the asynchronous model and rounds asked for in [NFM03].

The idea of ordering rounds naturally can be seen as a way to impose a synchronous behavior on asynchronous systems. Due to the large number of possible interleavings in asynchronous systems, synchronous algorithm are simpler to specify and verify, and in general to reason about. Introducing *layers* in [MR02], defines submodels of asynchronous models that are very close to being synchronous, which is used for a model-independent analysis of consensus problem. Bringing the setting closer to synchronous was also used for automated verification of distributed algorithms in [KQH18, vGKB$^+$19].

Having rounds (or layers) that are *communication closed* is the key argument that allows us to simplify the problem of reasoning about all rounds, to the reasoning about a single representative round. Checking whether the existing rounds are communication closed, is a non-trivial problem addressed in [GS86, DDMW19]. A similar idea can be found in [BEJQ18], where the authors present an algorithm for deciding whether an algorithm has an equivalent k-*synchronous* computation, that is, if it can be decomposed to the sequence of rounds that contain at most k send events and k corresponding receive events.

CHAPTER 7

Conclusions

In this thesis we explored the applicability of reduction techniques for computer-aided verification of fault-tolerant distributed algorithms in the parameterized setting. While the distributed algorithms literature and verification methods predominantly focus on safety properties, it is a folklore knowledge that only the interplay between safety and liveness makes distributed algorithms meaningful. For instance, the algorithm that does nothing is safe. Our verification technique is designed to support both safety and liveness properties, expressed in linear temporal logic that only uses temporal operators **F** and **G**. This fragment of linear temporal logic, denoted by $\mathsf{ELTL}_{\mathsf{FT}}$, is carefully crafted: (i) it is expressive enough to contain the specifications of all our benchmarks, and (ii) the parameterized model checking problem is still decidable for this logic.

We restrict the class of distributed algorithms under analysis. We focus on asynchronous algorithms where processes do not have IDs, communicate by broadcasting messages, and compare the number of received messages against thresholds. Examples of such algorithms are reliable broadcast, condition-based consensus, non-blocking atomic commitment, etc. By exploiting reduction, we have developed a technique for parameterized model checking of such algorithms, supported by the fully automated tool Byzantine Model Checker (ByMC) [KW18].

Thresholds in distributed algorithms are a necessary ingredient for achieving fault-tolerance. Waiting for an acknowledgment from, e.g., $t + 1$ messages, where t is an upper bound on the number of faulty processes, ensures that a process does not make progress before it receives a message from at least one correct process. The semantics of more involved thresholds, like $(n + 3t)/2$, is not easy to understand from the arithmetic expression. That is one of the reasons why the design of threshold-based distributed algorithms is an immensely difficult task. While the literature shows hand-written proofs of algorithms, which confirm that the given thresholds are the suitable ones, the discussion on how to come up with such thresholds is usually omitted. Our technique for automatic synthesis of parameterized thresholds addresses this issue. One only needs to come up with (i) specifications, which are well-known from distributed algorithms literature, and (ii) with a "skeleton" of an algorithm, that is, an algorithm with wholes instead of thresholds. Our technique synthesizes thresholds such that, when inserted to the skeleton, they yield the algorithm that is correct for all legal values of parameters. Hence, our method systematically derives thresholds from specification. This can be seen as a more structured approach of designing FTDAs.

All the techniques presented in this thesis extend the basic idea of reduction. This is a method for reasoning about dependency of concurrently executed events in a path. Our idea is to reorder independent transitions in such a way that the transitions of the same nature are grouped together. Next we replace this sequence of similar transitions by a single accelerated transition. This decreases the length of paths, and therefore it allows us to analyze only "short" paths in a system. This seemingly simple form of reduction proved to be the key for cutting down the complexity of parameterized model checking of threshold-based distributed algorithms. We have also seen in this thesis that for the extension of the setting, e.g., coin tosses and unboundedly many rounds, the extension of this reduction for multiple rounds made the first automated verification of randomized distributed algorithms.

This thesis demonstrates that reduction is a powerful and a promising method for the parameterized analysis, whose extensions can lead to the further development of parameterized verification and synthesis of fault-tolerant distributed algorithms. We have set the stable background for such an analysis.

7.1 Technical Contributions

The techniques presented in this thesis allow us to solve challenges C1–C4 from Figure 1.2 in Section 1.2.

Challenge C1 Chapter 3 deals with verification of reachability properties of threshold-based FTDAs. As a starting point for this work we use the work [KVW14]. We keep the same modeling, namely threshold automata, and reuse the idea of accelerated transitions, where multiple processes are allowed to perform the same step in one global transition. This reduces the size of executions. Nonetheless, the reduction technique PARA^2 presented here, demonstrates a vast improvement in efficiency and reliability of checking reachability properties. Our contribution can be summarized as follows:

- *SMT-based bounded model checking.* As we use counter systems, and keep track of values of counters, that are natural numbers, transitions are described by incrementing and decrementing counters. This allows us to encode transitions in linear integer arithmetic. Thus, we can use SMT solvers to efficiently analyze executions.

