TU WIEN Informatics

Modeling and Verification of Synchronous Fault-Tolerant Distributed Algorithms

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

eingereicht von

Ilina Stoilkovska, MSc
Matrikelnummer 1328320

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Zweitbetreuung: Privatdoz. Dipl.-Ing. Dr.techn. Josef Widder

Diese Dissertation haben begutachtet:

Nathalie Bertrand	Ahmed Rezine

Wien, 8. Februar 2021

Ilina Stoilkovska

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Modeling and Verification of Synchronous Fault-Tolerant Distributed Algorithms

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

Ilina Stoilkovska, MSc
Registration Number 1328320

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Second advisor: Privatdoz. Dipl.-Ing. Dr.techn. Josef Widder

The dissertation has been reviewed by:

Nathalie Bertrand	Ahmed Rezine

Vienna, 8th February, 2021

Ilina Stoilkovska

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Ilina Stoilkovska, MSc

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, 8. Februar 2021

Ilina Stoilkovska

Acknowledgements

This thesis was written during a time when a global pandemic caused an overwhelming amount of uncertainty and anxiety, and in many ways changed life as we know it. Although challenging in itself, the task of writing this thesis has often brought me a sense of security and certainty, and has been a welcoming distraction from the worrying circumstances of the year 2020.

A result of over five years of work, this thesis would not have been possible without the continuous guidance and support of my advisors, Florian Zuleger and Josef Widder, at every stage of my research journey. Reflecting on the years past, I am now certain that I could not have wished for better advisors. Your insightful advice and patient encouragement have been a beacon to my determination to push this work beyond the finish line. Thank you for persistently believing in me, even when I did not believe in myself.

To my colleagues and collaborators Benjamin Aminof, Igor Konnov, and Sasha Rubin, thank you for the countless discussions and constructive criticism, which greatly improved the quality of the results presented in this thesis, as well as my writing style. Many thanks to Stephan Merz, who hosted me during my research stay at Inria Nancy. My sincere gratitude goes to my examination committee, in particular to the external examiners Nathalie Bertrand and Ahmed Rezine, who agreed to review this thesis, and in addition provided valuable feedback.

Thanks to all the colleagues from the Forsyte group at TU Wien for providing a productive environment for my PhD research. Through the various meetings, workshops, and retreats I have broadened my knowledge of formal methods, and learned invaluable soft skills. Special thanks to my office mates Jure Kukovec, Marijana Lazić, and Thanh Hai Tran. My thanks also extend to the administrative staff – Juliane Auerböck, Beatrix Buhl, Eva Nedoma, and Toni Pisjak – for ensuring that everything ran smoothly. Thanks to my Informal Systems colleagues who, for the past year, have been cheering for me to finish the thesis.

I have been fortunate to be a part of the doctoral college LogiCS for the duration of my PhD studies, where I met many kind people who I am now glad to call my friends. Many thanks to Anna Prianichnikova for fostering a friendly and supportive atmosphere within the doctoral college. To all the LogiCS students, thank you for the nice memories

we made together during the past few years. Warmest thanks to Medina Andreşel, Gerald Berger, Kees van Berkel, Emir Demirović, Marijana Lazić, Anna Lukina, Tobias Kaminski, Benjamin Kiesl, Alëna Rodionova, Zeynep Saribatur, and Matthias Schlaipfer, for all the lunches, coffee breaks, conference trips, Japanese Thursdays, and Friday drinks.

To Alëna, Alina, Anna, Medina, Marijana, Matthias, Tobi, Serge, Sopo, and the Angels, thank you for always being there for me. I cannot begin to express my gratitude for your friendship. To Amr, Andi, Hanna, and Itzel, thank you for making every time we meet seem like no time passed between our reunions. To Miljana Bliznac and Ana Sokol, thank you for making me feel at home in Vienna. To the Cuca family, thank you for your warm hugs and for being my family away from home.

Lastly, my profound gratitude goes to my parents and my sister. I would not have made it here without your constant encouragement and unconditional love. Thank you for nurturing my inclinations towards STEM from my earliest years, and for incessantly supporting me in every adventure. I only regret that my father could not witness the end of my PhD journey.

Vienna, February 2021

Abstract

Distributed systems have a wide range of applications: from autonomous vehicles, via data centers, to cryptocurrencies. A distributed system consists of multiple components, called processes, which execute an algorithm locally and coordinate globally in order to reach a common goal. During the lifetime of a distributed system, some of its processes can exhibit faulty behaviors, which do not comply with the underlying algorithm, and whose severity and occurrence are unpredictable. The effects that the faults produce could cause a failure of the whole system, which is highly undesirable. To circumvent these effects and ensure that the system works correctly even in the case when a portion of its processes exhibit faulty behaviors, distributed algorithms are designed to be *fault-tolerant*.

Design of fault-tolerant distributed algorithms has been an active research field in the area of theoretical computer science for over 40 years. In the past, their applications were limited to safety-critical systems. More recently, we see a revived interest in the design and implementation of fault-tolerant distributed algorithms, which is in part due to the ubiquity of cloud computing and the development of blockchain technologies. As fault-tolerant distributed algorithms are implemented in systems where a high degree of availability and reliability is expected, the correctness of these algorithms, as well as their implementations, is paramount. Both the process of design and implementation of fault-tolerant distributed algorithms are led by humans, namely, by algorithm designers and by software engineers, respectively. To ensure correctness, algorithm designers need to provide theoretical guarantees that the system running a given fault-tolerant distributed algorithm continues to operate correctly even if a fraction of its processes is faulty, while software engineers should produce bug-free software, that correctly implements the given algorithm. Needless to say, both tasks are quite challenging, and thus susceptible to human error.

Applying automated verification techniques provides means for finding bugs at early stages in the process of designing and implementing *correct* fault-tolerant distributed algorithms. However, several characteristics, inherent to distributed systems, represent obstacles for the direct application of automated verification techniques. First, there is a lack of a formal model for fault-tolerant distributed algorithms. Typically, in the distributed systems literature, these algorithms are described using pseudocode and natural language, which is not precise for formal verification. Further, such descriptions can be ambiguous, they can contain implicit assumptions, and their pen-and-paper

proofs of correctness, often accompanying the pseudocode, can be erroneous. The second obstacle is the non-determinism that originates from the concurrent executions as well as the presence of faults, which significantly increases the global state space. Finally, fault-tolerant distributed algorithms are parameterized, in the sense that their execution is influenced by a set of parameters, whose values are a priori unknown. These parameters include the number n of processes, the number f of faulty processes, and the upper bound t on the number of faulty processes, related to one another by a *resilience condition*. Thus, an algorithm is correct if and only if it behaves correctly for all values assigned to its parameters, which implies that reasoning about the correctness of such an algorithm amounts to reasoning about the correctness of infinitely many finite instances of the algorithm. In the formal methods literature, this problem is called the *parameterized verification problem*, and is known to be undecidable in general. Many techniques for tackling the parameterized verification problem have been proposed, and they can be classified into (i) sound, but incomplete methods which are practically useful, and (ii) sound and complete methods which are able to reason about a specific class of problems.

The goal of this thesis is to increase the integration of automated verification techniques with fault-tolerant distributed algorithms, by identifying means to precisely specify them and develop techniques for automated verification of their correctness that can deal with the parameters symbolically, and thus surmount the undecidability of parameterized verification. The results we present focus on *synchronous* fault-tolerant distributed algorithms, a class of algorithms whose computations are organized in rounds, and where the processes execute the algorithm in lock-step. To be able to solve the parameterized verification challenge for these algorithms, in this thesis, we:

1. propose a modeling approach, that can be used to formally model the behavior of the processes running a given synchronous fault-tolerant distributed algorithm, operating in an environment that allows a portion of the processes to behave in a faulty way. We thus produce formal models that represent a suitable input for automated verification tools.

2. define techniques for reducing certain instances of the parameterized verification problem for synchronous fault-tolerant distributed algorithms to some decidable problem, while preserving soundness (and in some cases, completeness as well). In this way, we are able to give an answer to the parameterized verification problem by automatically solving the decidable problem.

3. show the usefulness of the proposed modeling and verification techniques by running experiments, in which we automatically verify several synchronous fault-tolerant distributed algorithms from the literature, for any number of participating processes. We have considered 19 synchronous algorithms in total, and to our knowledge, we are the first to automatically verify 18 of them, i.e., only one of the algorithms we considered was verified before.

Kurzfassung

Verteilte Systeme haben viele Anwendungsgebiete: von autonomen Fahrzeugen über Datenzentren bis hin zu Kryptowährungen. Ein verteiltes System besteht aus mehreren Komponenten, sogenannten Prozessen, die lokal einen Algorithmus ausführen und global koordinieren, um ein gemeinsames Ziel zu erreichen. Während der Lebensdauer eines verteilten Systems können einige seiner Prozesse fehlerhafte Verhaltensweisen aufweisen, die nicht dem zugrundeliegenden Algorithmus entsprechen und deren Schweregrad und Auftreten nicht vorhersehbar sind. Die Auswirkungen der Fehler können zu einem Ausfall des gesamten Systems führen, was höchst unerwünscht ist. Um diese Auswirkungen zu vermeiden und sicherzustellen, dass das System auch dann ordnungsgemäß funktioniert, wenn ein Teil seiner Prozesse ein fehlerhaftes Verhalten aufweist, sind die verteilten Algorithmen auf Fehlertoleranz ausgelegt.

Das Design fehlertoleranter verteilter Algorithmen ist seit über 40 Jahren ein aktives Forschungsgebiet im Bereich der theoretischen Informatik. In der Vergangenheit waren dessen Anwendungen auf sicherheitskritische Systeme beschränkt. In jüngerer Zeit sehen wir ein wiederbelebtes Interesse am Entwurf und der Implementierung fehlertoleranter verteilter Algorithmen, was zum Beispiel auf die Allgegenwärtigkeit von Cloud Computing und die Entwicklung von Blockchain-Technologien zurückzuführen ist. Da fehlertolerante verteilte Algorithmen für Systeme mit höhen Anforderungen bezüglich Verfügbarkeit und Zuverlässigkeit designed werden, ist die Korrektheit dieser Algorithmen sowie ihrer Implementierungen von größter Bedeutung. Sowohl der Entwurfsprozess als auch die Implementierung fehlertoleranter verteilter Algorithmen werden von Menschen geleitet, nämlich von Protokoldesignern bzw. Softwareentwicklern. Um die Korrektheit zu gewährleisten, müssen theoretische Garantien dafür gegeben werden, dass das System, auf dem ein bestimmter fehlertoleranter verteilter Algorithmus ausgeführt wird, auch dann ordnungsgemäß funktioniert, wenn ein Teil seiner Prozesse fehlerhaft ist. Softwareentwickler sollten fehlerfreie Software erstellen, die den gegebenen Algorithmus korrekt implementiert. Beide Aufgaben sind nichttrivial und daher anfällig für menschliches Versagen.

Die Anwendung automatisierter Verifikationstechniken bietet Mittel zum frühzeitigen Auffinden von Fehlern beim Entwerfen und Implementieren korrekter fehlertoleranter verteilter Algorithmen. Einige Merkmale, die verteilten Systemen eigen sind, stellen jedoch Hindernisse für die direkte Anwendung automatisierter Verifikationstechniken dar.

Erstens fehlt ein formales Modell für fehlertolerante verteilte Algorithmen. Typischerweise werden diese Algorithmen in der Literatur zu verteilten Systemen unter Verwendung von Pseudocode und natürlicher Sprache beschrieben, was für die formale Verifizierung nicht ausreichend präzise ist. Ferner können solche Beschreibungen mehrdeutig sein, sie können implizite Annahmen enthalten und ihre manuelle Korrektheitsnachweise können fehlerhaft sein. Das zweite Hindernis ist der Nichtdeterminismus, der von der gleichzeitigen Ausführung verschiedener Prozesse sowie dem Vorhandensein von Fehlern herrührt, was den globalen Zustandsraum erheblich vergrößert. Schließlich sind fehlertolerante verteilte Algorithmen parametrisiert, in dem Sinne, dass ihre Ausführung durch eine Reihe von Parametern beeinflusst wird, deren Werte a priori unbekannt sind. Diese Parameter umfassen die Anzahl n von Prozessen, die Anzahl f von fehlerhaften Prozessen und die Obergrenze t für die Anzahl von fehlerhaften Prozessen, die durch eine Ausfallsicherheitsbedingung miteinander in Beziehung stehen. Daher ist ein Algorithmus genau dann korrekt, wenn er sich bezüglich aller Werte, die seinen Parametern zugewiesen werden, korrekt verhält. Dies bedeutet, dass das Schlussfolgern über die Korrektheit eines solchen Algorithmus dem Schlussfolgern über die Korrektheit unendlich vieler endlicher Instanzen des Algorithmus gleichkommt. In der Literatur zu formalen Methoden wird dieses Problem als parametrisiertes Verifizierungsproblem bezeichnet und es ist bekannt, dass dieses Problem im Allgemeinen unentscheidbar ist. In der Literatur findet man viele Techniken zur Bewältigung des parametrisierten Verifizierungsproblems, welche in (i) korrekte, aber unvollständige Methoden, die praktisch nützlich sind, und (ii) korrekte und vollständige Methoden, die aber nur für spezifische Klassen von Problemen angewendet können, eingeteilt werden können.

Ziel dieser Arbeit ist es, die Integration automatisierter parametrisierter Verifikationstechniken und fehlertoleranter verteilter Algorithmen zu verbessern, indem Mittel zu ihrer genauen Spezifizierung identifiziert und Techniken für die automatisierte Verifizierung ihrer Korrektheit entwickelt werden. Die Ergebnisse, die wir präsentieren, konzentrieren sich auf *synchrone* fehlertolerante verteilte Algorithmen, eine Klasse von Algorithmen deren Ausführungen in Runden organisiert sind und deren Prozesse den Algorithmus im synchron ausführen. Um in dieser Arbeit die Herausforderung der parametrisierten Verifizierung für diese Algorithmen lösen zu können:

1. schlagen wir einen Modellierungsansatz vor, mit dem das Verhalten der Prozesse, die einen bestimmten synchronen fehlertoleranten Algorithmus ausführen, formal modelliert werden kann. Dabei operiert der Algorithmus in einer Umgebung, die es einem Teil der Prozesse erlaubt, sich fehlerhaft zu verhalten. Wir entwickeln daher formale Modelle, die ein geeignetes Eingabeformat für automatisierte Verifizierungswerkzeuge darstellen.

2. definieren wir Techniken zur Reduzierung bestimmter Instanzen des parametrisierten Verifizierungsproblems für synchrone fehlertolerante verteilte Algorithmen auf ein entscheidbares Problem, unter Wahrung der Korrektheit (und in einigen Fällen auch der Vollständigkeit). Für diese Instanzen können wir das parametrisierte

Verifizierungsproblem durch automatisches Lösen eines entscheidbaren Problems lösen.

3. evaluieren wir die vorgeschlagenen Modellierungs- und Verifikationstechniken mit Hilfe von Experimenten. Mittels selbst-entwickelten Prototypen von Verifikationstechniken verifizieren wir automatisch mehrere synchrone fehlertolerante verteilte Algorithmen aus der Literatur für eine beliebige Anzahl von beteiligten Prozessen. Wir betrachteten insgesamt 19 synchrone Algorithmen. Nach unserem Kenntnisstand sind wir die Ersten, die 18 von ihnen automatisch verifizieren konnten, d.h. nur einer der von uns betrachteten Algorithmen wurde zuvor verifiziert.

Contents

CHAPTER 1

Introduction

Software systems drive every aspect of our daily life. Regardless of their purpose, it is expected that these systems work correctly in all circumstances. During the lifecycle of a software system, its complexity increases, and ensuring its correctness and reliability becomes a challenging task. To avoid the single point of failure scenario, software systems often become *distributed*: their code gets replicated to multiple processes that are interconnected, and that communicate and cooperate to solve a common task. In addition to ensuring that the software run by each process works correctly, in such a distributed environment, a further challenge is to ensure that a failure of a single process does not cause the failure of the distributed system as a whole.

At the core of every distributed system lies an algorithm, run by all the processes that constitute the distributed system. In real-life executions of a distributed system, some of its processes can exhibit behaviors that do not comply with the underlying algorithm. These behaviors can occur due to power outages, human error during maintenance, adversarial attacks, etc., and their severity and frequency are unpredictable. The processes that exhibit such behaviors are called *faulty* processes. The effects that the faulty processes may have on the system as a whole could, e.g., cause the failure of the whole system, which is highly undesirable. To circumvent these effects and ensure that the system works correctly even in the case when a portion of its processes exhibit faulty behaviors, the distributed algorithms are designed to be *fault-tolerant*.

There are many examples of occurrences of real-world failures in distributed systems [BK14, DHSZ03]. Consider the following example from the cloud computing setting. In April 2011, a network configuration change triggered a service disruption for a cluster of nodes, which were part of a distributed replicated data store, used by Amazon Elastic Compute Cloud (EC2) instances in one US East Region availability zone [Ser11]. The degraded cluster of nodes then caused a disruption in the operation of the distributed data store in multiple availability zones in the US East Region, which made the EC2 service unavailable for 11 hours. Customers using the EC2 service and the affected distributed data store

service suffered unavailability issues until the AWS team was able to recover the system, which took around 80 hours. Improved fault-tolerant design could prevent the occurrence of such and similar events in large-scale distributed systems.