- *Execution schemas.* Instead of checking all executions of bounded length, we introduce representatives, so-called schemas, and prove that checking only them is a necessary and sufficient condition for investigating correctness of an FTDA.

Challenge C2 Building on the results of Chapter 3, in Chapter 4 we raise this idea to the next level so we can check both safety and liveness of FTDAs. As our contributions, we single out the following:

- *Lasso-shaped executions.* As in the classic result by Vardi and Wolper [VW86], we observe that it is sufficient to search for counterexamples that have the form of a lasso, i.e., after a finite prefix an infinite loop is entered. Based on this, we analyze specifications automatically, in order to enumerate possible shapes of lassos depending on temporal operators F and G and evaluations of threshold guards.
- *Property specific parameterized path reduction.* We automatically do offline partial order reduction using the algorithm's description. To this end, we introduce a more refined mover analysis for threshold guards and temporal properties. We extend the PARA^2 method for reachability, so that we maintain invariants, which allows us to go beyond reachability and verify specifications with the temporal operators F and G.
- *A short counterexample property.* By combining acceleration [KVW17] with the previous two points, we obtain a short counterexample property, that is, that infinite executions (which may potentially be counterexamples) have "equivalent" representatives of bounded length. The bound depends on the process code and is independent of the parameters. The equivalence is understood in terms of temporal logic specifications that are satisfied by the original executions and the representatives, respectively. We show that the length of the representatives increases only by a constant factor, compared to reachability checking from Chapter 3. This implies a so-called completeness threshold [KOS$^+$11] for threshold-based algorithms and our fragment of LTL.
- *Complete bounded model checking.* Consequently, we only have to check a reasonable number of SMT queries that encode parameterized and bounded-length representatives of executions. We show that if the parameterized system violates a temporal property, then SMT reports a counterexample for one of the queries. We prove that otherwise the specification holds for all system sizes.
- *Efficient model checking of safety and liveness of FTDAs.* Our theoretical results and our implementation push the boundary of liveness verification for fault-tolerant distributed algorithms. While prior results [JKS$^+$13a] scale just to two out of ten benchmarks from [KVW15], we verified safety and liveness of all ten. These benchmarks originate from distributed algorithms [CT96, ST87b, BT85, MMPR03, Ray97, Gue02, DS06, BGMR01, SvR08] that constitute the core of important services such as replicated state machines.

Based upon this we develop two new ideas: (i) synthesis of parameterized threshold guards, and (ii) verification of randomized distributed algorithms.

Challenge C3 Knowing how to check whether a given FTDA satisfies a temporal property, allows us to address synthesis of FTDAs. Chapter 5 considers automated synthesis of parameterized threshold guards in FTDAs for a given sketch FTDA and a specification, that yields an FTDA that is correct by construction, for any number of processes and any number of faults. Our main contributions are the following:

- *ByMC and the CEGIS loop.* Counterexample-guided inductive synthesis [ABJ$^+$13] is a technique that requires two counterparts: (i) a generator of candidate solutions, and (ii) a verification oracle that checks if candidates are correct. In our case we use our technique from Chapter 4 as the verification oracle, implemented in the ByMC tool.
- *Learning from counterexamples.* For sophisticated benchmarks that demand computationally expensive verification procedure, it is important to have as few as possible verification calls. For that we make the generator learn from counterexamples. Existence of execution schemas comes to play here, as detecting a counterexample gives us a concrete execution and its schema. Based on the SMT schema analysis, one easily obtains all candidate thresholds that make this schema violate specifications. This drastically reduces the search space for the coefficients of threshold guards.
- *Loop termination using sane guards.* Still, reduced search space does not mean that the search space is finite, and therefore, we have no guarantees that our CEGIS loop terminates. In order to assure termination, we restrict the search to so-called sane guards, that are, as the name suggests, those that are semantically plausible. In other words, we do not allow negative number of sent messages, as well as more messages than processes if each process is able to send up to one message. We prove that the search space for sane guards is bounded.
- *Synthesizing parameterized FTDAs.* This loop allows us to synthesize threshold guards that together with a given sketch form an FTDA that is correct for any system size. In our experiments we have synthesized algorithms from [ST87b, WS07, SvR08] for different fault models, and also we have shown that theoretical resilience conditions are tight. That is, there is no solution if we have more faults than theoretically predicted. Moreover, we have synthesized new versions of these algorithms that satisfy slightly different specifications.

Challenge C4 Chapter 6 presents the first successful automated technique for parameterized verification of randomized FTDAs. We extend threshold automata to round-based algorithms with coin toss transitions. For the new framework we achieve the following:

1. *Round-based reduction.* For safety verification we introduce a method for compositional round-based reasoning. This allows us to invoke a reduction similar to the one in [EF82], that extends the PARA^2 technique for safety and liveness. We highlight necessary fairness conditions on individual rounds. This provides us with specifications to be checked on a one-round automaton.