Design of fault-tolerant distributed algorithms is a well-established research field in theoretical computer science, with its origins dating to the 1980s. The way in which these algorithms are designed is aligned with the following approach: (i) an algorithm designer identifies the problem that should be solved by the designed algorithm, as well as the desired (safety and liveness) properties that a system of processes running this algorithm should satisfy, (ii) they specify the process behavior using pseudocode and natural language, (iii) they provide theoretical guarantees that the system running the algorithm continues to operate correctly even if a fraction of its processes is faulty, and (iv) they establish the correctness of the algorithm by manually proving that the system consisting of multiple processes (some of which may be faulty), running the same algorithm, satisfies the desired properties. Due to the ambiguity of natural language and lack of formal semantics of pseudocode, this can lead to possible incorrect interpretations when implementing the algorithm. Also, this can result in erroneous pen-and-paper proofs of correctness.

The application of automated verification techniques can assist the design of *correct* fault-tolerant distributed algorithms, and can be used to guide their correct implementations. In recent years, interest in fault-tolerant distributed algorithms has been on the rise, partially due to the rapid expansion of data centers, cloud computing, and blockchain technologies. When designing a fault-tolerant distributed algorithm, it is not uncommon for the algorithm designer to introduce bugs while trying to optimize the algorithm. Therefore, it is desirable to be able to quickly check whether an optimization did not break the desired behavior, which is where automated verification techniques can be of great benefit.

However, using automated verification techniques in this setting comes at a price, as fault-tolerant distributed algorithms pose several challenges. One challenge is the lack of a formal model for reasoning about fault-tolerant distributed algorithms. As they are specified using pseudocode and natural language by the algorithm designers, direct application of automated verification techniques is not possible. To use these techniques, a verification engineer needs to come up with a precise formal model of the underlying algorithm that resolves the ambiguities and captures the implicit assumptions originating in the pseudocode and natural language descriptions. Another challenge is the concurrency and the presence of faults, which introduce a lot of non-determinism that increases the global state space, which is why the choice of tools and techniques that can efficiently deal with the non-determinism is important. A final challenge is the scale at which the automated verification is performed. By design, the executions of fault-tolerant distributed algorithms are parameterized by the number of processes and faults. By fixing the values of these parameters, we obtain systems of different sizes that we can feed to an automated verification tool. Due to the limitations of the existing tools, often the system sizes for which we can verify a given fault-tolerant distributed algorithms are quite

small, ranging up to 10 processes. Since real-world systems often consist of hundreds of processes, we are faced with the problem of scale, which in turn motivates the application of *parameterized verification*. Here, we are interested in verifying an algorithm for *all* values of the parameters, which implies the need of reasoning about finite-state systems of all sizes, that is, reasoning about infinitely many finite-state systems. As we will discuss further in this chapter, this problem is undecidable in general. The main challenge is to develop techniques using which we can reason about the parameters symbolically, and thus avoid the need of reasoning about infinitely many finite-state systems.

In this thesis, we propose approaches for *modeling* and *parameterized verification* of *synchronous* fault-tolerant distributed algorithms. A synchronous fault-tolerant distributed algorithm runs on a fully connected network of $n \in \mathbb{N}$ processes, which communicate with each other by exchanging messages. The computations are organized in rounds. In each round, a process sends a message to all others, and updates its local state based on the messages received by other processes. The processes work synchronously in the sense that they simultaneously broadcast messages or update their local state based on the received messages. In particular, we want to automatically verify the correctness of a distributed algorithm executed by n processes, where:

1. the execution of the algorithm adheres to the synchronous execution semantics,
2. the processes operate in an environment where f processes are faulty,
3. the algorithm was designed such that it provides correctness guarantees if at most t processes are faulty,

for all values of the parameters n, t, and f that satisfy some arithmetic condition, e.g., $f \leq t < n$.

The vast majority of the existing literature on verification of fault-tolerant distributed algorithms considers *asynchronous distributed algorithms*. That is, the designed verification methods are adapted for the interleaving semantics of asynchronous systems, where executions of individual processes are arbitrarily intertwined, and where there are no guarantees about the time it takes a process to take a step, or for messages to be delivered. While this model is close to the real world, a well-known impossibility result [FLP85] imposes a limit to which problems can be solved in the asynchronous setting in the presence of faults. At the same time, there is substantial literature on *round-based distributed algorithms*, that are *not* designed for interleaving semantics. In these algorithms, the computations proceed in rounds, in which processes perform send, receive, and local computation transitions in lock-step. The verification methods developed for interleaving semantics are not directly applicable to algorithms which operate under the round-based semantics. Moreover, it is not clear if the effectiveness of these verification methods are due to the assumptions imposed by the interleaving semantics itself. Understanding why these techniques work for interleaving semantics and trying to adapt them to the round-based semantics is a challenging task.

The parameterized verification techniques for synchronous fault-tolerant distributed algorithms that we propose are inspired by techniques developed for asynchronous algorithms, and are adapted to the synchronous semantics. We are interested in verifying synchronous fault-tolerant distributed algorithms for two reasons. First, they are used in distributed real-time systems, where the underlying hardware and network infrastructure exhibits predictable timing behavior. The designers of such systems, which are implemented in e.g., cars and airplanes, are willing to exploit these timing guarantees at the algorithmic level. Second, recent work on reducing an interesting class of asynchronous algorithms to the synchronous semantics, discussed in more detail in Section 1.4, allows for application of methods designed for synchronous semantics for verifying the correctness of asynchronous systems. Thus, we believe that the methods we present in this thesis will not only be applicable to synchronous algorithms, but also to asynchronous algorithms that can be reduced to the synchronous semantics.

In the remainder of this chapter, we give a brief overview on the modeling frameworks and verification techniques for synchronous fault-tolerant distributed algorithms that we introduce in the subsequent chapters, we survey related work, and we describe the algorithms we will analyze using the proposed techniques. We will first start with an overview of the synchronous computation model in Section 1.1, and the different fault models in Section 1.2. Then, in Section 1.3, we present the research challenges that we will address in this thesis. Section 1.4 gives an overview of related work, and Section 1.5 outlines the methodologies we used to address the research challenges introduced in thesis and briefly discusses the solutions we propose for tackling them. In Section 1.6, we list the algorithms that we are interested in verifying, and discuss their characteristics. Finally, in Section 1.7, we give a list of contributions, as well as a roadmap for the rest of the thesis.

1.1 Synchronous Computation Model

We start with defining the synchronous computation model. Although this definition is common in the distributed systems literature, it is not precise for formal verification.

We assume that the processes in a synchronous distributed system are nodes in a fully connected network, with directed edges. The nodes in the network communicate by exchanging messages, using directed message channels. Each message channel holds at most one message at all times, and moreover, it is reliable, that is, there is no message loss or duplication.

Processes and Messages. Let $P = \{1, \ldots, n\}$ denote the set of *processes*, with $n \in \mathbb{N}$, and $\mathcal{M}$ the set of all possible *message types* of the messages that the processes exchange. We assume that the set $\mathcal{M}$ contains a special message type, denoted by $\perp$, which represents the null message. Let $i, j \in P$ be two processes in the synchronous network, and let $\mathit{channel}(i, j)$ denote the directed message channel from process i to process j. The process i *sends* a message of type $m \in \mathcal{M}$ to a process j, by placing

the message in the channel $channel(i, j)$. The process i *broadcasts* a message of type $m \in \mathcal{M}$ by sending the message of type m to all processes $j \in P$. The process j *receives* a message from the process i by reading the contents of the channel $channel(i, j)$ and by resetting the value of $channel(i, j)$ to $\perp$.

Each process $i \in P$ has several local variables that determine its local state. These local variables have initial values that determine the initial local state of a process $i \in P$. The types of the messages that process i sends depend on the values of its local variables. The values of the local variables of process i are updated based on the messages that process i receives.

Synchronous Computation. In the synchronous computation model, in each step of the system, the processes in P perform the following steps synchronously, in lock-step:

(SC1) every process $i \in P$ sends a message of some type $m \in \mathcal{M}$ to all processes in P (including itself), i.e., process i broadcasts the message,

(SC2) every process $i \in P$ receives the messages sent to it by the processes in P,

(SC3) every process $i \in P$ updates its local variables based on its local state and the received messages.

The sequence of these three steps is called a *round*. In some algorithms, several different rounds are combined into a *phase*. That is, we will assume that the execution of a synchronous system is organized in phases, which in turn consists of one or multiple rounds, where each round consists of the three steps (SC1) to (SC3), defined above. In the case where there is a single round per phase, the notions of phase and round coincide. In the remainder of this thesis, when we consider algorithms that have multiple rounds per phase, we will explicitly distinguish between phases and rounds. For the algorithms that have one round per phase, we abuse the notation and assume that the executions are organized in rounds.

Example 1.1. Consider the pseudocode given in Figure 1.1, which used to describe the behavior of a process $i \in P$ that runs the algorithm FloodSet [Lyn96, p. 103]. The process i has the following local variables:

- `v`, that stores the input value of process i (line 1),
- `W`, that stores the set of values that process i has seen so far. Initially, it is set to the singleton set containing the value of `v` (line 2),
- `dec`, that stores the value that process i decides on. Initially, it is set to $\perp$, which denotes that the process has not decided yet (line 3).

```
1  v := input({0, 1})
2  W := {v}
3  dec := ⊥
4  for each round 1 through t + 1 do {
5    broadcast W
6    receive values X_j from all j ∈ P for which a message arrives
7    W := W ∪ ⋃_j X_j
8    if round = t + 1 then
9      if |W| = 1 then dec := w, where W = {w}
10     else dec := v_0
11 }
```

Figure 1.1: The pseudocode of the algorithm FloodSet, code for process $i \in P$

In FloodSet, the set $\mathcal{M}$ of message types is the power set of the set $\mathtt{V} = \{0, 1\}$ of values.

Lines 1 to 3 of the pseudocode define the initial local state of the process. One iteration of the loop starting on line 4 represents one round. There is one round per phase, which means that the executions of FloodSet are organized in rounds. The algorithm runs for $t + 1$ rounds. In each round, process i performs the three steps:

1. it sends the set W of values it has seen so far to all other processes (line 5),
2. receives the messages X_j from the other processes $j \in P$ (line 6),
3. updates its set W of values with the newly received values from the other processes (line 7).

In the $(t + 1)$-st round, process i updates its variable dec as follows. If W is a singleton set, i.e., if $\mathtt{W} = \{\mathsf{w}\}$, for some $\mathsf{w} \in V$, then process i decides on the value w. Otherwise, process i decides on the default value v_0. □

1.2 Fault Model

In this thesis, we will analyze algorithms where the faults that can occur in the system are faults committed by the processes. That is, we do not consider link failures, which occur when a link between two processes fails to deliver a message. From a process failure point of view, if a link failure occurs in a system, it can be attributed either to the sender or the receiver process.

A process is *faulty*, if its behavior differs from the one specified by the algorithm. In the modeling that we propose, we suppose that the message channels are reliable. That is, if a message is lost, in our modeling of synchronous algorithms, we can deduce that either the process that sent this message or the one that received it is faulty.

There are several different fault models considered in the distributed algorithms literature. Different fault models have different severity, which is measured by the amount of uncertainty the faulty processes introduce in the execution of the algorithm. In this thesis, we will analyze synchronous algorithms that are designed to tolerate faults of the following kind:

- *crash faults*, where faulty processes stop their execution prematurely and cannot restart,
- *send omission faults*, where faulty processes omit to send some of the messages that they should send,
- *Byzantine faults*, where faulty processes act in a completely arbitrary way.

We refer the reader to the standard textbooks in distributed computing [Lyn96, AW04] for more details on other fault models.

Crash Faults. Crash faults are the simplest kinds of faults; they are also the least severe of the three kinds of faults introduced here. A process commits a crash fault in a round if it stops participating prematurely in the algorithm at any point during the execution.

The uncertainty that crash faults introduce is exhibited by the fact that a crash-faulty process might send a message only to some of the other processes in the round in which it crashes. As a crash-faulty process can stop while executing any of the three steps of a given round (i.e., steps (SC1) – (SC3) defined above), it can happen that it stops while executing step (SC1) (i.e., in the middle of a broadcast, when the process sends a message to all other processes). Thus, a crash-faulty process may only manage to send messages to a subset $P' \subseteq P$ of processes in the round in which it crashes.

Send Omission Faults. Send omission faults are more severe than crash faults. In each round, a send-omission-faulty process can omit to send messages to some of the other processes.

Thus, the uncertainty that send omission faults introduce is also due to sending a message to a subset $P' \subseteq P$ of processes. Contrary to the crash fault model, where a faulty process stops prematurely, in the send omission fault model, a faulty process continues to participate in the algorithm, by sending messages to a subset P' of processes and updating its local variables based on the messages it received. Moreover, the subset P' of processes to which a faulty process sends a message can change from round to round. Another characteristic of the send-omission-faulty processes is that they commit faults only on the sending side – they are still able to correctly receive all messages sent to them. This means that send-omission-faulty processes act in the same way as the correct processes on the receiving side, i.e., they correctly receive messages and are able to correctly update their local variables based on the received messages.

Byzantine Faults. Byzantine faults are the most severe faults that we will consider. A Byzantine-faulty process acts arbitrarily: for example, it may send different messages to different processes in the same round, it may omit to send messages, or it may update its local state to an arbitrary local state, without taking into account the messages it received. Observe that, as we have directed message channels, we assume that a Byzantine-faulty process cannot impersonate another process. That is, since a process can check which process is the sender of a message on the receiveing side, a Byzantinte-faulty process cannot send messages on behalf of another process.

Byzantine faults introduce the highest amount of uncertainty in the execution of the system, as there are no limitations or assumptions imposed on the behavior of the Byzantine-faulty processes (as opposed to crash-faulty or send-omission-faulty processes, where the behavior of the faulty processes is constrained).

Resilience Condition. To be able to deal with the different levels of severity and the uncertainty introduced by the different kinds of faults, fault-tolerant distributed algorithms are designed by taking into account specific assumptions about the number of faults that may occur in the system. These assumptions ensure that the system operates correctly, even if a portion of the participating processes exhibits faulty behavior. They are captured by a so-called *resilience condition*, which is an expression over the number n of processes, the upper bound t on the number of faults, and the number f of actual faults.

The restrictiveness of the resilience condition depends on the fault model. For most of the algorithms that tolerate crash faults, the resilience condition is expressed as the inequality $n > t \geq f$. This means that, for an algorithm to tolerate crash faults, it is enough that a single process is correct. The fact that send omission faults are more severe than crash faults is also reflected in the resilience condition: for some algorithms tolerating send omission faults, we have $n > 2t \wedge t \geq f$. Thus, some algorithms require that the majority of processes is correct in order to tolerate send omission faults. Finally, as Byzantine faults are the most severe, algorithms designed to tolerate Byzantine faults need a higher ratio of correct to faulty processes. For most of the algorithms that tolerate Byzantine faults, it is assumed that $n > 3t \wedge t \geq f$. This implies that to be able to tolerate Byzantine faults, in most cases, more than two-thirds of the participating processes have to be correct.

The constraints on the number of processes and faults imposed by the resilience condition are non-trivial, and are an area of study in the field of distributed systems. Getting the correct resilience condition for an algorithm often requires rigorous mathematical proofs. For example, the necessary and sufficient bound $n > 3t$ for Byzantine faults was first derived and proved in [PSL80].

Example 1.2. The FloodSet algorithm, whose pseudocode is given in Figure 1.1, is designed to tolerate at most t crash faults, under the resilience condition $n > t \geq f$.

Consider an execution of the algorithm where there are $n = 3$ processes and $t = f = 1$ faults. The algorithm runs for $t + 1 = 2$ rounds. Let the set P contain the processes i_1, i_2, i_3, and the default value $\mathsf{v}_0 = 0$. Initially, at the beginning of round 1, let the value of the variable $\mathtt{W}$ be $\{0\}$ for i_1 and $\{1\}$ for i_2 and i_3. Suppose that i_1 crashes in round 1 and manages to send its set $\mathtt{W}$ only to i_2, as depicted below:

process	initially	rnd 1	after i_1 crashes	rnd 2	after i_2, i_3 decide
i_1	$\mathtt{W} = \{0\}$, $\mathtt{dec} = \bot$		crashed		crashed
i_2	$\mathtt{W} = \{1\}$, $\mathtt{dec} = \bot$	$\longrightarrow$	$\mathtt{W} = \{0, 1\}$, $\mathtt{dec} = \bot$	$\longrightarrow$	$\mathtt{W} = \{0, 1\}$, $\mathtt{dec} = 0$
i_3	$\mathtt{W} = \{1\}$, $\mathtt{dec} = \bot$		$\mathtt{W} = \{1\}$, $\mathtt{dec} = \bot$		$\mathtt{W} = \{0, 1\}$, $\mathtt{dec} = 0$

At the beginning of round 2, the value of $\mathtt{W}$ is $\{0, 1\}$ for i_2 and $\{1\}$ for i_3. In round 2, process i_2, which is correct, sends its value $\mathtt{W} = \{0, 1\}$ to i_3, and they both decide 0, which is the default value. □

1.3 Research Challenges

Formalization Challenge. The specification of the process behavior using pseudocode (as is done in Figure 1.1), although well understood in the distributed system community, is not precise for formal verification. Our goal is to propose a *modeling approach* for synchronous fault-tolerant distributed algorithms, that will precisely capture the process behavior and as well as the faulty environment in which the processes operate, and that will produce formal models suitable for verification.

Verification Challenge. Given a synchronous fault-tolerant distributed algorithm, once we produce a formal model which is suitable for verification, the next step is to verify its correctness. To do so, we will check if the formal model satisfies the properties of the given algorithm, which are encoded as formulas in some temporal logic, such as LTL for example. There are two decision problems that we will address in this thesis: *fixed-size model checking* and *parameterized model checking*, both formally stated below.

1.3.1 Modeling Approach

To tackle the formalization challenge, we propose a *modeling approach* which:

- gives a formal definition of the process pseudocode using a *process specification*, denoted by $\mathsf{proc}(n, t, f)$,
- formalizes the assumptions imposed by the fault model using an *environment specification*, denoted by $\mathsf{env}(n, t, f)$,
- builds a formal model of the given synchronous fault-tolerant distributed algorithm, by composing n process specifications and an environment specification in a *synchronous system specification*, denoted by $\mathsf{Sys}(n, t, f)$.

The pseudocode of a synchronous fault-tolerant distributed algorithm is parameterized by the parameters: n, the number of processes, and t, the upper bound on the number of faults. The parameter f, representing the actual number of faults, is not visible to the processes, but only to the environment. However, when modeling the process behavior formally using a process specification, the parameter f may be needed to describe some process behaviors. Hence, we have that both the process and environment specifications, as well as the synchronous system specification, are parameterized by the parameters n, t, and f.

Process Specification. While the pseudocode specifies the process variables and how they are updated, it does not specify how the broadcast is performed, or how the processes receive messages. A process specification $\mathsf{proc}(n, t, f)$ formalizes the pseudocode by giving a formal definition of:

1. the local variables of a process, as well as the initial values of these local variables,
2. the receive variables of a process, where the messages received from other processes are stored,
3. how a process generates messages that it sends to the other processes,
4. how a process updates its variables.

The process specification $\mathsf{proc}(n, t, f)$ should precisely encode the process behavior, and allow for efficient application of formal verification techniques.

Environment Specification. Another modeling issue is the formalization of the behavior of the faulty processes in the system. As the pseudocode does not specify the behavior of the faulty processes, the environment specification $\mathsf{env}(n, t, f)$ is used to encode this implicit process behavior. That is, given a process specification $\mathsf{proc}(n, t, f)$ and a fault model, we propose an environment specification $\mathsf{env}(n, t, f)$ associated with $\mathsf{proc}(n, t, f)$, which defines how the faulty processes behave in the given fault model. Moreover, the environment specification $\mathsf{env}(n, t, f)$ encodes how the correct processes are affected by the behavior of the faulty ones. In other words, it describes the semantics of the process code when it is executed in an unreliable distributed environment.

Synchronous System Specification. Once we define the process specification, and capture the assumptions imposed by the fault model in the environment specification for a given synchronous fault-tolerant distributed algorithm, we proceed by defining a synchronous system specification, which represents a formal model of the given algorithm. In this thesis, we will specify the behavior of all n processes in the system using the same process specification.

A *synchronous system specification* is the synchronous parallel composition

$$\mathsf{Sys}(n,t,f) = \underbrace{\mathsf{proc}(n,t,f) \parallel \cdots \parallel \mathsf{proc}(n,t,f)}_{n} \Big\| \mathsf{env}(n,t,f)$$

consisting of n processes that follow the process specification $\mathsf{proc}(n,t,f)$, operating in an environment that follows the environment specification $\mathsf{env}(n,t,f)$, which ensures that f processes are faulty, where $f \leq t$.

The synchronous system specification $\mathsf{Sys}(n,t,f)$ is used to represent all the systems obtained by instantiating the parameters n,t,f with values $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$, respectively. That is, given values $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f})$ denotes a *transition system* that models the behavior of a concrete instance of the algorithm modelled by the system specification $\mathsf{Sys}(n,t,f)$. For the algorithms that we analyze in this thesis, instantiating $\mathsf{Sys}(n,t,f)$ with values $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ results in a *finite-state* transition system $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f})$. Hence, in the remainder of this thesis, we will use the synchronous system specification $\mathsf{Sys}(n,t,f)$ to represent the infinite family

$$\{\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f}) \mid \mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N} \text{ satisfy the resilience condition}\}$$

of finite-state transition systems $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f})$.

Example 1.3. A process specification $\mathsf{proc}(n,t,f)$ for the pseudocode of FloodSet in Figure 1.1 should define: the process variables $\mathtt{v},\mathtt{W},\mathtt{dec}$, how the broadcast in line 5 is performed, how the process receives messages in line 6, and how the process updates the variables $\mathtt{W}$ and $\mathtt{dec}$. The environment specification $\mathsf{env}(n,t,f)$ should encode the behavior of crash-faulty processes. That is, $\mathsf{env}(n,t,f)$ should define how the faulty processes crash, and which subset of processes they send a message to while crashing. □

1.3.2 Verification Problems

Given a synchronous system specification $\mathsf{Sys}(n,t,f)$, that represents a formal model of an algorithm, and a temporal formula ϕ, that encodes one of its properties, we state the following two decision problems:

1. in *fixed-size model checking*, we want to verify whether the temporal formula ϕ holds in an instance of the given synchronous system specification $\mathsf{Sys}(n,t,f)$, obtained by fixing the values of the parameters n, t, and f.

2. in *parameterized model checking*, we want to verify whether the temporal formula ϕ holds in all instances of a given synchronous system specification $\mathsf{Sys}(n,t,f)$, that is, for all values of the parameters n, t, and f.

Formally, the fixed-size model checking problem is defined as follows:

Fixed-Size Model Checking Problem FSMCP	
Input:	• synchronous system specification $\mathsf{Sys}(n, t, f)$ • temporal formula ϕ • values $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition
Question:	Does ϕ hold in $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$?

The fixed-size model checking problem is useful when we are interested in verifying the correctness of a synchronous fault-tolerant distributed algorithm, for which we know the values of the parameters a priori. The fixed-size model checking problem can be reduced to the finite-state model checking problem. As the resulting transition system $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, obtained by instantiating $\mathsf{Sys}(n, t, f)$ with the values $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$, is a finite-state system, it can be passed as input to a model checker, together with the formula ϕ. The output of the model checker is used as the answer to the fixed-size model checking question.

However, it is not always the case that the values of the parameters are known a priori. Furthermore, for large values of the parameters, the model checker used to solve the finite-state model checking problem might run into state space explosion. As we will see in Chapter 3, where we give experimental results for the fixed-size model checking problem, for some algorithms the model checker we used ran into state space explosion for $\mathsf{n} > 5$. Thus, it is more desirable to solve the parameterized model checking problem, formally defined as follows:

Parameterized Model Checking Problem PMCP	
Input:	• synchronous system specification $\mathsf{Sys}(n, t, f)$ • temporal formula ϕ
Question:	Does ϕ hold in $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, for all values $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition?

The parameterized model checking problem is undecidable in general. Still, for some classes of inputs, the parameterized model checking problem is decidable. The decidability often depends on the characteristics of the input, e.g., the network topology (i.e., if the underlying communication graph is fully connected, or if the processes are arranged in some particular configuration, such as a ring), the conditions that the processes check in order to update their local variables, or the temporal logic used to express the property ϕ. For inputs that do not fall into any decidable class, there exist sound but incomplete techniques and semi-decision procedures that can be used to solve the parameterized model checking problem.

1.4 State of the Art

The research presented in this thesis lies in the intersection of two fields from theoretical computer science, namely, distributed computing and formal methods. In this section, we give a brief overview of existing literature related to the results proposed in this thesis.

1.4.1 Fault-Tolerant Distributed Algorithms

So far, in this chapter, we discussed the synchronous computation and the various fault models. These topics are discussed in more details in several textbooks focused on distributed computing, such as [AW04, Lyn96, Ray10]. The problems in distributed computing of interest to this thesis, such as consensus, k-set agreement, non-blocking atomic commit, and authenticated broadcast were introduced in the 1980s. The synchronous consensus problem was introduced by Lamport, Shostak, and Pease in [PSL80, LSP82] as the "Byzantine Generals Problem". The k-set agreement problem was introduced by Chaudhuri, Herlihy, and Lynch in [CHLT00]. The non-blocking atomic commit problem originated in database theory research [Ske81, BHG87]. The simulated authenticated broadcast problem was introduced by Srikanth and Toueg in [ST87]. We will discuss these problems in distributed computing in more detail in Section 1.6.

More recently, the expansion of blockchain technologies has motivated new research in distributed computing. Bitcoin [Nak08] proposed proof of work as a new approach for solving consensus. Often criticized for its environmental impact, proof of work requires the processes participating in consensus to use a huge amount of computing power. Proof of stake was introduced as a more environmentally-friendly variant of proof of work, where the participating processes are required to put a portion of their digital cryptocurrency assets at stake in order to participate in consensus. Among others, the Tendermint blockchain system [Buc16] uses a proof of stake consensus algorithm, which draws ideas and inspirations from traditional Byzantine fault-tolerant distributed algorithms.

1.4.2 Model Checking

Proposed in the early 1980s, independently by Clarke and Emerson [CE81] and Queille and Sifakis [QS82], model checking has been an active area of research in the field of formal methods. It is the subject of numerous textbooks, such as [CGL94, BK08], as well as a recently published handbook [CHVB18]. The main idea of model checking is to use formulas in some temporal logic, such as LTL, CTL, or CTL*, to express the correct behavior of a system. To do so, one typically needs to give a formalization of *safety* properties, expressing that nothing bad ever happens in a correct system, *liveness* properties, expressing that something good eventually happens, and *fairness* properties, expressing that some desirable behavior is recurring. Formally, the model checking problem is stated as follows: given a finite-state transition system M and a temporal logic property ϕ, the model checking question is whether the system M satisfies the property ϕ, namely, whether $M \models \phi$.

If the model checking question is answered positively for a given property ϕ and a system M, we can conclude that the property ϕ holds in the system M. Otherwise, if the model checking question is answered negatively, a counterexample is produced, which is a witness that the property ϕ is violated in the system M. For a safety property, the counterexample execution is finite, while for a liveness property the counterexample execution is an infinite execution.

In order to answer the model checking problem, early implementations of model checkers (e.g., [CES86]) exhaustively explored the state space of the system M. This technique is scalable for systems whose state space is relatively small; for complex systems with large state space, exhaustively exploring it runs into a problem called *state space explosion*. In systems consisting of multiple components, such as those used to model a system of n processes running a distributed algorithm, the state space grows exponentially with the number of processes. Several techniques have been proposed over the years in order to overcome the state explosion problem, and we mention some of them in the following.

Partial order reduction [Val89, God90, Pel93] is used to reason about the independence of local transitions in asynchronous systems with multiple components, and whether two local transitions can be executed in swapped order. By identifying local transitions that can commute without changing the global outcome, the state space that the model checker needs to explore is reduced, and only considers representatives of different interleavings that lead to the same global outcome.

Symbolic model checking [BCM+92] represents the states of a system *symbolically*, while exploiting the similarities between states in the state space. This drastically increased the number of states that can be handled by a model checker, and allowed for practical verification of both existing industrial hardware and software systems. The initial symbolic representation was based on binary decision diagrams (BDDs) [Jr.78]; later, SAT-based [BCCZ99] and interpolation-based [McM03] techniques were introduced. The SAT-based bounded model checking introduced in [BCCZ99] encodes executions of a fixed length k in the system M in propositional logic, and uses efficient SAT solvers to search for a counterexample of the property ϕ up to length k.

The goal of abstraction techniques is to produce an abstract system $\widehat{M}$, given a transition system M as input. The states of the abstract system $\widehat{M}$ store less information than those of the system M, but still enough information in order to ensure that model checking a property ϕ in the abstract system $\widehat{M}$ is a sound procedure for model checking the property ϕ in the system M. Existential abstraction [CGL94] builds an abstract system $\widehat{M}$ which is an overapproximation of the concrete system M, such that every behavior of the concrete system can be mapped to a behavior in the abstract system. Predicate abstraction [GS97] constructs abstract systems whose states are valuations of Boolean predicates expressed over the variables in the concrete states. Often, the abstract system $\widehat{M}$ may have behaviors which are not reproducible in the concrete system. When such non-reproducible behaviors lead to a counterexample to a property ϕ in the abstract system $\widehat{M}$, the abstraction can be refined [CGJ+00] in order to eliminate them from the abstract system.

1.4.3 Parameterized Model Checking

A system consisting of an arbitrary number of components is called a *parameterized system*, where its parameters typically include the number n of components, often called system size. The parameterized model checking problem, as introduced for synchronous

system specifications in Section 1.3.2, is formulated as the following question: Does the parameterized system $M(n)$ satisfy a temporal logic property ϕ, for any system size? Thus, when reasoning about parameterized systems, one needs to reason about infinitely many systems, obtained by instantiating the parameters with specific values. The parameterized model checking problem is undecidable in general [AK86]. In [Suz88], the undecidability of the parameterized model checking problem is shown for parameterized systems consisting of n identical processes in a ring topology.

Typical methods that help prove decidability are symmetry reduction [EN95, EN03, ES96] and cutoffs [EN95, EN03, CTTV04]. With symmetry reduction, the goal is to use the symmetry of the parameterized system $M(n)$ (that is, the fact that all components in $M(n)$ behave the same), and show that checking ϕ for all n processes is equivalent to checking ϕ for a single process or a pair of processes. The idea of cutoffs is to show that there exists a small system size $\mathsf{n}' \in \mathbb{N}$, such that any instance $M(\mathsf{n})$ of the parameterized system $M(n)$, for $\mathsf{n} \in \mathbb{N}$, can be mapped to the small system $M(\mathsf{n}')$.

To show undecidability for a given class of inputs, i.e., parameterized systems and temporal formulas, a common approach is a reduction from the halting problem of two-counter machines [Min67]. The idea is to describe the behavior of a two-counter machine using a parameterized system $M(n)$, that has the same properties as the inputs of interest, and encode the halting question using the temporal property ϕ. We will show an undecidability result by reduction from two-counter machines in Chapter 5.

An overview of decidability and undecidability results for classes of asynchronous systems is given in [BJK+15]. For synchronous systems, Emerson and Namjoshi [EN96] show decidability of the parameterized model checking problem for a specific class of inputs and temporal formulas. Synchronous fault-tolerant distributed algorithms do not fall in the class of inputs for which [EN96] show decidability. Therefore, in this thesis we develop techniques for answering the parameterized model checking problem for these algorithms.

Abstraction techniques are applied in order to build an abstract system $\widehat{M}$ that captures the behavior of every instance $M(\mathsf{n})$ of the parameterized system $M(n)$, where $\mathsf{n} \in \mathbb{N}$. Abstraction is a sound, but incomplete method for verifying the correctness of parameterized systems. That is, the correctness of the abstract system then implies the correctness of the parameterized system. Studied in the field of abstract interpretation [CC77], various abstraction methods have been applied to reason about parameterized systems, and we mention several which are related to the work presented in this thesis. A general method for applying abstractions to parameterized systems was proposed in [KP00]. Counter abstraction [PXZ02] aims at reducing the state space by producing an abstract system where in each state, it is recoded whether there are zero, one, or more components (processes) in a given local state. Environment abstraction [CTV06] combines ideas from predicate abstraction and counter abstraction, in order to allow support for parameterized model checking of systems where the components are not symmetric and can be infinite-state. Both counter and environment abstraction were used for verification of mutual exclusion protocols. Compositional model checking [CLM89, McM99]

is a technique for reasoning about properties of parameterized systems consisting of identical components, by reasoning about the local properties of the components. The CMP method [CMP04, Krs05] combines abstraction, compositional model checking, and counterexample-guided abstraction refinement for verification of safety properties, that has been applied to industrial cache-coherence protocols.

Other techniques that have been developed for tackling the parameterized model checking problem include backward reachability based on well-quasi orderings [ACJT96] and regular model checking [BJNT00]. The former was used to verify safety of finite-state transition systems over infinite data domains, by reducing it to backwards reachability checking using a fixpoint computation, provided that the systems given as input are well-structured. Regular model checking is an automata-based framework for modeling and verification of infinite-state and parameterized systems, where the states are modeled as words in a regular language, and where the transitions are finite state transducers, mapping one state (word) to another.