2. *Almost-sure termination under round-rigid adversaries.* For probabilistic liveness verification, we explain how to reduce to proving termination with positive probability within a fixed number of rounds. To do so, we justify the restriction to round-rigid adversaries, that is, adversaries that respect the order of rounds. In contrast to existing work that proves almost-sure termination for fixed number of participants, these are the first parameterized model checking results for probabilistic properties.

3. *Verification of randomized FTDAs.* We checked the specifications that emerge from the previous two points and thus verified challenging benchmarks in the parameterized setting. We verify Ben-Or's [Ben83] and Bracha's [Bra87] classic consensus algorithms, and the more recent algorithms 2-set agreement [MMR18], and RS-Bosco [SvR08].

7.2 Future Work

Parameterized verification and synthesis of fault-tolerant distributed algorithms are immensely difficult tasks. That there is a trade-off between degree of automation and generality, confirms the fact that there is no fully automated technique for verification nor synthesis for comprehensive classes of distributed algorithms. Moreover, we have seen that these tasks are often undecidable, especially in the parameterized case [AK86, BJK$^+$15, JB14]. In our work we give priority to automated verification, at the expense of the class of the algorithms we are able to address.

Parameterized verification of FTDAs Currently, we focus on algorithms with the following characteristics:

- the timing model is asynchronous,
- processes communicate by message passing,
- all processes are identical, i.e., we have symmetric systems,
- process only broadcast messages,
- transitions are guarded by thresholds.

Any extension of the class of fault-tolerant distributed algorithms with these properties would be a significant advance in the field. Similar to our approach in Chapter 6, where we introduced coin tosses to the threshold automata framework and almost-sure termination, we believe that tackling one of these restrictions at a time will lead to more general extensions of our technique. To this end, dealing with threshold guards is crucial, and in this thesis we provide a solid base.

Synthesis of parameterized FTDAs Our synthesis approach from Chapter 5 is a first step towards full synthesis of parameterized threshold-based FTDAs from specifications, and therefore there is much room for improvement. Considering the efficiency of the method, we conjecture that thorough verification in every iteration of the CEGIS loop is not necessary. Verification in our case means occasionally checking even hundreds or thousand of schemas. We believe that there are only a few schemas that dramatically reduce the search space, and therefore detecting and checking only them in the first place, would significantly improve efficiency.

To ensure termination of the synthesis loop, we restrict the search space, and thus the class of algorithms for which the impossibility result formally applies. First, while we restrict the search to sane guards, the same synthesis loop can also be used to synthesize other guards. However, in order to ensure termination, a suitable characterization of sought-after guards should be provided by the user. Second, for reliable broadcast we consider only threshold guards with integer coefficients that can express thresholds like $n-t$ or $2t+1$. For BOSCO, we only allow division by 2, and can express thresholds like $\frac{n}{2}$ or $\frac{n-t}{2}$. While from a theoretical viewpoint these restrictions limit the scope of our results, we are not aware of a distributed algorithm where processes wait for messages from, say, $\frac{n}{7}$ or $\frac{n}{1000}$ processes. To strengthen our completeness claim, we would need to formally explain why only small denominators are used in fault-tolerant distributed algorithms. We conjecture that for every FTDA that uses a rational with a large denominator, there is an equivalent FTDA that uses a small denominator. Our current technique does not allow us to compare executions of different algorithms.

Parameterized verification of randomized FTDAs Similarly, our technique from Chapter 6 is our starting point in the parameterized verification of randomized FTDAs. The technique fully verifies non-probabilistic properties, by reducing them to one-round properties. This reduction requires manual efforts of the user. It is an open question if the automated translation of multi-round properties to one-round properties is even possible.

Furthermore, we are only able to verify probabilistic properties with probability 1, and under round-rigid adversaries. Our approach is based on a path reduction that requires swapping adjacent transitions in order to obtain an execution that satisfies the same properties, and in which rounds are ordered naturally. Unfortunately, for reasoning about probabilistic termination, we need to reason about computational trees instead of paths. Thus, our PARA^2 techniques currently cannot deal with arbitrary adversaries. We conjecture that for an arbitrary adversary, there exists a round-rigid adversary that preserves probabilistic properties. This would mean that our method is complete.

Bibliography

[ABEL12] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. Detecting fair non-termination in multithreaded programs. In *CAV*, pages 210–226, 2012.

[ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In *CAV*, LNCS, pages 305–318, 1998.

[ABJ$^+$13] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8, 2013.

[AGP16] Francesco Alberti, Silvio Ghilardi, and Elena Pagani. Counting constraints in flat array fragments. In *IJCAR*, volume 9706 of *LNCS*, pages 65–81, 2016.

[AHH13] Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. All for the price of few. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 476–495. Springer, 2013.

[AJKR14] Benjamin Aminof, Swen Jacobs, Ayrat Khalimov, and Sasha Rubin. Parameterized model checking of token-passing systems. In *VMCAI*, volume 8318 of *Lecture Notes in Computer Science*, pages 262–281. Springer, 2014.