1.4.4 Verifying Fault-Tolerant Distributed Algorithms

Parameterized verification of fault-tolerant distributed algorithms has recently been addressed with a wide range of techniques, most of which focus on asynchronous distributed algorithms. We survey both existing parameterized model checking techniques and deductive verification techniques, based on producing mechanized proofs using proof assistants.

Several consensus algorithms were verified for small system sizes in [TS11,NTK12,DTT14], by applying model checking to the fixed size instances of up to, e.g., six processes. In the parameterized case, a framework for verifying fault-tolerance of distributed protocols based on regular model checking was proposed in [FKL08]. In this framework, the fault model specification is separate from the process specification, which is similar our idea of separating the process and environment specifications, and keeping the system specification modular. The approach was manually applied to verify the correctness of an authenticated broadcast protocol that tolerates crash faults in the parameterized case. Parameterized model checking of safety properties for fault-tolerant distributed algorithms using counter abstractions and SMT solvers was proposed in [AGOP16]. Using this approach, the authors automatically verified two authenticated broadcast protocols, operating under different fault models, namely crash, send-omission, general omission, and Byzantine faults.

The threshold automata framework [KVW14, KVW17] was introduced for modeling and parameterized model checking of *asynchronous* fault-tolerant distributed algorithms. In [KVW14], a bound on the diameter for reachability properties was computed, and used as a completeness threshold for SAT-based bounded model checking over an abstract system, obtained using parametric interval abstraction [JKS+13]. SMT-based bounded model checking was applied directly to systems of threshold automata in [KVW15], without the need for abstraction, which eliminated spurious counterexamples and im-

proved efficiency. Safety and liveness verification of systems of threshold automata was proposed in [KLVW17]. In [BKLW19], threshold automata were extended in order to be able to model and verify randomized asynchronous fault-tolerant distributed algorithms. The tool ByMC [KW18] implements the techniques for parameterized model checking of asynchronous threshold automata. The full expressive power and the decidability of various decision problems of different kinds of threshold automata was studied in [KKW18]. Recently, the complexity of verification problems for threshold automata was established [BEL20]. In this thesis, we propose the extension of threshold automata to synchronous systems, and study the parameterized verification problem for systems of synchronous threshold automata.

The Heard-Of model [CS09] was proposed as a formalization framework for round-based, message passing distributed algorithms, where the computation and fault models are captured by so-called communication predicates. This framework enables a systematic encoding of (asynchronous, synchronous, or partially synchronous) round-based algorithms and facilitates the comparison between different algorithms. For example, a syntactic characterization of algorithms that solve consensus in a fragment of the heard-of model was proposed in [BW20]. Several parameterized verification techniques are designed for algorithms formalized in the heard-of model. For partially synchronous consensus algorithms, expressed in an extension of the heard-of model, [DHV$^+$14] introduced a consensus logic and (semi-)decision procedures for verifying user-provided invariants. In [MSB17], a characterization of partially synchronous consensus algorithms in the heard-of model was given. Based on this characterization, the authors proved cut-off theorems specialized to the properties of consensus: agreement, validity, and termination. More recently, [GREP20] proposed an approach for parameterized verification of safety properties of round-based algorithms, that can be expressed in the heard-of model, by combining overapproximation and backward reachability analysis.

Many fault-tolerant distributed algorithms have been formalized using the specification language TLA+ [Lam02], e.g., [GL03, Lam11, MAK13]. As TLA+ is equipped with an explicit state model checker, TLC [YML99], and a proof system, TLAPS [CDLM10], often TLC is used to debug the TLA+ specification of a given algorithm, for small and fixed system sizes, and show the correctness of the algorithm in the parameterized case by writing a machine-checkable proof in TLAPS. In [TKW20], a cutoff result for failure-detection algorithms was presented, the algorithm from [CT96] was specified using TLA+, and TLC and APALACHE [KKT19] (a new symbolic model checker for TLA+) were used to verify its correctness for the cutoff size. In this thesis, we will apply TLC to a specification of an abstract system, encoded using TLA+. Other theorem provers have also been used to certify the correctness of fault-tolerant distributed algorithms. PVS [ORS92] was used in [LR93], where a bug in an already published synchronous consensus algorithm tolerating hybrid faults was reported, and in [SWR02] to verify Byzantine agreement with link failures, in addition to process failures. Isabelle/HOL [NPW02] was used to certify the correctness of several protocols encoded in the heard-of model in [CS09, DM12].

These semi-automated proofs require a great amount of human intervention and under-

standing of the distributed algorithms. IVy [PMP+16, MP20] is an interactive verification tool, whose goal is achieving higher automation when producing a formal proof of a distributed algorithm, by reducing the amount of human guidance as much as possible. The main idea of the IVy methodology is to encode the algorithms in the effectively-propositional fragment (EPR) of first-order logic. It is not always straightforward to encode distributed algorithms and their verification conditions in EPR, but once it is done, the verification condition check is fully automatic.

1.4.5 Verifying Distributed Systems

In the previous section, we surveyed methods for verifying fault-tolerant distributed algorithms. We now discuss efforts developed to verify the correctness of implementations of fault-tolerant distributed algorithms.

Systems that implement fault-tolerant distributed algorithms are very complex and increasingly hard to get right. With increasing numbers of systems that implement fault-tolerant distributed algorithms [Bur06, JRS11, MAK13], there is an interest in developing tool support for eliminating flaws in distributed algorithms and their implementations by means of automated verification.

IronFleet [HHK+17] implements a variant of the algorithm MultiPaxos [Lam98] in Dafny [Lei10], which allows for Hoare-logic style program verification. Verdi [WWP+15] verifies an implementation of the Raft protocol [OO14] using the Coq proof assistant [dt04], and translates the Coq proof into a verified implementation in OCaml. Chapar [LBC16] also uses the Coq to OCaml translation to obtain a verified implementation of a distributed key-value store. When verifying implementations of distributed systems, one has to be very careful about the assumptions, about the calls to unverified external libraries, and about the correctness of the specifications themselves. An empirical study [FZWK17] analyzed the three verified implementations produced by IronFleet, Verdi, and Chapar, and reported existence of bugs in the interfaces between the verified code and the unverified external libraries or operating system.

Recently, it was observed in order to automatically verify asynchronous distributed programs, one can define reductions [EF82, CCM09] in order to reduce reasoning about an asynchronous system to reasoning about a synchronous system, equivalent to the original one [DDMW19]. Due to the non-determinism that comes from the faults and the asynchronous computation model, these synchronized versions of the asynchronous programs have different fault and computation semantics to those considered in this thesis. We list some of the approaches based on the idea of reductions. PSync [DHZ16] was introduced as a domain-specific language for specifying and implementing fault-tolerant distributed algorithms, which is based on the heard-of model and can be translated to the consensus logic of [DHV+14], and thus the same invariant checking techniques can be applied. A decision procedure for invariant checking of a given asynchronous message-passing program, whose computations can be reduced to computations of an equivalent round-based program with a bounded number of send operations per round, is presented

in [BEJQ18]. Several methods based on Lipton reduction [Lip75] were introduced in order to reduce asynchronous programs to other programs, for which reasoning is easier. For example, [BvGKJ17] introduces canonical sequentialization, which is a sequential program, equivalent to a given asynchronous message-passing program, and [KEH+20] defines inductive sequentialization, a sequential program to which a given asynchronous program is reduced by combining reduction, abstraction, and inductive reasoning. Asynchronous programs are reduced to equivalent synchronous programs in [KQH18], and their invariants checked using a dedicated model checker. Another technique for reducing asynchronous to synchronous programs was given in [vGKB+19], where the correctness of the obtained synchronous program is established using Hoare-style verification conditions and SMT solvers.

1.5 Methodological Approach

As discussed in Section 1.3, we address two research challenges: the formalization and the verification challenge for synchronous fault-tolerant distributed algorithms. In this section we outline the approaches we propose in order to address these two challenges.

1.5.1 Formalization challenge: Proposed models

Producing a formal model of a system is a very important step for verification. Often, the right balance between abstraction and detail has to be struck in a model in order for it to be suitable for efficient verification. In Section 1.3.1, we discussed a modeling approach based on process and environment specifications, which allow us to formally specify the behavior of a process as an entity operating in a faulty environment. The system specification is a composition of n processes and an environment that captures the constraints imposed by the fault model and message communication.

The following are ideal features of process and environment specifications, and should be taken into account when defining them as formalizations for synchronous fault-tolerant distributed algorithms. On the one hand, there should be a clear mapping between the process and environment specifications and the pseudocode or English description of the algorithm in question. On the other hand, the specifications should be precise and compact, that is, they should formalize the behavior of the processes and the environment of the system correctly, while only keeping relevant information in the model. Finally, the system specification, obtained by composing n processes and an environment, should be a suitable input to automated verification techniques.

When defining process and environment specifications, an important ingredient to take into account are the algorithms we are interested in modeling. The way in which the process and environment specifications are defined typically depends on key features of the algorithms, such as their communication and fault models. For example, we will model algorithms that communicate using message passing, and thus have to formalize the way in which the processes exchange messages in the process specification. In Section 1.6

we discuss the algorithms we picked as benchmarks, and which were modeled using the process and environment specifications presented in this thesis. Our set of benchmarks contains both simple and complicated algorithms, which are designed to tolerate either crash, send omission, or Byzantine. We also consider algorithms that are designed to tolerate hybrid faults, that is, whose fault model can be obtained by combining two or more different fault models (and which tolerate faults with different levels of severity). In particular, for hybrid-tolerant algorithms, we consider the variant tolerating both send omission and Byzantine faults. There certainly are many synchronous fault-tolerant distributed algorithms that we do not consider as benchmarks in this thesis, as the design of distributed algorithms is an active research field. Still, the benchmarks we picked are diverse, to the extent that our techniques for parameterized model checking can be applied.

In this thesis, we will introduce three different process and environment specifications:

- process and environment variables and functions, defined in Chapter 2,
- synchronous threshold automata, defined in Chapter 4,
- synchronous threshold automata with receive variables, defined in Chapter 6.

We briefly list their features below, and discuss their benefits and drawbacks.

Process and Environment Variables and Functions. In Chapter 2, we propose a process specification defined by *process variables* and *process functions*. The process variables store values local to a process, while the process functions define the way in which the values of the process variables get updated. The environment specification which accompanies this process specification is defined by *environment variables*. The environment variables include, e.g., the round number and the set of faulty processes. The environment variables that capture the non-deterministic occurrence of faults in the system (such as the set of faulty processes) are updated non-deterministically. In this case, the synchronous system specification represents the infinite family of *synchronous transition systems*, whose variables store the values of the process variables for each process, and the value of the environment variables. The transition relation uses the process functions to define the variable updates from one state to another.

While the process specification defined in such a way is very close to the original pseudocode description of the algorithm, the obtained synchronous system specification is still parameterized, and needs to be abstracted in order to obtain a system which is suitable for model checking. Further, the environment specification we propose is tailored to capture environments of algorithms that tolerate crash faults, and its extension to other fault models is not immediate. Naturally, the question that arises is whether we can propose models that allow us to encode algorithms which tolerate other fault models, such as send-omission or Byzantine faults.

Synchronous Threshold Automata. In Chapter 4, we introduce *synchronous threshold automata*, which are process specifications that model the process behavior at a higher level of abstraction than the process variables and functions. In synchronous threshold automata, the values of the process local variables are encoded using *locations*, and the variable updates (that is, the move from one location to another) is encoded by *rules*. The rules are guarded using linear integer arithmetic expressions over the number of processes in a given set of locations and the parameters. The environment specification for synchronous threshold automata is defined by an *environment assumption*, which is a constraint over the number of processes allowed in certain locations. For example, in a synchronous threshold automaton, we might have locations that encode that a process is faulty. The environment assumption constrains the number of processes in faulty locations to be less than or equal to the number f of faults. The synchronous system specification induced by a synchronous threshold automaton represents the infinite family of *counter systems*. A counter system stores, for each location of the synchronous threshold automaton, the number of processes currently in that location. The transition relation uses the rules of the synchronous threshold automaton to update the counters, while maintaining the environment assumption.

Encoding the process behavior using synchronous threshold automata allowed us to model synchronous distributed algorithms that tolerate crash, send omission, Byzantine, and hybrid faults. As we will see in Chapter 5, we developed efficient verification techniques for algorithms whose process behavior is encoded using synchronous threshold automata.

Synchronous Threshold Automata with Receive Variables. In the synchronous threshold automata, defined in Chapter 4, the guards express conditions over the number of sent messages. However, the pseudocode of the algorithms we consider as benchmarks in this thesis are predicated by the number of received messages rather than by the number of sent messages. For example, in Figure 1.1, in line 7, the process updates the values stored in its set `W` with the set of messages received from other processes.

In the synchronous computation model, we observe that each process receives all the messages sent by correct processes. As there are faulty processes in the system, the number of received messages may deviate from the number of correct processes that sent a message. To obtain a process specification in the shape of a synchronous threshold automaton that faithfully models the process behavior, described by the pseudocode, often a non-trivial manual abstraction step needs to be performed. This abstraction step translates conditions over the receive variables (occurring in the pseudocode) to conditions over the number of sent messages (occurring on the guards of the rules in the synchronous threshold automaton, and represented by the number of processes in a given location).

To automate this step, in Chapter 6 we extend synchronous threshold automata with *receive variables*. The receive variables are counters that count the number of received messages of each message type. This process specification is closer to the pseudocode and it is easier to encode manually. The synchronous system specification, induced

by a synchronous threshold automaton with receive variables, is an infinite family of finite-state *synchronous transition systems*, whose states store for every process the location the process is in, as well as the values of its receive variables.

However, the verification techniques that we develop for synchronous threshold automata without receive variables are not directly applicable to synchronous threshold automata with receive variables. Therefore, in Chapter 7, we develop a *quantifier-elimination based* procedure that eliminates the receive variables, and given a synchronous threshold automaton with receive variables creates a corresponding synchronous threshold automaton without receive variables. We then establish a correspondence between the synchronous transition system, induced by a given synchronous threshold automaton with receive variables, and the counter system, induced by the obtained synchronous threshold automaton where the receive variables are eliminated.

1.5.2 Verification Challenge: Techniques for Parameterized Model Checking

When we are faced with the challenge of solving the parameterized model checking problem for given synchronous system specification $\mathsf{Sys}(n, t, f)$ and temporal formula ϕ as input, we first need to establish if the problem is decidable or undecidable for the given input. We discussed several techniques for establishing decidability or undecidability of the parameterized model checking problem in Section 1.4.3.

Still, even if we can show undecidability of the parameterized model checking problem for the given $\mathsf{Sys}(n, t, f)$ and ϕ as input, we can apply procedures that allow us to give a positive answer to the parameterized model checking problem. In the remainder of this section, we will give an overview of the approaches for answering the parameterized model checking question that we propose in this thesis.

Abstraction. Abstraction is a sound, but incomplete technique for solving the parameterized model checking problem. It aims at applying sound transformations to the parameterized system specification given as input in order to obtain a system specification that does not depend on the parameters. The obtained *abstract* system specification represents a single *abstract* finite-state transition system, which simulates the behavior of every *concrete* finite-state transition system, represented by the original parameterized system specification. The existence of *simulation* [BK08] ensures that for every transition in some concrete finite-state system, there exists a transition in the abstract finite-state system. That is, the abstract finite-state system is an *overapproximation* of every concrete finite-state transition system.

Let $\mathsf{Sys}(n, t, f)$ be a synchronous system specification representing the infinite family $\{\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \mid \mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ satisfy the resilience condition$\}$ of finite-state transition systems $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$. Let α be an *abstraction mapping*, which applied to the system specification $\mathsf{Sys}(n, t, f)$ results in an abstract system specification $\widehat{\mathsf{Sys}}$, where $\widehat{\mathsf{Sys}}$ is an abstract

finite-state transition system that simulates the behavior of every finite-state transition system $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f})$.

If $\widehat{\mathsf{Sys}}$ simulates every $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f})$, the parameterized model checking problem is reduced to the finite-state model checking problem as follows. Given a synchronous system specification $\mathsf{Sys}(n,t,f)$, an abstraction mapping α, and a temporal formula ϕ, we have:

$$\widehat{\mathsf{Sys}} \models \phi \quad \text{implies} \quad \mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f}) \models \phi \text{ for all } \mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N} \text{ that satisfy the resilience condition}$$

where $\widehat{\mathsf{Sys}}$ is the result of applying α to $\mathsf{Sys}(n,t,f)$.

That is, if the answer to the finite-state model checking problem with $\widehat{\mathsf{Sys}}$ and ϕ as input is positive, we can conclude that the answer to the parameterized model checking problem with $\mathsf{Sys}(n,t,f)$ and ϕ as input is positive. However, if $\widehat{\mathsf{Sys}} \not\models \phi$, we cannot directly conclude that $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f}) \not\models \phi$, for every $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition. Due to the abstraction, $\widehat{\mathsf{Sys}}$ has executions that cannot be reproduced in any $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f})$. In this case, the counterexample execution that witnesses $\widehat{\mathsf{Sys}} \not\models \phi$ is called a *spurious* counterexample. *Refinement* of the abstraction [CGJ+00] with a spurious counterexample is not immediate in the parameterized setting, as the counterexample execution in the abstract system may contain transitions that, when concretized, belong to different finite-state concrete instances.

In Chapter 3, we introduce an abstraction technique that allows us to reduce the parameterized model checking problem over the parameterized system $\mathsf{Sys}(n,t,f)$ to the finite-state model checking problem over the abstract system $\widehat{\mathsf{Sys}}$. This abstraction technique is tailored to synchronous system specifications where the process specification is defined by process variables and functions.

Example 1.4. Consider the pseudocode of the algorithm FloodSet, presented in Figure 1.1. Two processes, i and j, that have the same values stored in the set `W`, broadcast the same message in line 5. Similarly, if processes i and j receive the same messages from other processes, they update the values stored in the set `W` to the same value in line 7. Thus, instead of storing the values of the local variables for each of the n processes in the system $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f})$, where n, t, and f are the values assigned to the parameters n, t, and f, we can store whether there exists a process that has a certain value for every local variable. In FloodSet, the processes have finitely many local variables (`v`, `W`, and `dec`) which take values from finite domains. Storing information about the existence of processes that have certain values for the local variables results in an abstract system $\widehat{\mathsf{Sys}}$, whose set of states is finite and independent of the values assigned to the parameters.

This is the main idea of the abstraction technique that we will present in Chapter 3. The abstraction step thus maps a state of the system $\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f})$, for any values n, t, and f that satisfy the resilience condition, to a state of the abstract finite-state system $\widehat{\mathsf{Sys}}$. □

Bounded diameter. Another technique for solving the parameterized model checking problem that we will present in this thesis is based on the concept of *bounded diameter* [BCCZ99].

Let $\mathsf{Sys}(n, t, f)$ be a synchronous system specification representing the infinite family $\{\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \mid \mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N} \text{ satisfy the resilience condition}\}$ of finite-state transition systems $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$. Let $d(\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})) \in \mathbb{N}$ be the *diameter* (i.e., the longest shortest path between any two nodes) of the graph representing the finite-state transition system $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, where the nodes of the graph are the states of $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ and the edges are its transitions. The existence of a bound $d(\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f}))$ on the diameter of the transition system means that every state in $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ can be reached from every other state in $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ by a sequence τ of transitions, whose length $|\tau|$ is less than or equal to $d(\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f}))$, that is, $|\tau| \leq d(\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f}))$. Let D be an upper bound on the diameters $d(\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f}))$ of $\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ for every $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition. We call D the *diameter* of the parameterized system $\mathsf{Sys}(n, t, f)$.

If such a value D exists and can be computed effectively, then for a class of safety properties, which can be verified by showing that a bad state is never reached, the parameterized model checking problem can be reduced to the bounded model checking problem as follows. Given a synchronous specification $\mathsf{Sys}(n, t, f)$, a safety property ϕ, and a diameter D, we have:

$$
\begin{gathered}
\text{for all } \mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N} \text{ that satisfy the resilience condition,} \\
\mathsf{Sys}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \models \phi \quad \text{iff} \\
\text{there do not exist } \mathsf{n}', \mathsf{t}', \mathsf{f}' \in \mathbb{N} \text{ that satisfy the resilience condition and} \\
\text{a sequence } \tau \text{ of transitions in } \mathsf{Sys}(\mathsf{n}', \mathsf{t}', \mathsf{f}') \text{ with } |\tau| \leq D, \\
\text{such that } \tau \text{ reaches a state } s \text{ where } s \not\models \phi
\end{gathered}
$$

Thus, parameterized model checking of a safety property ϕ can be reduced to searching for values $\mathsf{n}', \mathsf{t}', \mathsf{f}' \in \mathbb{N}$ of the parameters, and a sequence τ of transitions in the transition system $\mathsf{Sys}(\mathsf{n}', \mathsf{t}', \mathsf{f}')$ of length $|\tau| \leq D$, such that the property ϕ is violated in the last state reached by τ.

This technique allows us to verify safety for a larger class of algorithms that tolerate crash, send omission, Byzantine, and hybrid faults, which were not verified before, and which we modeled using synchronous threshold automata (introduced in Chapter 4). In Chapter 5, we show undecidability of parameterized reachability for synchronous threshold automata, which implies that we cannot always compute the diameter D. Nevertheless, for a class of algorithms, we show that computing the diameter is decidable. Furthermore, we develop a semi-decision procedure for checking if the diameter D exists. We use this semi-decision procedure to compute the bound on the diameter for algorithms that do not fall in the decidable class. We note that this bound is quite small for all the algorithms we consider, and it does not depend on the parameters. Thus, we can efficiently answer the

parameterized model checking problem for safety properties of synchronous fault-tolerant distributed algorithms using bounded model checking.

Example 1.5. Consider the algorithm FloodSet, whose pseudocode is given in Figure 1.1. An execution of the algorithm for $\mathsf{n} = 5$, $\mathsf{t} = \mathsf{f} = 3$ is given below:

process	initially		i_1 crashed		i_2 crashed		i_3 crashed		no new crashes
i_1	$\mathtt{W} = \{0\}$		crashed		crashed		crashed		crashed
i_2	$\mathtt{W} = \{1\}$		$\mathtt{W} = \{0,1\}$		crashed		crashed		crashed
i_3	$\mathtt{W} = \{1\}$	$\longrightarrow$	$\mathtt{W} = \{1\}$	$\longrightarrow$	$\mathtt{W} = \{0,1\}$	$\longrightarrow$	crashed	$\longrightarrow$	crashed
i_4	$\mathtt{W} = \{1\}$		$\mathtt{W} = \{1\}$		$\mathtt{W} = \{1\}$		$\mathtt{W} = \{0,1\}$		$\mathtt{W} = \{0,1\}$
i_5	$\mathtt{W} = \{1\}$		$\mathtt{W} = \{1\}$		$\mathtt{W} = \{1\}$		$\mathtt{W} = \{1\}$		$\mathtt{W} = \{0,1\}$

In this execution, the processes i_1, i_2, i_3 each crash in a separate round, and manage to send their set $\mathtt{W}$ of values only to the process that crashes in the next round, except in round 3, where i_3 sends only to the correct process i_4. The same final configuration, i.e., when the processes i_1, i_2, i_3 have crashed and the processes i_4, i_5 store both values 0 and 1 in their set $\mathtt{W}$, can be reached from the initial configuration using the following *shorter* execution:

process	initially		i_1, i_2, i_3 crashed		no new crashes
i_1	$\mathtt{W} = \{0\}$		crashed		crashed
i_2	$\mathtt{W} = \{1\}$		crashed		crashed
i_3	$\mathtt{W} = \{1\}$	$\longrightarrow$	crashed	$\longrightarrow$	crashed
i_4	$\mathtt{W} = \{1\}$		$\mathtt{W} = \{0,1\}$		$\mathtt{W} = \{0,1\}$
i_5	$\mathtt{W} = \{1\}$		$\mathtt{W} = \{1\}$		$\mathtt{W} = \{0,1\}$

If we are able to find a number $D \in \mathbb{N}$, such that all possible executions of FloodSet, for any values of the parameters n, t, and f, can be shortened to executions of length at most D, we say that FloodSet has a *bounded diameter*, and we can reduce the parameterized model checking of safety properties to bounded model checking. □

The only drawback of this technique is the fact that we currently can only handle a class of safety properties, which for our benchmarks is enough, as their safety properties fall in this class, and their liveness properties are simple termination properties, which express that they run for a predetermined number of rounds. We leave the study of completeness thresholds for general safety and liveness properties for future work.

1.5.3 Methodology

We now give a brief overview on the general research strategy that we applied in order to tackle the parameterized model checking problem for synchronous fault-tolerant distributed algorithms and obtain the results presented in this thesis.

Our first idea was to develop an abstraction technique that will allow us to construct an abstract finite-state system $\widehat{\mathsf{Sys}}$, which simulates the finite-state systems from the infinite family $\{\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f}) \mid \mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ satisfy the resilience condition$\}$. We surveyed existing works on abstraction for parameterized systems in order to find out which techniques are suitable for synchronous fault-tolerant distributed algorithms. We thus combined existential abstraction [CGL94], as well as counter abstraction [PXZ02] and compositional reasoning [Krs05]. These techniques were not applied to synchronous fault-tolerant distributed algorithms before. We have combined them and adapted them to synchronous systems, obtaining an abstraction technique that allowed us to verify six synchronous fault-tolerant distributed algorithms from the literature, which were not verified before. These results were presented in Chapter 3.

While the abstraction technique gave us first results for parameterized verification of synchronous fault-tolerant distributed algorithms, it has a few limitations. On the one hand, it was designed for a specific class of algorithms, namely those that tolerate crash faults. On the other hand, lot of resources were needed in order to conduct the experimental evaluation, and we were able to conclude that the approach did not scale well. We used TLA+ [Lam02] to formally specify the abstract finite-state system $\widehat{\mathsf{Sys}}$ and used the model checker TLC, one of the tools associated with TLA+, in order to verify the safety and liveness properties of our benchmarks. As TLC is an explicit-state model checker and the transition relation of the obtained abstract system $\widehat{\mathsf{Sys}}$ is highly non-deterministic, the model checker took a long time to enumerate all states (in the order of magnitude of hours, and in some cases, days). This led us to investigate different parameterized model checking techniques, which are applicable to systems that model algorithms that tolerate other fault models as well, while reducing the amount of computing power needed to run experiments, as well as the environmental footprint of the experimental evaluation.

Still, the model checking with TLC uncovered that the abstract system $\widehat{\mathsf{Sys}}$ has a small diameter for all our benchmarks. This motivated us to further investigate whether all the systems in the infinite family $\{\mathsf{Sys}(\mathsf{n},\mathsf{t},\mathsf{f}) \mid \mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ satisfy the resilience condition$\}$ have a small and bounded diameter. Inspired by the bounded diameter approach defined in [KVW14], applied to asynchronous fault-tolerant distributed algorithms modeled using threshold automata, we shifted our focus and tried to adapt the threshold automata framework to the synchronous setting (Chapter 4). We developed a semi-decision procedure that allowed us to compute a bound on the diameter for our benchmarks. Thus, we were able to apply bounded model checking as a complete verification procedure. This allowed us to improve the parameterized verification results for the already verified crash-tolerant benchmarks (which were verified using the abstraction technique), and to obtain new parameterized verification results for 16 additional algorithms, which were not verified before. We present these results in Chapter 5.

To apply these two techniques for parameterized verification, one needs to either: (i) produce an encoding of the abstract system $\widehat{\mathsf{Sys}}$ in some specification language (e.g., TLA+) and give it as input to a model checker (e.g., TLC), or (ii) encode of the process behavior

using a synchronous threshold automaton and give it as input to our semi-decision procedure for computing the diameter, and then, together with the computed diameter, to our bounded model checking procedure for verifying safety properties. Both encoding tasks are non-trivial, as they require background knowledge about: (i) the domain, in order to come up with the correct abstract encoding of the global states and transition relation, and (ii) the interplay between sent and received messages, in order to come up with the correct guard expressions in the synchronous threshold automaton. Hence, the next step in our work was to investigate ways in which we can ease the process of producing an encoding.

To do so, we focused on the synchronous threshold automata framework, since we obtained better parameterized verification results for a larger number of diverse benchmarks. Our goal was to introduce an encoding of the process behavior, which directly matches the pseudocode, is easy to produce manually, and can be automatically verified with our existing techniques. We thus proposed a new variant of synchronous threshold automata, by introducing receive variables, which we present in Chapter 6. The addition of receive variables allowed us to produce a faithful model of a synchronous fault-tolerant distributed algorithm, given its pseudocode, and explicitly encode the relationship between the sent and received messages in the environment specification. We also developed an automatic translation procedure, presented in Chapter 7, which maps a synchronous threshold automaton with receive variables to a synchronous threshold automaton with no receive variables, to which our bounded model checking technique from Chapter 5 is applicable. In this way, we proposed a fully automated method for parameterized verification of synchronous fault-tolerant distributed algorithms, by bridging the gap between the pseudocode description and automated verification procedures.

Last but not least, we were interested in whether the abstraction, bounded model checking, and automatic translation techniques proposed in this thesis are efficient and applicable to the set of benchmarks we are interested in verifying. Therefore, along with every technique we propose, we provide a set of experiments that show its usefulness. From the experimental results, we were able to conclude that the bounded model checking technique from Chapter 5 performs better than the abstraction technique from Chapter 3, and is able to capture more diverse benchmarks. Further, the automatic translation technique from Chapter 7 revealed some inconsistencies between some of the synchronous threshold automata produced as output of the translation, and the synchronous threshold automata we encoded manually in Chapter 5, which, by manual inspection, we classified as glitches in the manual encodings. This confirmed our hypothesis that manually producing synchronous threshold automata encodings is error-prone, and justified the need for introducing the receive variable variant of synchronous threshold automata.

1.6 Benchmarks

In this section, we introduce the synchronous fault-tolerant distributed algorithms that we will use as benchmarks in our experimental evaluations. We start by introducing the

Table 1.1: A list of the benchmarks that we will use in the experimental evaluation of the results presented in this thesis.

benchmark	problem	reference	fault model
EDAC	consensus	[CS04, Fig. 1]	crash
ESC	consensus	[Ray10, Fig. 3.3]	crash
FairCons	consensus	[Ray10, Fig. 2.2]	crash
FloodSet	consensus	[Lyn96, p. 103]	crash
PhaseKing	consensus	[BGP89, Fig. 4]	Byzantine
HybridKing	consensus	[BSW11, Fig. 2]	hybrid
ByzKing	consensus	[BSW11, Fig. 2]	Byzantine
OmitKing	consensus	[BSW11, Fig. 2]	send omission
PhaseQueen	consensus	[BGP, Fig. 1]	Byzantine
HybridQueen	consensus	[BSW11, Fig. 1]	hybrid
ByzQueen	consensus	[BSW11, Fig. 1]	Byzantine
OmitQueen	consensus	[BSW11, Fig. 1]	send omission
FloodMin	k-set agreement	[Lyn96, p. 163]	crash
FloodMinOmit	k-set agreement	[Lyn96, p. 163]	send omission
kSetOmit	k-set agreement	[Ray10, Fig. 7.1]	send omission
NBAC	non-blocking atomic commit	[Ray10, Fig. 6.1]	crash
SAB	authenticated broadcast	[ST87, Fig. 2]	Byzantine
HybridSAB	authenticated broadcast	[BSW11, Fig. 4]	hybrid
OmitSAB	authenticated broadcast	[BSW11, Fig. 4]	send omission

problems that these algorithms solve, namely, *consensus*, *k-set agreement*, *non-blocking atomic commit*, and *authenticated broadcast*. We then give a brief description of the algorithms we consider. An overview of all benchmarks, the problems they solve, and the faults they tolerate is given in Table 1.1.

1.6.1 Consensus

Consensus is a central problem in the area of distributed computing. In consensus, the goal of the processes is to reach an agreement on a single value, which has been proposed by some process. To solve the consensus problem, the processes in a distributed system start by proposing their initial values, and coordinate in order to reach an agreement. Once a process makes a decision, it cannot change it, that is, its decision is irrevocable.

We will consider consensus algorithms that tolerate crash, send omission, Byzantine, and hybrid faults. Depending on the fault model, there are subtle differences in the correctness conditions of the consensus algorithm. For example, we point the reader to the classical textbook on distributed algorithms, [Lyn96], where different definitions

of the properties are stated for the crash and Byzantine models. Here, as an example, we present the properties that a consensus algorithm that tolerates crash faults has to satisfy:

- *Validity.* A value that is not an initial value of any process is not a value that is decided on.
- *Agreement.* No two correct processes decide on different values.
- *Termination.* Every correct process eventually decides.

We now present the consensus algorithms from the distributed algorithms literature that we will use as benchmarks in the experimental evaluation in Chapters 3 and 5.

FloodSet – a consensus algorithm from [Lyn96, p. 103], whose pseudocode is presented in Figure 1.1, and whose features are described in Example 1.1. The algorithm is designed to tolerate crash faults, under the resilience condition $n > t \geq f$.

FairCons – a consensus algorithms from [Ray10, Fig. 2.2] that tolerates crash faults under the resilience condition $n > t \geq f$. It has a similar structure to FloodSet; the differences are that the processes maintain a single (best) value they have seen so far, in contrast to FloodSet, where the processes maintain a set of values. The best value is updated as the minimum of all received values. A process broadcasts its best value only in rounds where the best value differs from the best value of the previous round.

EDAC – an early deciding consensus algorithm [CS04, Fig. 1], that tolerates crash faults under the resilience condition $n > t+1 \wedge t \geq f$. In EDAC, every process keeps track of the messages received in the previous round. A process can decide before round $t+1$ if the messages received in the previous round do not differ from the messages received in the current round. This means that, unlike FloodSet and FairCons, that terminate within $t+1$ rounds, EDAC terminates within $min(f+1, t+1)$ rounds.