[AK86] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 15:307–309, 1986.

[AKR$^+$18] Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. Parameterized model checking of rendezvous systems. *Distributed Computing*, 31(3):187–222, 2018.

[ARS$^+$18] Benjamin Aminof, Sasha Rubin, Ilina Stoilkovska, Josef Widder, and Florian Zuleger. Parameterized model checking of synchronous distributed algorithms by abstraction. In *VMCAI*, volume 10747 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2018.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[AT12] Marcos Aguilera and Sam Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, pages 1–11, 2012. online first.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing.* Wiley, 2nd edition, 2004.

[BAS02] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[BBJ$^+$16a] Benjamin Bisping, Paul-David Brodmann, Tim Jungnickel, Christina Rickmann, Henning Seidler, Anke Stüber, Arno Wilhelm-Weidner, Kirstin Peters, and Uwe Nestmann. A constructive proof for FLP. *Archive of Formal Proofs*, 2016.

[BBJ16b] Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and Byzantine-resilient distributed systems. In *CAV*, volume 9779 of *LNCS*, pages 157–176, 2016.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207, 1999.

[BCD$^+$11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[BCG89] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Inf. Comput.*, 81:13–31, 1989.

[BCM$^+$90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *LICS*, pages 428–439. IEEE Computer Society, 1990.

[BDMS13] Martin Biely, Pamela Delgado, Zarko Milosevic, and André Schiper. Distal: a framework for implementing fault-tolerant distributed algorithms. In *DSN*, pages 1–8, 2013.

[BEJQ18] Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. On the completeness of verifying message passing programs under bounded asynchrony. In *CAV (2)*, volume 10982 of *Lecture Notes in Computer Science*, pages 372–391. Springer, 2018.

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC*, pages 27–30, 1983.

[BFLP08] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. Fast: acceleration from theory to practice. *STTT*, 10(5):401–424, 2008.

[BFLS05] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Philippe Schnoebelen. Flat acceleration in symbolic model checking. In *ATVA*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005.

[BGMR01] Francisco Vilar Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *PaCT*, volume 2127 of *LNCS*, pages 42–50, 2001.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract regular model checking. In *CAV*, LNCS, pages 372–386, 2004.

[Bie13] Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. *Proceedings of SAT Competition 2013; Solver and*, page 51, 2013.

[BJK$^+$15] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

[BKLW18] Nathalie Bertrand, Igor Konnov, Marijana Lazić, and Josef Widder. Verification of Randomized Distributed Algorithms under Round-Rigid Adversaries. *HAL*, hal-01925533, Nov 2018.

[BKSS11] Péter Bokor, Johannes Kinder, Marco Serafini, and Neeraj Suri. Efficient model checking of fault-tolerant distributed protocols. In *DSN*, pages 73–84, 2011.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[BM83] Robert S Boyer and J Strother Moore. Proof-checking, theorem proving, and program verification. Technical report, TEXAS UNIV AT AUSTIN INST FOR COMPUTING SCIENCE AND COMPUTER APPLICATIONS, 1983.

[BMWK09] Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, volume 5643 of *LNCS*, pages 64–78, 2009.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[Bra12] Aaron R. Bradley. IC3 and beyond: Incremental, inductive verification. In *CAV*, volume 7358 of *Lecture Notes in Computer Science*, page 4. Springer, 2012.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[Buc16] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of Blockchains. Master's thesis, University of Guelph, 2016.

[CBM09] Bernadette Charron-Bost and Stephan Merz. Formal verification of a consensus algorithm in the heard-of model. *IJSI*, 3(2–3):273–303, 2009.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[CCD$^+$14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *CAV*, volume 8559 of *LNCS*, pages 334–342, 2014.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, volume 6173 of *LNCS*, pages 142–148, 2010.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71, 1981.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR*, pages 395–407, 1995.

[CGJ97] Edmund M. Clarke, Orna Grumberg, and Somesh Jha. Verifying parameterized networks. *ACM Trans. Program. Lang. Syst.*, 19(5):726–750, 1997.

[CGJ$^+$03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, 1994.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[Chu63] Alonzo Church. Logic, arithmetic and automata. In *Proc. 1962 Intl. Congr. Math.*, pages 23–25, 1963.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In *VMCAI*, volume 2937 of *LNCS*, pages 85–96, 2004.

[CL98] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *CONCUR*, volume 1466 of *LNCS*, pages 317–331, 1998.

[CS09] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[CTTV04] Edmund Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. Verification by network decomposition. In *CONCUR 2004*, volume 3170, pages 276–291, 2004.

[CTV08] Edmund Clarke, Murali Talupur, and Helmut Veith. Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In *TACAS'08/ETAPS'08*, pages 33–47. Springer, 2008.

[CW95] E. Rodney Canfield and S. Gill Williamson. A loop-free algorithm for generating the linear extensions of a poset. *Order*, 12(1):57–75, 1995.

[DDMW19] Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. working paper or preprint, Jan 2019.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34:77–97, 1987.

[DEGM15] Antoine Durand-Gasselin, Javier Esparza, Pierre Ganty, and Rupak Majumdar. Model checking parameterized asynchronous shared-memory systems. In *CAV (Part I)*, pages 67–84, 2015.

[DF09] Rayna Dimitrova and Bernd Finkbeiner. Synthesis of fault-tolerant distributed systems. In *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2009.

[DHJ$^+$16] Danny Dolev, Keijo Heljanko, Matti Järvisalo, Janne H. Korhonen, Christoph Lenzen, Joel Rybicki, Jukka Suomela, and Siert Wieringa. Synchronous counting and computational algorithm design. *J. Comput. Syst. Sci.*, 82(2):310–332, 2016.

[DHV$^+$14] Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, volume 8318 of *LNCS*, pages 161–181, 2014.

[DHZ16] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415, 2016.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[DLW18] Cezara Drăgoi, Marijana Lazić, and Josef Widder. Communication-closed layers as paradigm for distributed systems: A manifesto. In *Sinteza*, pages 131–138, 2018.

[Doe77] Thomas W. Doeppner. Parallel program correctness through refinement. In *POPL*, pages 155–169, 1977.

[DS06] Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *DSN*, pages 137–146, 2006.

[EF82] Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.

[EFM99] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999.

[EGM16] Javier Esparza, Pierre Ganty, and Rupak Majumdar. Parameterized verification of asynchronous shared-memory systems. *J. ACM*, 63(1):10:1–10:48, 2016.

[EK03] E. Allen Emerson and Vineet Kahlon. Model checking guarded protocols. In *LICS*, pages 361–370. IEEE, 2003.

[EN95] E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *POPL*, pages 85–94, 1995.

[EN03] E. Allen Emerson and Kedar S. Namjoshi. On reasoning about rings. *Int. J. Found. Comput. Sci.*, 14(4):527–550, 2003.

[EQT09] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.

[Esp14] Javier Esparza. Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In *STACS*, volume 25 of *LIPIcs*, pages 1–10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[EVW02] Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. First-order logic with two variables and unary temporal logic. *Inf. Comput.*, 179(2):279–295, 2002.

[Ext15] Andy Extance. The future of cryptocurrencies: Bitcoin and beyond. *Nature*, 526, 2015. `http://dx.doi.org/10.1038/526021a`.

[FB15] Fathiyeh Faghih and Borzoo Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *TAAS*, 10(3):21:1–21:26, 2015.

[FBTK16] Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, and Sandeep S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. In *FORTE*, volume 9688 of *LNCS*, pages 124–141, 2016.

[FFQ05] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. *IEEE Trans. Softw. Eng.*, 31(4):275–291, 2005.

[FKL08] Dana Fisman, Orna Kupferman, and Yoad Lustig. On verifying fault tolerance of distributed protocols. In *TACAS*, volume 4963 of *LNCS*, pages 315–331. Springer, 2008.

[FKP13] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *POPL*, pages 129–142, 2013.

[FKP15] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proof spaces for unbounded parallelism. In *POPL*, pages 407–420, 2015.

[FKP16] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proving liveness of parameterized programs. In *LICS*, pages 185–196, 2016.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FPPZ06] Yi Fang, Nir Piterman, Amir Pnueli, and Lenore D. Zuck. Liveness with invisible ranking. *STTT*, 8(3):261–279, 2006.

[FS13] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.

[GKS$^+$14] Annu Gmeiner, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In *Formal Methods for Executable Software Models*, LNCS, pages 122–171. Springer, 2014.

[Gme15] Annu Gmeiner. *Parameterized model checking of fault-tolerant distributed algorithms*. PhD thesis, TU Wien, 2015.

[God90] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, volume 531 of *LNCS*, pages 176–185, 1990.

[GOS16] GOS. Distributed ledger technology: beyond block chain. A report by the UK Government Chief Scientific Adviser. GS/16/1, 2016. `https://www.gov.uk/government/publications/distributed-ledger-technology-blackett-review`.

[GS86] Rob Gerth and Liuba Shrira. On proving communication closedness of distributed layers. In *FSTTCS*, pages 330–343, 1986.

[GS92] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *CAV*, volume 1254 of *LNCS*, pages 72–83, 1997.

[GT14] Adrià Gascón and Ashish Tiwari. A synthesized algorithm for interactive consistency. In *NFM*, volume 8430 of *LNCS*, pages 270–284. Springer, 2014.

[Gue02] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[GV08] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.

[HHK$^+$15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *SOSP*, pages 1–17, 2015.

[HHK$^+$17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, 2017.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[JB14] Swen Jacobs and Roderick Bloem. Parameterized synthesis. *LMCS*, 10(1:12), 2014.

[JKS$^+$13a] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD*, pages 201–209, 2013.

[JKS$^+$13b] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Towards modeling and model checking fault-tolerant distributed algorithms. In *SPIN*, volume 7976 of *LNCS*, pages 209–226, 2013.

[KAB$^+$07] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: language support for building distributed systems. In *ACM SIGPLAN PLDI*, pages 179–188, 2007.