ESC – an early stopping consensus algorithm from [Ray10, Fig. 3.3], that tolerates crash faults under the resilience condition $n > t \geq f$. Similarly to EDAC, in ESC processes compare the number of messages received in the current and previous round, and stop their execution within $min(f+2, t+1)$ rounds.

PhaseKing – a consensus algorithm from [BGP, Fig. 2], [BGP89, Fig. 4] that tolerates Byzantine faults under the resilience condition $n > 3t \wedge t \geq f$. In PhaseKing, the computations are organized in phases, with three rounds per phase. In each phase, a dedicated process, i.e., a *king*, acts as a coordinator. That is, one round in every phase is reserved for the king's broadcast. The processes update their local variables based both on the messages received from other processes in the other two rounds, as well as the message received from the king in the king's broadcast round. The algorithm runs for $t+1$ phases.

PhaseQueen – a consensus algorithm from [BGP, Fig. 1] that tolerates Byzantine faults under the resilience condition $n > 4t \wedge t \geq f$. Similarly to PhaseKing, in PhaseQueen the computation is organized in phases, with a designated process, a *queen*, acting as a coordinator in every phase. There are two rounds per phase in PhaseQueen: the first round is the universal message exchange, and the second is reserved for the queen's broadcast.

HybridKing – a consensus algorithm from [BSW11, Fig. 2], inspired by PhaseKing, and designed to tolerate hybrid faults, i.e., designed to tolerate multiple kinds of faults simultaneously. We focus on the case where HybridKing tolerates both Byzantine and send omission faults. In this case, we have the resilience condition $n > 3t_b + 2t_o \wedge t_b \geq f_b \wedge t_o \geq f_o$, where t_b, f_b (t_o, f_o) are the maximal and actual number of Byzantine (send omission) faults, respectively.

ByzKing – a variant of HybridKing, where no send omission faults occur, i.e., where t_o is set to 0. Thus, ByzKing tolerates Byzantine faults, under the resilience condition $n > 3t_b \wedge t_b \geq f_b$.

OmitKing – a variant of HybridKing, where no Byzantine faults occur, i.e., where t_b is set to 0. Thus, OmitKing tolerates send omission faults, under the resilience condition $n > 2t_o \wedge t_o \geq f_o$.

HybridQueen – a consensus algorithm from [BSW11, Fig. 1], inspired by PhaseQueen, and designed to tolerate hybrid faults. We consider the hybrid variant of this algorithm where we have send omission faults and Byzantine faults, under the resilience condition $n > 4t_b + 2t_o \wedge t_b \geq f_b \wedge t_o \geq f_o$, where t_b, f_b, t_o, f_o are defined as for HybridKing.

ByzQueen – a variant of HybridQueen, where no send omission faults occur, i.e., where t_o is set to 0. Thus, ByzQueen tolerates Byzantine faults, under the resilience condition $n > 4t_b \wedge t_b \geq f_b$.

OmitQueen – a variant of HybridQueen, where no Byzantine faults occur, i.e., where t_b is set to 0. Thus, OmitQueen tolerates send omission faults, under the resilience condition $n > 2t_o \wedge t_o \geq f_o$.

We applied the techniques presented in Chapter 3, to verify the consensus algorithms FloodSet, FairCons, EDAC, and ESC. In Chapter 5, we used all consensus algorithms as benchmarks, except EDAC and ESC.

1.6.2 *k*-set Agreement

The k-set agreement problem is a relaxation of the consensus problem, in the sense that the processes coordinate to reach an agreement on up to k distinct values. Observe that by setting k to 1, we obtain consensus.

We will consider algorithms that solve the k-set agreement problem under crash and send omission faults. The following properties need to be satisfied by an algorithm that solves k-set agreement:

- *Validity.* A value that is not an initial value of any process is not a value that is decided on.
- *k-Agreement.* The set of values decided by the correct processes has cardinality at most k.
- *Termination.* Every correct process eventually decides.

We assume that the processes running an algorithm that solves the k-set agreement problem have initial values from a set V of values, where $|V| \geq k$. For our benchmarks, we look at the following k-set agreement algorithms where we set k either to 1 or 2.

FloodMin – a k-set agreement algorithm from [Lyn96, p. 163], that tolerates crash faults under the resilience condition $n > t \geq f$. The processes start with a value from the set V of values and eventually decide on a value from V, such that not more than k different values are decided.

FloodMinOmit – a variant of FloodMin that tolerates send omission faults under the resilience condition $n > t \geq f$.

kSetOmit – a k-set agreement algorithm from [Ray10, Fig. 7.1], that tolerates send omission faults under the resilience condition $n > t \geq f$, similar to FloodMin, where in each round at most k processes broadcast their values.

In Chapter 3, we analyzed FloodMin for $k = 1$ and $k = 2$. In Chapter 5, we analyzed FloodMin, FloodMinOmit, and kSetOmit, for $k = 1$ and $k = 2$.

1.6.3 Non-Blocking Atomic Commit

The non-blocking atomic commit problem is a distributed agreement problem which originated in distributed databases. The goal of the processes is to decide whether to *commit* or *abort* a transaction, that is, the processes need to agree on one of the two values – *commit* or *abort*. If all processes propose *commit*, they all decide *commit*. However, if at least one process proposed *abort*, all processes decide *abort*.

The following properties should be satisfied by an algorithm that solves the non-blocking atomic commit problem:

- *Justification.* If a process decides *commit*, then all processes proposed *commit*.

- *Obligation.* If all processes propose *commit*, and there are no failures, then *commit* is the only possible decision value.
- *Agreement.* No two correct processes decide on different values.
- *Termination.* Every correct process eventually decides.

In our experiments, we used the following algorithm as a benchmark that solves the non-blocking atomic commit problem:

NBAC – a non-blocking atomic commit algorithm from [Ray10, Fig. 6.1], that tolerates crash faults under the resilience condition $n > t \geq f$. This algorithm runs for $t + 2$ rounds: in the first round, the processes exchange their proposals, either *commit* or *abort*. If a process receives n messages, and if no *abort* message is within them, then the process initializes a local variable v to 1. Otherwise, the variable v is initialized to 0. After the first round, the processes run a consensus algorithm for $t + 1$ rounds, where they decide on whether to *commit* (if the output of consensus is 1) or *abort* (if the output of consensus is 0), where the variable v holds the initial value for each process in the consensus instance. In our experiments, we use FloodSet as an underlying consensus algorithm.

We verified NBAC using the techniques presented in Chapter 3.

1.6.4 Authenticated Broadcast

Many algorithms that solve agreement problems, such as consensus, k-set agreement, or non-blocking atomic commit rely on broadcasts in the message exchange. To restrict the effect the faulty processes have on the system, many algorithm designers require that the messages are authenticated. One way to authenticate messages is by using digital signatures. That is, a process $i \in P$ sends a signed message to all other processes, and a process $j \in P$, with $i \neq j$, accepts i's message, if it can verify its signature. Digital signatures are an expensive method of authentication, due to the computational overhead that they introduce.

Authenticated broadcast algorithms simulate digital signatures and satisfy the following properties:

- *Unforgeability.* If process i is correct, and it does not broadcast a message in round r, then no other process ever accepts a round r message by process i.
- *Correctness.* If a correct process i broadcasts a message in round r, then every correct process accepts a round r message by process i in round r.
- *Relay.* If a correct process accepts some round r message by process i in round r', then every other correct process accepts the round r message by process i in round $r' + 1$ or earlier.

The following broadcast algorithms were designed to simulate authenticated broadcasts, without using digital signatures.

SAB – a simulated authenticated broadcast primitive from [ST87, Fig. 2], that tolerates Byzantine faults under the resilience condition $n > 3t \wedge t \geq f$. This algorithm is designed to satisfy the three properties listed above without the use of digital signatures. To achieve this, it is required that several processes act as witnesses of a broadcast by a correct process. A correct process accepts a message that was broadcast by another process if there are enough witnesses for this message.

HybridSAB – a simulated authenticated broadcast primitive from [BSW11, Fig. 4], inspired by SAB, and designed to tolerate hybrid faults. That is, we consider the case when HybridSAB tolerates send omission and Byzantine faults, under the resilience condition $n > 3t_b + 2t_o \wedge t_b \geq f_b \wedge t_o \geq f_o$.

OmitSAB – a variant of HybridSAB, where no Byzantine faults occur, i.e., where $t_b = 0$. The algorithm OmitSAB tolerates send omission faults under the resilience condition $n > 2t_o \wedge t_o \geq f_o$.

We verified these three algorithms using the techniques presented in Chapter 5.

1.7 Contributions and Roadmap

The goal of this thesis is to find suitable formal models for encoding synchronous fault-tolerant distributed algorithms and to introduce techniques that reduce parameterized model checking of synchronous fault-tolerant distributed algorithms to finite-state model checking or bounded model checking. In this thesis:

- we propose a modeling approach for synchronous fault-tolerant distributed algorithms, consisting of process and environment specifications;
- we instantiate this modeling approach and propose three different process and environment specifications:

  (S1) process and environment variables and functions,

  (S2) synchronous threshold automata, and

  (S3) synchronous threshold automata with receive variables.

  These three formal models allow us to encode synchronous fault-tolerant distributed algorithms from the literature and apply efficient verification techniques;
- we present a sound abstraction-based technique for solving the parameterized model checking problem for algorithms whose process behavior is modeled using process variables and functions (S1);

- we propose a sound and complete bounded model checking technique for verifying safety properties of synchronous fault-tolerant distributed algorithms, whose process specification is given as a synchronous threshold automaton (S2);
- we introduce a sound and complete quantifier-elimination based procedure that translates synchronous threshold automata with receive variables (S3) to synchronous threshold automata with no receive variables (S2) (the latter being a valid input to the bounded model checking technique);
- we show the effectiveness of the proposed methods by running experimental evaluations on various synchronous fault-tolerant benchmarks, that tolerate different kinds of faults, listed in Table 1.1;
- using the abstraction-based technique, we automatically verified six different synchronous fault-tolerant distributed algorithms for the first time. These algorithms tolerate crash faults and were modeled using (S1);
- using the bounded model checking technique, we improved the verification times for the crash-tolerant benchmarks, previously verified using the abstraction technique, and automatically verified 13 additional synchronous fault-tolerant distributed algorithms for the first time. The algorithms we verified using bounded model checking tolerate crash, send-omission, Byzantine, and hybrid faults, and were modeled using (S2);
- we implemented an automatic verification procedure, which given a synchronous threshold automaton with receive variables (S3), representing a formal model of a synchronous fault-tolerant distributed algorithm close to its pseudocode description:
  1. automatically generates a synchronous threshold automaton with no receive variables (S2) by applying the quantifier-elimination based translation procedure,
  2. automatically verifies the safety properties of the underlying algorithm using the bounded model checking technique, applied to the synchronous threshold automaton (S2) obtained as output of the translation in step 1.

The remainder of this thesis is organized as follows. In Chapter 2 we propose a formal model for synchronous fault-tolerant distributed algorithms, where the process and environment specification is given using process and environment variables and functions (S1). Then, in Chapter 3, we present an abstraction technique for the synchronous system specification obtained in Chapter 2, by composing n process specifications and an environment specification. We model and automatically verify six synchronous algorithms, and present our experimental results. These two chapters are based on our work published in [ARS+18].

Chapter 4 introduces synchronous threshold automata (S2), as a way for producing process and environment specifications which are better suited for efficient applications of

formal verification techniques. The bounded model checking technique for parameterized model checking of safety properties, that we developed for systems of n synchronous threshold automata, is presented in Chapter 5. In this chapter, we also give a proof of undecidability of the parameterized reachability problem, and report on results of applying the bounded model checking technique to 16 algorithms, modeled using synchronous threshold automata. Chapters 4 and 5 are an extension of our results published in [SKWZ19].

In Chapter 6, we introduce synchronous threshold automata with receive variables (S3) – a process specification that bridges the gap between the pseudocode description of the process behavior and the synchronous threshold automaton used for verification. Chapter 7 proposes a translation procedure, based on quantifier elimination, which maps a synchronous threshold automaton with receive variables to a synchronous threshold automaton, as defined in Chapter 4. By applying this translation, we can reduce parameterized model checking of safety properties for systems of synchronous threshold automata with receive variables to bounded model checking of safety properties for systems of synchronous threshold automata with no receive variables. We originally proposed the introduction of receive variables in threshold automata and the translation procedure for the *asynchronous* case in [SKWZ20]. As asynchronous systems are out of the scope of this thesis, in Chapters 6 and 7, we adapt these notions to the synchronous case. The results presented in Chapters 6 and 7 were later published in [SKWZ21].

Finally, Chapter 8 gives a summary of the results presented in this thesis and proposes directions for future work.

CHAPTER 2

Process Variables and Functions

In this chapter, we propose a process and environment specification for a class of synchronous fault-tolerant distributed algorithms. More precisely, we will focus on consensus, k-set agreement, and non-blocking atomic commit algorithms that tolerate crash faults. As we saw in Section 1.6, these are algorithms where the processes exchange messages in order to reach a decision, even when some of the processes may fail by crashing. Adhering to the modelling approach introduced in Section 1.3.1, we model the synchronous fault-tolerant distributed algorithm using a synchronous system specification that is composed of n copies of a process specification and an environment specification. The system specification obtained in such a way is *parameterized* in the parameters n, t, and f, where n denotes the number of processes, f is the number of faults, and t is the upper bound on the number of faults.

To model a process, we define a process specification $\mathsf{proc}(n, t, f)$ consisting of *process variables* and *process functions*. The process variables either store values from a finite domain, or are one-dimensional arrays of size n that store information about the other processes in the system, such as, e.g., the messages received from the other processes in the previous round. The process functions define the way in which the values of the process variables get updated. We define the process variables and functions in Section 2.1.

To describe the updates of the finite domain process variables, we define a language of *guarded assignments*. This language is powerful enough to capture constructs that typically occur in synchronous distributed algorithms, such as conditional constructs and iteration over process identifiers. For instance, a process i can check whether there is a process j from which a message was received in the current and the previous round. This construct is used in *early deciding/stopping* consensus algorithms, such as EDAC and ESC. Guards that compare the round number against a parameter, which we call *termination guards*, are typically used in synchronous agreement algorithms to check

whether a certain round is reached, that is, whether it is safe for a correct process to make a decision. The guarded assignments are formalized in Section 2.1.3.

To model the environment, we introduce an environment specification $\mathsf{env}(n, t, f)$ that consists of *environment variables*. The proposed environment specification is tailored to model crash faults, that is, faults exhibited by processes that stop working and cannot restart. As a process can crash in the middle of its execution, it can be the case that it sends a message only to a subset of processes. The environment variables keep track of the round number, the crashed processes, and for each crashed process, the subset of processes that receive a message from it in the round in which it crashes. We discuss the environment specification in more detail in Section 2.2.

We define the synchronous system specification $\mathsf{STS}(n, t, f)$ obtained by composing n copies of the process specification $\mathsf{proc}(n, t, f)$ and an environment specification $\mathsf{env}(n, t, f)$. As $\mathsf{proc}(n, t, f)$ and $\mathsf{env}(n, t, f)$ define process and environment variables, respectively, the system specification $\mathsf{STS}(n, t, f)$ defines *system variables*, which combine both the process and environment variables. The system specification is tailored to model agreement algorithms that tolerate crash faults, which means that its *resilience condition* is expressed by the inequality $n > t \geq f$. The synchronous system specification $\mathsf{STS}(n, t, f)$ will be used to represent the infinite family $\{\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \mid \mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ satisfy the resilience condition$\}$ of finite-state *synchronous transition systems*. We define the synchronous system specification and transition system in Sections 2.3 and 2.4, respectively.

Finally, in Section 2.5, we will define the logic that we will use to formalize the properties of an algorithm encoded using the defined system specification.

2.1 Process Specification: Process Variables and Functions

In this section, we propose a process specification $\mathsf{proc}(n, t, f)$, defined by process variables and process functions.

2.1.1 Process Variables

We start by introducing the process variables and their domains. We then introduce two special variables, used to store: (1) whether a process has failed and (2) the messages received by other processes.

Definition 2.1 (Process variables)**.** Let V_{proc} be a finite set of *process variables*, which is partitioned into:

- the set $\mathsf{cntl}(V_{\mathsf{proc}}) = \{x_{cv} \mid 1 \leq cv \leq |\mathsf{cntl}(V_{\mathsf{proc}})|\}$ of *process control variables*,
- the set $\mathsf{nbhd}(V_{\mathsf{proc}}) = \{\mathbf{y}_{nv} \mid 1 \leq nv \leq |\mathsf{nbhd}(V_{\mathsf{proc}})|\}$ of *process neighborhood variables*. □

A process uses its control variables to store information local to itself, and its neighborhood variables to store information about other processes. For example, a process can use a control variable to store a value it has decided on. It can use a neighborhood variable to store the values that the other processes sent to it in the current round.

Definition 2.2 (Values of process variables)**.** For a process variable $z \in V_{\mathsf{proc}}$, let D_z denote the finite set of *values* associated with z. The process variable $z \in V_{\mathsf{proc}}$ ranges over the set:

- D_z, if z is a process control variable, that is, if $z \in \mathsf{cntl}(V_{\mathsf{proc}})$,
- D_z^n, if z is a process neighborhood variable, that is, if $z \in \mathsf{nbhd}(V_{\mathsf{proc}})$.