[KE17] Alex Klinkhamer and Ali Ebnenasir. Synthesizing parameterized self-stabilizing rings with constant-space processes. In *FSEN*, volume 10522 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2017.

[KKW10] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, volume 6174 of *LNCS*, pages 654–659. Springer, 2010.

[KKW18] Jure Kukovec, Igor Konnov, and Josef Widder. Reachability in parameterized systems: All flavors of threshold automata. In *CONCUR*, volume 118 of *LIPIcs*, pages 19:1–19:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[KLVW17a] Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. Para^2: Parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. *Formal Methods in System Design*, 51(2):270–307, 2017.

[KLVW17b] Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, pages 719–734, 2017.

[KM95] Robert P. Kurshan and Kenneth L. McMillan. A structural induction theorem for processes. *Inf. Comput.*, 117(1):1–11, 1995.

[KN02] Marta Z. Kwiatkowska and Gethin Norman. Verifying randomized byzantine agreement. In *FORTE*, pages 194–209, 2002.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011.

[KNS01] Marta Z. Kwiatkowska, Gethin Norman, and Roberto Segala. Automated verification of a randomized distributed consensus protocol using cadence SMV and PRISM. In *CAV*, pages 194–206, 2001.

[KOS$^+$11] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. Linear completeness thresholds for bounded model checking. In *CAV*, volume 6806 of *LNCS*, pages 557–572, 2011.

[KP00] Yonit Kesten and Amir Pnueli. Control and data abstraction: the cornerstones of practical formal verification. *STTT*, 2:328–342, 2000.

[KQH18] Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Synchronizing the asynchronous. In *CONCUR*, pages 21:1–21:17, 2018.

[KS03] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *VMCAI*, volume 2575 of *LNCS*, pages 298–309, 2003.

[KVW14] Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In *CONCUR*, volume 8704 of *LNCS*, pages 125–140, 2014.

[KVW15] Igor Konnov, Helmut Veith, and Josef Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV (Part I)*, volume 9206 of *LNCS*, pages 85–102, 2015.

[KVW16] Igor Konnov, Helmut Veith, and Josef Widder. What you always wanted to know about model checking of fault-tolerant distributed algorithms. In *PSI 2015, Revised Selected Papers*, volume 9609 of *LNCS*, pages 6–21. Springer, 2016.

[KVW17] Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 252:95–109, 2017.

[KW18] Igor Konnov and Josef Widder. Bymc: Byzantine model checker. In *ISoLA (3)*, volume 11246 of *LNCS*, pages 327–342. Springer, 2018.

[KZ10] Igor V. Konnov and Vladimir A. Zakharov. An invariant-based approach to the verification of asynchronous parameterized networks. *J. Symb. Comput.*, 45(11):1144–1162, 2010.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam02] Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers.* Addison-Wesley, 2002.

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370, 2016.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LKWB17] Marijana Lazić, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *OPODIS*, volume 95 of *LIPIcs*, pages 32:1–32:20, 2017.

[LR81] Daniel J. Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *POPL*, pages 133–138, 1981.

[LR93] P. Lincoln and J. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *FTCS*, pages 402–411, 1993.

[LS89] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report 44, SRC, 1989.

[LS04] Jérôme Leroux and Grégoire Sutre. On flatness for 2-dimensional vector addition systems with states. In *CONCUR 2004-Concurrency Theory*, pages 402–416. Springer, 2004.

[LS05] Jérôme Leroux and Grégoire Sutre. Flat counter automata almost everywhere! In *ATVA*, volume 3707 of *LNCS*, pages 489–503, 2005.

[Lub84] Boris D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. I. *Acta Informatica*, 21(2):125–169, 1984.

[Lyn96] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

[MB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 1579 of *LNCS*, pages 337–340. 2008.

[MFJB18] Nahal Mirzaie, Fathiyeh Faghih, Swen Jacobs, and Borzoo Bonakdarpour. Parameterized synthesis of self-stabilizing protocols in symmetric rings. In *OPODIS*, volume 125 of *LIPIcs*, pages 29:1–29:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[MMPR03] Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

[MMR18] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Randomized k-set agreement in crash-prone and byzantine asynchronous systems. *Theor. Comput. Sci.*, 709:80–97, 2018.

[MP18] Kenneth L. McMillan and Oded Padon. Deductive verification in decidable fragments with ivy. In *SAS*, volume 11002 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2018.

[MPST14] Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In *SSS*, volume 8756 of *LNCS*, pages 237–251, 2014.

[MR02] Yoram Moses and Sergio Rajsbaum. A layered analysis of consensus. *SIAM J. Comput.*, 31(4):989–1021, 2002.

[MSB17] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *CAV*, pages 217–237, 2017.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `https://bitcoin.org/bitcoin.pdf`.

[Net10] Netflix. 5 lessons we have learned using AWS. 2010. retrieved on Nov. 7, 2016. `http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html`.

[NFM03] Uwe Nestmann, Rachele Fuzzati, and Massimo Merro. Modeling consensus in a process calculus. In *CONCUR*, pages 393–407, 2003.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–320, 2014.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *CAV*, volume 697 of *LNCS*, pages 409–423, 1993.

[PHL$^+$18] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *PACMPL*, 2(POPL):26:1–26:33, 2018.

[PHM$^+$18] Oded Padon, Jochen Hoenicke, Kenneth L. McMillan, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Temporal prophecy for proving temporal properties of infinite-state systems. In *FMCAD*, pages 1–11. IEEE, 2018.

[PMP$^+$16] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630, 2016.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190. ACM Press, 1989.

[PS00] Amir Pnueli and Elad Shahar. Liveness and acceleration in parameterized verification. In *CAV*, LNCS, pages 328–343, 2000.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[PTP$^+$16] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *DSN*, pages 156–167, 2016.

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with $(0,1,\infty)$- counter abstraction. In *CAV*, volume 2404 of *LNCS*, pages 93–111. 2002.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[Ray97] Michel Raynal. A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In *HASE*, pages 209–214, 1997.

[RGBC15] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. *ECEASST*, 72, 2015.

[SB06] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electronic Notes in Theoretical Computer Science*, 149(1):79–96, 2006.

[SG89] Ze'ev Shtadler and Orna Grumberg. Network grammars, communication behaviors and automatic verification. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1989.

[SKWZ18] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. *HAL*, hal-01925653, 2018.

[ST87a] K. T. Srikanth and Sam Toueg. Optimal clock synchronization. *J. ACM*, 34(3):626–645, 1987.

[ST87b] T.K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comp.*, 2:80–94, 1987.

[Suz88] Ichiro Suzuki. Proving properties of a ring of finite-state machines. *Inf. Process. Lett.*, 28(4):213–214, 1988.

[SvR08] Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *DISC*, volume 5218 of *LNCS*, pages 438–450, 2008.

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *PACMPL*, 2(POPL):28:1–28:30, 2018.

[Tea08] The Amazon S3 Team. Amazon s3 availability event: July 20, 2008. *Amazon Web Services: Service health dashboard*, 2008. `http://status.aws.amazon.com/s3-20080720.html`.

[TLA] TLA+ toolbox. `http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html`.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *LNCS*, pages 491–515. Springer, 1991.

[vGBR16] Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*, pages 599–613, 2016.

[vGKB$^+$19] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *PACMPL*, 3(POPL):59:1–59:30, 2019.

[VW86] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 322–331, 1986.

[WL89] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 1989.

[WS07] Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Dist. Comp.*, 20(2):115–140, 2007.

[WWP$^+$15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

Marijana Lazić

Curriculum Vitae

+43 677 61357236
+381 64 2648549
lazic@forsyte.tuwien.ac.at
forsyte.at/lazic

Personal Information

Born	May 6, 1988 in Šabac, Serbia
Languages	Serbian (native), English (fluent), German and Spanish (basic knowledge)
Address	Technische Universität Wien, Institut für Logic and Computation 192/4, Favoritenstraße 9-11, 1040, Vienna, Austria
Web	`http://forsyte.at/~lazic/`

Education

2015-Present	**Ph.D. studies in Computer Science**, *Technische Universität Wien, Austria*.
Supervisors	Privatdoz. Dr. Josef Widder and Dr. Igor Konnov (Initially my supervisor was Prof. Dr. Helmut Veith✝)
2011–2012	**Master studies in Mathematics**, *University of Novi Sad, Serbia*.
Prof. title	Master Professor in Mathematics
GPA	9.61/10 (Excellent)
Thesis	Power structures and Multialgebras
Supervisor	Prof. Dr. Rozalia Madarasz Szilagyi
2007–2011	**Bachelor studies in Mathematics**, *University of Novi Sad, Serbia*.
Prof. title	Bachelor with Honors in Mathematics Teaching
GPA	9.47/10 (Exceptionally good)

Publications

2018 Cezara Dragői, Marijana Lazić, and Josef Widder. Communication-closed layers as paradigm for distributed systems: A manifesto. In *Sinteza*, pages 131–138, 2018.

2018 Nathalie Bertrand, Igor Konnov, Marijana Lazić, and Josef Widder. Verification of Randomized Distributed Algorithms under Round-Rigid Adversaries. *HAL*, hal-01925533, Nov 2018.

2017 Marijana Lazić, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *OPODIS*, volume 95 of *LIPIcs*, pages 32:1–32:20, 2017.

2017 Igor V. Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 719–734, 2017. A pre-print including the proofs is available at `http://arxiv.org/abs/1608.05327`.

2017 Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. Para^2: Parameterized Path Reduction, Acceleration, and SMT for Reachability in Threshold-Guarded Distributed Algorithms. *Formal Methods in System Design*, 2017.

2016 Erhard Aichinger, Marijana Lazić, and Nebojša Mudrinski. Finite generation of congruence preserving functions. *Monatshefte für Mathematik*, (181):35–62, 2016.