Every neighborhood variable $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}})$, the set $D_{\mathbf{y}}$ of values contains a special *null value*, denoted by $\perp$. □

The process control variables store a single value, while the process neighborhood variables are one-dimensional arrays of size n, where n is the parameter denoting the number of processes in the system. Thus, the size of each $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}})$ is parameterized by the number n of processes, and each array cell of the one-dimensional array variable $\mathbf{y}$ takes a value from the set $D_{\mathbf{y}}$. The null value $\perp$ in the set $D_{\mathbf{y}}$ of values, for $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}})$, is used to represent an empty cell in the one-dimensional array $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}})$.

Definition 2.3 (Special process variables)**.** The set V_{proc} of process variables contains the following two special process variables:

- the failure flag $\mathit{fld} \in \mathsf{cntl}(V_{\mathsf{proc}})$, which ranges over $D_{\mathit{fld}} = \{\perp, \top\}$, and used to store whether the process has failed or not,
- the message array $\mathbf{msg} \in \mathsf{nbhd}(V_{\mathsf{proc}})$, which ranges over $D^n_{\mathbf{msg}}$, and is used to store the messages the process receives in the current round.

For convenience, we write $\mathcal{M}$ instead of $D_{\mathbf{msg}}$ to denote the set of values stored in the cells of the message array $\mathbf{msg}$, and call it the set of *message types*. □

A local state of a process is defined by a valuation of the variables $z \in V_{\mathsf{proc}}$, where they are assigned values from their respective sets D_z of values.

Definition 2.4 (Local states.)**.** The set $L(n) = \prod_x D_x \times \prod_{\mathbf{y}} D^n_{\mathbf{y}}$, of *local states*, for $x \in \mathsf{cntl}(V_{\mathsf{proc}})$ and $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}})$, contains valuations of the process variables V_{proc}. □

As the size of the process neighborhood variables $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}})$ is parameterized by the number n of processes, the set $L(n)$ of local states is also parameterized by the number n of processes. By assigning a value $\mathsf{n} \in \mathbb{N}$ to the parameter n, we obtain a finite set $L(\mathsf{n})$,

```
1  best := input(V)
2  dec := ⊥
3  for each round 1 through ⌊t/k⌋ + 1 do {
4    broadcast best
5    receive values b_1, ... b_ℓ from others
6    best := min {b_1, ... b_ℓ}
7  }
8  dec := best
```

Figure 2.1: The pseudocode of FloodMin

where the process neighborhood variables $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}})$ are one-dimensional arrays of size n. Thus, the set $L(n)$ represents an infinite family $\{L(\mathsf{n}) \mid \mathsf{n} \in \mathbb{N}\}$ of finite sets of process local states for different system sizes.

An assignment of values from the sets D_x of values to the control variables $x \in \mathsf{cntl}(V_{\mathsf{proc}})$ defines a control state of a process.

Definition 2.5 (Control states)**.** The set $C = \prod_x D_x$, for $x \in \mathsf{cntl}(V_{\mathsf{proc}})$, of *control states* contains valuations of the process control variables $\mathsf{cntl}(V_{\mathsf{proc}})$. □

The set C of control states is a finite set, as there are finitely many process control variables that take values from finite sets of values. Observe that each local state $\mathit{local} \in L(\mathsf{n})$, for $\mathsf{n} \in \mathbb{N}$, contains a control state. We will denote by $\mathit{local.control} \in C$ the control state associated with the local state $\mathit{local} \in L(\mathsf{n})$. Moreover, for a local state $\mathit{local} \in L(\mathsf{n})$ and a process variables $z \in V_{\mathsf{proc}}$, we will denote by $\mathit{local}.z$ the value that the local state *local* assigns to the variable z.

Definition 2.6 (Initial control states)**.** The set $C_0 \subseteq C$ of *initial control states* contains valuations of the process control variables $\mathsf{cntl}(V_{\mathsf{proc}})$, where every process control variable $x \in \mathsf{cntl}(V_{\mathsf{proc}})$ is assigned an initial value from D_x. The initial value of the special control variable *fld* is $\perp$. □

By setting the initial value of the special control variable *fld* to $\perp$, we assume that initially every process that participates in the algorithm is correct. That is, every process can fail by crashing only during the execution algorithm, but not before. In general, for certain analyses, it might be beneficial to distinguish the scenarios where processes have initially failed from the scenarios where all processes are initially correct. For the verification questions of interest in this thesis, it suffices consider only the latter scenarios, which are captured by Definition 2.6.

Example 2.1. The pseudocode of the algorithm FloodMin is presented in Figure 2.1. For simplicity of presentation, we consider the case when $k = 1$. As discussed in Section 1.6.2, by setting $k = 1$, the algorithm FloodMin solves the consensus problem. Each process running FloodMin has a variable `best`, which stores its input value from the set `V` (line 1),

and a variable `dec`, which stores the value the process decides on. The variable `best` is updated in each round as the minimum of all received values (line 6). FloodMin runs for $\lfloor t/k \rfloor + 1$ rounds (line 3), which in case of $k = 1$ amounts to $t + 1$ rounds. The variable `dec` is assigned the value of the variable `best` (line 8) after the loop on line 3 terminates. The set of message types is the set $\mathcal{M} = \mathtt{V} \cup \{\perp\}$, as each process broadcasts its value `best` in line 4, and we assume that the set $\mathcal{M}$ of message types contains the special null value $\perp$. For the case when $k = 1$, we assume that $\mathtt{V} = \{0, 1\}$.

To model the process behavior, we define the following process variables for processes running FloodMin:

- $best \in \mathsf{cntl}(V_{\mathsf{proc}})$, a control variable ranging over the set $D_{best} = \{0, 1\}$ of values, used to store the value of the variable `best`,
- $dec \in \mathsf{cntl}(V_{\mathsf{proc}})$, a control variable ranging over the set $D_{dec} = \{0, 1, \perp\}$, used to store the value of the variable `dec`,
- $\mathit{fld} \in \mathsf{cntl}(V_{\mathsf{proc}})$, the failure flag, ranging over the set $D_{\mathit{fld}} = \{\perp, \top\}$, and
- $\mathbf{msg} \in \mathsf{nbhd}(V_{\mathsf{proc}})$, the message array, ranging over $\mathcal{M}^n$.

Initially, *best* is one of the values from the set $\{0, 1\}$ (line 1). The initial value of *dec* is $\perp$, denoting that the process has not decided yet (line 2). From Definition 2.6, we have that the initial value of *fld* is $\perp$. Hence, the set C of control states is $C = D_{best} \times D_{dec} \times D_{\mathit{fld}}$, and the set $C_0 \subseteq C$ is $C_0 = D_{best} \times \{\perp\} \times \{\perp\}$.

Suppose $\mathsf{n} = 6$. An example local state from the set $L(\mathsf{n}) = C \times \mathcal{M}^{\mathsf{n}}$ is given below:

	best	*dec*	*fld*	**msg**
$\mathit{local} \in L(\mathsf{n})$:	1	$\perp$	$\perp$	$[1, 1, 0, 1, 1, \perp]$

When a process i, with $1 \leq i \leq \mathsf{n} = 6$, is in the local state *local* given above, it has the value *best* set to 1, it neither has decided, nor failed, and it has received a message with value $0 \in \mathcal{M}$ from process 2, and a message with value $1 \in \mathcal{M}$ from all other processes, except from process 6, from which it does not receive a message, as $\mathit{local}.\mathbf{msg}[6] = \perp$. □

2.1.2 Process Functions

We are now ready to define the process functions that define the value of the message that each process sends in a round, and the way in which it updates its local state based on the messages received in the current round.

Definition 2.7 (Process functions)**.** Let F_{proc} be a finite set of *process functions* that contains the following functions:

- the *message generation* function $\mathit{send\_msg} : C \to \mathcal{M}$,

- the *message translation* function $translate_{\mathbf{y}} : \mathcal{M} \to D_{\mathbf{y}}$, for each $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}}) \setminus \{\mathbf{msg}\}$, such that $translate_{\mathbf{y}}(\bot) = \bot$,
- the *control state update* function $update_{n,t,r} : L(n) \to C$, which is parameterized by the parameters n and t, and the round number r. □

We use process functions to formally encode the process behavior. The message generation function *send_msg* maps process control states to the set of message types. It is used by a process i to compute the type of message it will send, based on its current control state.

Example 2.2. Consider the algorithm FloodMin, for $k = 1$, whose pseudocode is given in Figure 2.1. The process function *send_msg* maps control states to message types as follows. As in line 4 of the pseudocode, each process broadcasts its value *best*, for every $control \in C$, we have $send\_msg(control) = control.best \in \mathcal{M}$. □

The message translation function $translate_{\mathbf{y}}$, for $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}}) \setminus \{\mathbf{msg}\}$, translates a message type m from the set $\mathcal{M}$ of message types to a value from the set $D_{\mathbf{y}}$. It is used by a process i to map a message that process i received from process j to a value that process i stores in its neighborhood variable $\mathbf{y}$ for the process j. That is, using $translate_{\mathbf{y}}$, process i translates the received messages into some information about the other processes in its local state. Both the message generation and message translation functions are fixed and finite.

Example 2.3. Consider the pseudocode of the algorithm EDAC, given in Figure 2.2, which describes an early deciding consensus algorithm. Each process stores the set of values it has seen so far in the variable `W` and the decision value in the variable `dec`. Additionally, each process stores a Boolean flag `halt`, denoting if the process i has stopped running the algorithm because it has decided, and two sets `prev_r` and `curr_r` of processes, storing the processes from which process i received messages in the previous and current round, respectively.

To model the process behavior, we define control process variables $W, dec, halt, fld \in \mathsf{cntl}(V_{\mathsf{proc}})$ and a neighborhood variable $\mathbf{msg} \in \mathsf{nbhd}(V_{\mathsf{proc}})$ analogous to the way we defined process control and neighborhood variables for the algorithm FloodMin in Example 2.1. To capture the sets of processes from which process i receives messages, we proceed as follows. We introduce a neighborhood variable $\mathbf{prev\_r} \in \mathsf{nbhd}(V_{\mathsf{proc}})$, ranging over $D^n_{\mathbf{prev\_r}} = \{\bot, \top\}^n$, such that each $\mathbf{prev\_r}[j]$ is a Boolean flag that stores if a process received a message from process j in the previous round. To update this variable, we introduce the following translate function $translate_{\mathbf{prev\_r}} : \mathcal{M} \to D_{\mathbf{prev\_r}}$:

$$translate_{\mathbf{prev\_r}}(m) = \begin{cases} \top & \text{if } m \neq \bot \\ \bot & \text{otherwise} \end{cases}, \text{ where } m \in \mathcal{M} \text{ and } D_{\mathbf{prev\_r}} = \{\bot, \top\}$$

Observe that we do not need to introduce a neighborhood variable for the variable `curr_r` from the pseudocode, which stores the set of processes from which process i

```
1  v := input({0, 1})
2  W := {v}
3  halt := ⊥
4  dec := ⊥
5  prev_r := ∅
6  curr_r := ∅
7  while ¬halt do {
8    if dec = ⊥ then broadcast W
9    else broadcast (D, dec)
10   receive values X_j from all j ∈ P for which a message arrives
11   if dec ≠ ⊥ then halt = ⊤
12   if some message (D, v) arrives then halt = v
13   else
14     W := W ∪ ⋃_j X_j
15     prev_r := curr_r
16     curr_r := {j | a message from j arrived in this round}
17     if prev_r = curr_r then dec := min(W)
18 }
```

Figure 2.2: The pseudocode of the algorithm EDAC [CS04]

received a message in the current round. Instead, we can model the check whether the process received messages from the same set of processes in two consecutive rounds from line 17 by checking if $\mathbf{prev\_r}[j] = \top$ and $\mathbf{msg}[j] \neq \bot$, for every process j. □

2.1.3 Guarded Assignments

The control state update function $update_{n,t,r}$ is parameterized by the parameters n and t, and the round number r. By assigning values $\mathsf{n}, \mathsf{t}, \mathsf{r} \in \mathbb{N}$ to n, t, r, respectively, such that $\mathsf{t} < \mathsf{n}$, we obtain a finite function $update_{\mathsf{n},\mathsf{t},\mathsf{r}} : L(\mathsf{n}) \to C$, that a process in a system of size n uses to update its control state based on its current local state. Moreover, the function $update_{n,t,r}$ represents the infinite family $\{update_{\mathsf{n},\mathsf{t},\mathsf{r}} : L(\mathsf{n}) \to C \mid \mathsf{n}, \mathsf{t}, \mathsf{r} \in \mathbb{N} \text{ and } \mathsf{t} < \mathsf{n}\}$ of finite functions. To characterize this infinite family, in the following we introduce a language of guarded assignments.

We first define the syntax and semantics of guard propositions.

Definition 2.8 (Syntax of guard propositions)**.** We define the syntax of the *empty, control, neighborhood, and termination guard propositions* as follows:

empty	$\top$	
control	$x = \mathsf{v}$	where $x \in \mathsf{cntl}(V_{\mathsf{proc}})$ and $\mathsf{v} \in D_x$
neighborhood	$\exists j \ \bigwedge_{\psi \in \Psi(j)} \psi$	where $\Psi(j) \subseteq \{\mathbf{y}[j] = \mathsf{v} \mid \mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}}), \mathsf{v} \in D_{\mathbf{y}}\} \cup \{\mathbf{y}[j] \neq \mathsf{v} \mid \mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}}), \mathsf{v} \in D_{\mathbf{y}}\}$
termination	$r \geq \phi(n, t)$	where r is the round number and $\phi(n, t)$ is a linear arithmetic expression over the parameters. □

The guard propositions capture various constructs found in the distributed computing literature. For example, Boolean combinations of control guard propositions are used to check whether a process is in a certain control state. The neighborhood guard propositions are used to check the values that a process stores for the other processes in the system, such as, e.g., if a certain message has been received by at least one process. Termination guard propositions are used to capture when a process is ready to terminate, that is, when it has run the algorithm for some number of rounds that is determined by an arithmetic expression $\phi(n,t)$ over the parameters n and t.

To give a semantics to the guard propositions, we need to pick values for the parameters n, t, and the round number r, as well as a local state of a process. The formal semantics of the guard propositions is given below.

Definition 2.9 (Semantics of guard propositions)**.** The guard propositions are evaluated over tuples $(\mathit{local}, \mathsf{n}, \mathsf{t}, \mathsf{r})$, where $\mathsf{n}, \mathsf{t}, \mathsf{r} \in \mathbb{N}$ and $\mathsf{t} < \mathsf{n}$, are values assigned to n, t, r, respectively, and $\mathit{local} \in L(\mathsf{n})$ as follows:

$(\mathit{local}, \mathsf{n}, \mathsf{t}, \mathsf{r}) \models \top$	holds true
$(\mathit{local}, \mathsf{n}, \mathsf{t}, \mathsf{r}) \models x = \mathsf{v}$	if $\mathit{local}.x = \mathsf{v}$
$(\mathit{local}, \mathsf{n}, \mathsf{t}, \mathsf{r}) \models \exists j \ \bigwedge_{\psi \in \Psi(j)} \psi$	if there is j with $1 \leq j \leq \mathsf{n}$ such that for every $\psi \in \Psi(j)$ we have either $\mathit{local}.\mathbf{y}[j] = \mathsf{v}$, if $\psi \equiv \mathbf{y}[j] = \mathsf{v}$, or $\mathit{local}.\mathbf{y}[j] \neq \mathsf{v}$, if $\psi \equiv \mathbf{y}[j] \neq \mathsf{v}$
$(\mathit{local}, \mathsf{n}, \mathsf{t}, \mathsf{r}) \models r \geq \phi(n,t)$	if $\mathsf{r} \geq \phi(\mathsf{n}, \mathsf{t})$ □

We are now ready to define the guarded assignments.

Definition 2.10 (Guarded Assignment)**.** A *guarded assignment* is an expression of the form $\varphi \to \mathit{assig}$, where:

- φ is a *guard*, i.e., a Boolean combination (negation and conjunction) of guard propositions,
- *assig* is an *assignment*, i.e., a partial function, whose domain is $\mathsf{cntl}(V_{\mathsf{proc}})$, such that for every $x \in \mathsf{cntl}(V_{\mathsf{proc}})$, if $x \neq \mathit{fld}$ and $\mathit{assig}(x)$ is defined, then $\mathit{assig}(x) \in D_x$. □

Observe that we omit the special process control variable *fld* from the assignments *assig*. We do not allow the process itself to change the value of its failure flag *fld*, since its value is determined by the environment, as we will see below in Section 2.4. There, we will update the failure flag *fld* for every process in the context of the synchronous transition system, defined in Section 2.4.