Invited Talks

November 2018 **Parameterized Verification of Randomized Distributed Algorithms under Round-Rigid Adversaries**, TU Munich, Germany.

May 2018 **Synthesizing thresholds for fault-tolerant distributed algorithms**, *Formal Methods and Fault-Tolerant Distributed Computing: Forging an Alliance*, Dagstuhl seminar, Germany.

December 2017 **Parameterized Verification and Synthesis of Distributed Algorithms**, Tel Aviv University, Tel Aviv, Israel.

February 2017 **A Short Counterexample Property for Safety and Liveness Verification of Fault-tolerant Distributed Algorithms**, University of Novi Sad (Faculty of Sciences and Faculty of Technical Sciences), Novi Sad, Serbia.

July 2016 **Parameterized Verification of Liveness of Distributed Algorithms**, Yale University, New Haven, CT, USA.

July 2016 **Parameterized Verification of Liveness of Distributed Algorithms**, Amazon, New York City, NY, USA.

Other Talks

April 2018 **Synthesis of Distributed Algorithms with Parameterized Threshold Guards**, University of Novi Sad, Serbia.

December 2017 **Synthesis of Distributed Algorithms with Parameterized Threshold Guards**, The 2nd Winter School in Engineering and Computer Science on Formal Verification (student spotlight), Jerusalem, Israel.

January 2017 **A Short Counterexample Property for Safety and Liveness Verification of Fault-tolerant Distributed Algorithms**, joint work with Igor Konnov, Helmut Veith, and Josef Widder. The 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017), Paris, France.

May 2016 **Completeness of Bounded Model Checking of Threshold-Based Distributed Algorithms: Safety and Liveness**, joint work with Igor Konnov, Helmut Veith, and Josef Widder. The Third Workshop on Formal Reasoning in Distributed Algorithms (FRIDA 2016), Marrakech, Morocco.

February 2015 **On terms in 3-supernilpotent groups with an additional unary operation**, joint work with Erhard Aichinger, and Nebojša Mudrinski. The 89th Workshop on General Algebra (89. Arbeitstagung Allgemeine Algebra, AAA89), Dresden, Germany.

February 2014 **On a class of finite modular congruence lattices**, joint work with Erhard Aichinger, and Nebojša Mudrinski. The 87th Workshop on General Algebra (87. Arbeitstagung Allgemeine Algebra, AAA87), Linz, Austria.

Research Visits

Sep 2018 **Sergio Rajsbaum**, *Universidad Nacional Autónoma de México*, Mexico City.

Jun-Jul 2017
Nov-Dec 2017 **Yoram Moses**, *Technion, Israel Institute of Technology*, Haifa, Israel.

Oct-Dec 2016 **Prakash Panangaden and Vijay D'Silva**, *Simons Institute for the Theory of Computing, University of California, Berkeley*.

Jun-Aug 2014
Jan-Feb 2014
May-Jun 2013 **Erhard Aichinger**, *Johannes Kepler University*, Linz, Austria.

Awards and Grants

- Female promotion measure "Doktorandinnen ans Rednerpult", travel grant for attending Dagstuhl Seminar 18211, May 2018
- Travel Grants for attending GETCO 2018, FLoC 2018, CAV & VMW 2016, AAA89
- Travel Grant by Amazon for attending LICS 2016 and LMW 2016
- Zonta Club Wien 1 mobility grant for female researchers, 2016
- Relocation Grant by FFG, February 2015
- Grant of Serbian Ministry of Education, Science and Technological Development for PhD students for the school year 2012/2013
- Grant of Serbian Ministry of Education for the school years 2011/2012, 2010/2011, 2009/2010, 2008/2009

Work Experience

02/2015–05/2019 **Project Assistant**, *"Formal Methods in Systems Engineering", Institut for Information Systems 184/4, Technische Universität Wien, Vienna, Austria*.
- Teaching assistant in *Formal Methods in Computer Science*, Winter semester 2018

2013–2015 **Teaching assistant**, *University of Novi Sad, Novi Sad, Serbia*.
- *Algebra 1* (Applied Mathematics, Graduate Professor of Mathematics)
- *Algebra 2* (Applied Mathematics)
- *Algebra for students of informatics* (Information Technologies)
- *Business Mathematics* (Geography, Tourism and Hotel Management)
- *Geometry 1* (Graduate Professor of Mathematics)
- *Theoretical Foundations of Mathematics 1* (Information Technologies)
- *Theoretical Foundations of Mathematics 2, Automata theory* (Information Technologies)

06/2014–08/2014 **Project Assistant**, *Institute for Algebra, Johannes Kepler University, Linz, Austria*.

Scientific Services

- Subreviewer for CONCUR 2018/2017, FMCAD 2018/2017, PSSV 2018/2017, STACS 2017, ICDCN 2017
- Co-organizing CAV 2016 Buddy System
- Student volunteer at DISC 2017, FMCAD 2017, POPL & VMCAI 2017, CAV 2016
- Co-organizing The 89th Workshop on General Algebra (AAA90)