Definition 2.11 (Result of applying $\varphi \to \mathit{assig}$)**.** Given a guarded assignment $\varphi \to \mathit{assig}$, values $\mathsf{n}, \mathsf{t}, \mathsf{r} \in \mathbb{N}$ for n, t, r, respectively, such that $\mathsf{t} < \mathsf{n}$, and a local state $\mathit{local} \in L(\mathsf{n})$, a control state $\mathit{control} \in C$ is the *result of applying* the guarded assignment $\varphi \to \mathit{assig}$ to

the local state *local* if for every $x \in \mathsf{cntl}(V_{\mathsf{proc}})$ we have

$$control.x = \begin{cases} assig(x) & \text{if } (local, \mathsf{n}, \mathsf{t}, \mathsf{r}) \models \varphi \text{ and } assig(x) \text{ is defined} \\ local.x & \text{otherwise} \end{cases} \quad \square$$

We will use a finite set G of guarded assignments to characterize the function $update_{n,t,r}$. To ensure that the function $update_{n,t,r}$ deterministically updates the control state of a process given its local state, we require that the guards of the guarded assignments in the set G are pairwise mutually exclusive. This restriction is imposed by the deterministic update of process control variables in distributed algorithms, which is based on the messages that a process has received by other processes, whose values come from a fixed set.

Definition 2.12 (Characterization of $update_{n,t,r}$)**.** Given a set G of guarded assignments, with pairwise mutually exclusive guards, and values $\mathsf{n}, \mathsf{t}, \mathsf{r} \in \mathbb{N}$ for n, t, r, respectively, such that $\mathsf{t} < \mathsf{n}$, the function $update_{\mathsf{n},\mathsf{t},\mathsf{r}} : L(\mathsf{n}) \to C$ maps a local state $local \in L(\mathsf{n})$ to a control state $control \in C$, such that $update_{\mathsf{n},\mathsf{t},\mathsf{r}}(local) = control$ iff there exists a guarded assignment $\varphi \to assig \in G$ where *control* is the result of applying $\varphi \to assig$ to *local*. $\square$

Example 2.4. Consider the pseudocode of the algorithm FloodMin, for $k = 1$ again. To characterize the control state update function $update_{n,t,r}$, we define a set G of guarded assignments that capture variable assignments in the pseudocode. Recall Example 2.1, where we identified two variable assignments: in line 6 of the pseudocode, the value of the variable `best` is updated within the loop starting at line 3, and in line 8, the value of the variable `dec` is updated outside of the loop.

To define the guarded assignments we will use the following guard propositions:

- $r > t + 1$, a termination guard proposition which checks whether the loop in line 3 of the pseudocode should terminate. As $k = 1$, the loop bound $\lfloor t/k \rfloor + 1$ equals to $t + 1$. Observe that we use the notation $r > t + 1$ for the termination guard proposition $r \geq t + 2$;
- $best = 0$, a control guard proposition which checks if the process stores the value 0 in its variable *best*;
- $best = 1$, a control guard proposition which checks if the process stores the value 1 in its variable *best*;
- $\exists j$ $\mathbf{msg}[j] = 0$, a neighborhood guard proposition which checks if a message with the value 0 was received from another process.

The set G of guarded assignments that characterize the function $update_{n,t,r}$ for the algorithm FloodMin, for $k = 1$, contains the following guarded assignments:

$$g_1 : \neg(r > t+1) \wedge (best = 0) \rightarrow best := 0$$

$$g_2 : \neg(r > t+1) \wedge (best = 1) \wedge (\exists j\, \mathbf{msg}[j] = 0) \rightarrow best := 0$$

$$g_3 : \neg(r > t+1) \wedge (best = 1) \wedge \neg(\exists j\, \mathbf{msg}[j] = 0) \rightarrow best := 1$$

$$g_4 : (r > t+1) \wedge (best = 0) \rightarrow dec := 0$$

$$g_5 : (r > t+1) \wedge (best = 1) \rightarrow dec := 1$$

The first three guarded assignments are used to model the assignment in line 6, which computes the minimum of the received values and assigns it to the variable `best`. This assignment happens inside the loop starting on line 3, which runs as long as $\neg(r > t+1)$ holds. If the value of the variable `best` is 0, it is already the minimal value, hence the value of `best` remains 0. This is captured by the guarded assignment g_1. Otherwise, if the value of `best` is 1, it can update its value to 0 only if it has received a value 0 from another process. This is captured by the guarded assignment g_2, where the neighborhood guard proposition $\exists j\, \mathbf{msg}[j] = 0$ checks if a value 0 has been received. If the value of `best` is 1 and no value 0 has been received from another process, the value of `best` remains 1, captured by the guarded assignment g_3.

The last two guarded assignments encode the assignment of the value of the variable `best` to the variable `dec` in line 8 of the pseudocode. As line 8 is outside of the loop starting on line 3, both guarded assignments contain the conjunct $(r > t+1)$, which checks if the round number is greater than $t+1$, i.e., if the loop terminated. The value assigned to *dec* in the guarded assignments g_4 and g_5 depends on the respective value of the variable *best*.

Consider the local state $local \in L(n)$ of the algorithm FloodMin for $k = 1$, where $\mathsf{n} = 6$, presented in Example 2.1. Suppose $\mathsf{t} = 3$ and $\mathsf{r} = 2$. From the values of the variables in the local state, we obtain $(local, \mathsf{n}, \mathsf{t}, \mathsf{r}) \models g_2.\varphi$, since $\mathsf{r} \leq \mathsf{t}+1$, the value of the variable *best* in the local state *local* is 1, and there exists a process $j = 2$ such that $local.\mathbf{msg}[j] = 0$. Thus, the result of applying the function $update_{\mathsf{n},\mathsf{t},\mathsf{r}}$ to the local state *local* is the control state where the value of *best* is assigned 0, while the values of *dec* and *fld* remain $\perp$, as in the local state *local*, i.e., $update_{\mathsf{n},\mathsf{t},\mathsf{r}}(local) = \langle 0, \perp, \perp \rangle$. □

As we will see in Section 2.4, the process functions will be used as building blocks of the transition relation of the synchronous transition system composing $\mathsf{n} \in \mathbb{N}$ copies of the process specification and one copy of the environment specification. Later, in Chapter 3, where we abstract this transition system, we will also have to abstract the process update function $update_{n,t,r}$. Towards this end, a key step will involve abstracting the guarded assignments. This step is done syntactically, by defining abstract versions of the guard propositions.

2.2 Environment Specification: Environment Variables

In this section, we propose an environment specification $\mathsf{env}(n, t, f)$, defined by environment variables.

Definition 2.13 (Environment variables)**.** Let V_{env} be a finite set of *environment variables*, which contains:

- the round number r, which ranges over $D_r = \mathbb{N}$,
- the one-dimensional array of crashed flags **cr**, which ranges over $D^n_{\mathbf{cr}} = \{\bot, \top\}^n$,
- the two-dimensional array of receiver lists **Rcv**, which ranges over $D^{n \cdot n}_{\mathbf{Rcv}} = \{\bot, \top\}^{n \cdot n}$. □

The environment has fixed variables $r, \mathbf{cr}$, and **Rcv**, each with a fixed set of values, $D_r = \mathbb{N}, D_{\mathbf{cr}} = \{\bot, \top\}$, and $D_{\mathbf{Rcv}} = \{\bot, \top\}$, respectively. The variable r is used by the environment to record the number of the current round that the processes are executing. The one-dimensional array **cr** flags the processes that will crash in the current round. If a cell j in the **cr** array has value $\top$, this means that the process j crashes in the current round. The two-dimensional array **Rcv** is used to encode the subset of processes to which a process sends a message in the current round. For two processes i, j, a value $\top$ in the (i, j)-th cell of **Rcv** denotes that process i will receive a message from j in the current round. Observe that, if j is a correct process, the j-th column in **Rcv** is filled with $\top$, and if j is a failed process (that is, if the failed flag *fld* in its current control state is $\top$), the j-th column in **Rcv** is filled with $\bot$. When the process j crashes in the current round, the j-th column of **Rcv** contains both values from the set $\{\bot, \top\}$.

We will define the environment variable updates in the context of the transition relation of the transition system obtained by composing n processes specifications and the environment specification in Section 2.4.

2.3 Synchronous System Specification: System Variables

Suppose that the process specification $\mathsf{proc}(n, t, f)$ is defined by process variables V_{proc} and process functions F_{proc}, and that the environment specification $\mathsf{env}(n, t, f)$ is defined by environment variables V_{env}, as in Sections 2.1 and 2.2, respectively. The synchronous system specification $\mathsf{Sys}(n, t, f)$, which is a composition of n process specifications and an environment specification, is defined by *system variables* V_{sys} and process functions F_{proc}. The system variables are used to store the values of the process variables of each of the n processes and the environment variables.

Definition 2.14 (System Variables V_{sys})**.** Given the set V_{proc} of process variables and the set V_{env} of environment variables, let V_{sys} be the finite set of *system variables*, which is the union of the sets of:

- *control variables* $\mathsf{cntl}(V_{\mathsf{sys}})$, containing one-dimensional array variables $\mathbf{x}$ of size n, that range over D_x^n, where $x \in \mathsf{cntl}(V_{\mathsf{proc}})$ is a process control variable,
- *neighborhood variables* $\mathsf{nbhd}(V_{\mathsf{sys}})$, containing two-dimensional array variables $\mathbf{Y}$ of size $n \times n$, ranging over $D_{\mathbf{y}}^{n \cdot n}$, where $\mathbf{y} \in nbhd(V)$ is a process neighborhood variable,
- environment variables V_{env}.

Intuitively, the control variables from $\mathsf{cntl}(V_{\mathsf{sys}})$ are used to store the values of the process control variables from $\mathsf{cntl}(V_{\mathsf{proc}})$, for each of the n processes in the system. Similarly, the neighborhood variables from $\mathsf{nbhd}(V_{\mathsf{sys}})$ are used to store the process neighborhood variables from $\mathsf{nbhd}(V_{\mathsf{proc}})$.

In Definition 2.3, we defined the special process control variable $\mathit{fld} \in \mathsf{cntl}(V_{\mathsf{proc}})$ and the special neighborhood variable $\mathbf{msg} \in \mathsf{nbhd}(V_{\mathsf{proc}})$. These special variables occur in the system variables as well. That is, we have special system variables $\mathbf{fld} \in \mathsf{cntl}(V_{\mathsf{sys}})$ and $\mathbf{Msg} \in \mathsf{nbhd}(V_{\mathsf{sys}})$, which are used to store the failure flags and messages for each process, respectively. In particular, for two processes i, j, the value of $\mathbf{Msg}[i, j]$ is equal to the value of $\mathbf{msg}[j]$ of process i, that is, to the message process j sent to process i.

Example 2.5. For the algorithm FloodMin, we have the following system variables V_{sys}:

- $\mathbf{best} \in \mathsf{cntl}(V_{\mathsf{sys}})$, a one-dimensional array of size n, ranging over D_{best}^n,
- $\mathbf{dec} \in \mathsf{cntl}(V_{\mathsf{sys}})$, a one-dimensional array of size n, ranging over D_{dec}^n,
- $\mathbf{fld} \in \mathsf{cntl}(V_{\mathsf{sys}})$, a one-dimensional array of size n, ranging over $\{\bot, \top\}^n$,
- $\mathbf{Msg} \in \mathsf{nbhd}(V_{\mathsf{sys}})$, a two-dimensional array of size $n \cdot n$, ranging over $\mathcal{M}^{n \cdot n}$,
- $r \in V_{\mathsf{env}}$, the round number, ranging over $\mathbb{N}$,
- $\mathbf{cr} \in V_{\mathsf{env}}$, a one-dimensional array of size n, storing crash flags for every process, and ranging over $\{\bot, \top\}^n$,
- $\mathbf{Rcv} \in V_{\mathsf{env}}$, a two-dimensional array of size $n \cdot n$, storing receiver lists for every process, and ranging over $\{\bot, \top\}^{n \cdot n}$. □

2.4 Synchronous Transition System

Let V_{sys} be the set of system variables and F_{proc} the set of process functions. For values $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ for the parameters n, t, f, that satisfy the resilience condition, we define a synchronous transition system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ as follows.

Definition 2.15 (System $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$)**.** A *synchronous transition system* is the tuple $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) = \langle S(\mathsf{n}, \mathsf{t}, \mathsf{f}), S_0(\mathsf{n}, \mathsf{t}, \mathsf{f}), T(\mathsf{n}, \mathsf{t}, \mathsf{f}) \rangle$, for $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, where:

- $S(\mathsf{n},\mathsf{t},\mathsf{f})$ is a set of *global states*,
- $S_0(\mathsf{n},\mathsf{t},\mathsf{f})$ is a set of *initial global states*,
- $T(\mathsf{n},\mathsf{t},\mathsf{f})$ is a *global transition relation*. □

The set $S(\mathsf{n},\mathsf{t},\mathsf{f})$ of global states, the set $S_0(\mathsf{n},\mathsf{t},\mathsf{f})$ of initial global states, and the transition relation $T(\mathsf{n},\mathsf{t},\mathsf{f})$ of the synchronous transition system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, for values $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ of the parameters that satisfy the resilience will be formally defined below, namely in Definitions 2.16, 2.17 and 2.18, respectively.

We denote by $\mathsf{STS}(n,t,f)$ the *parameterized synchronous transition system*, which is used to represent the infinite family $\{\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) = \langle S(\mathsf{n},\mathsf{t},\mathsf{f}), S_0(\mathsf{n},\mathsf{t},\mathsf{f}), T(\mathsf{n},\mathsf{t},\mathsf{f})\rangle \mid \mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ satisfy the resilience condition$\}$ of finite-state synchronous transition systems.

2.4.1 Global States

We now define the global states $S(\mathsf{n},\mathsf{t},\mathsf{f})$ of the synchronous transition system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$.

Definition 2.16 (Global states $S(\mathsf{n},\mathsf{t},\mathsf{f})$)**.** The set $S(\mathsf{n},\mathsf{t},\mathsf{f})$ of *global states* contains all valuations of the system variables V_{sys}, where:

- the variables $\mathsf{cntl}(V_{\mathsf{sys}}) \cup \{\mathbf{cr}\}$ are one-dimensional arrays of size n,
- and the variables $\mathsf{nbhd}(V_{\mathsf{sys}}) \cup \{\mathbf{Rcv}\}$ are two-dimensional arrays of size $\mathsf{n} \cdot \mathsf{n}$. □

Given a global state $s \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ and a system variable $v \in V_{\mathsf{sys}}$, we will denote by $s.v$ the value that the state s assigns to the variable v. Additionally, for a global state $s \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ and a process i, with $1 \leq i \leq \mathsf{n}$, we define the following auxiliary notions:

- the control state of process i in state s, denoted by $s.control_i$, and representing the tuple $\langle s.\mathbf{x}_1[i],\ldots,s.\mathbf{x}_{cv}[i]\rangle \in C$, where $cv = |\mathsf{cntl}(V_{\mathsf{sys}})|$,
- the i-th row of the neighborhood variable $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}})$ in state s, denoted by $s.row_i^{\mathbf{Y}}$, and representing the tuple $\langle s.\mathbf{Y}[i,1],\ldots,s.\mathbf{Y}[i,\mathsf{n}]\rangle \in D_{\mathbf{y}}^{\mathsf{n}}$,
- the local state of process i in state s, denoted by $s.local_i$, and representing the tuple $\langle s.control_i, s.row_i^{\mathbf{Y}_1},\ldots,s.row_i^{\mathbf{Y}_{nv}}\rangle \in L(\mathsf{n})$, where $nv = |\mathsf{nbhd}(V_{\mathsf{sys}})|$.

We now define which global states from the set $S(\mathsf{n},\mathsf{t},\mathsf{f})$ of global states are also initial global states of the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$.

Definition 2.17 (Initial global states $S_0(\mathsf{n},\mathsf{t},\mathsf{f})$)**.** Let $S_0(\mathsf{n},\mathsf{t},\mathsf{f}) \subseteq S(\mathsf{n},\mathsf{t},\mathsf{f})$ denote the set of *initial global states*. A global state $s \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ is *initial*, that is, $s \in S_0(\mathsf{n},\mathsf{t},\mathsf{f})$ if:

1. $s.control_i \in C_0$, where $1 \leq i \leq \mathsf{n}$ and C_0 is the set of initial control states,

2. $s.\mathbf{Y}[i,j] = \bot$, where $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}})$ and $1 \leq i,j \leq \mathsf{n}$,
3. $s.r = 0$, where $r \in V_{\mathsf{env}}$,
4. $s.\mathbf{cr}[i] = \bot$, where $\mathbf{cr} \in V_{\mathsf{env}}$ and $1 \leq i \leq \mathsf{n}$,
5. $s.\mathbf{Rcv}[i,j] = \bot$, where $\mathbf{Rcv} \in V_{\mathsf{env}}$ and $1 \leq i,j \leq \mathsf{n}$. □

Intuitively, a state $s \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ is an initial global state, that is, $s \in S_0(\mathsf{n},\mathsf{t},\mathsf{f})$ if the values it assigns to the different variables are initial values. To ensure this, a state has to satisfy the conditions (1)-(5) in Definition 2.17. Namely, condition (1) in Definition 2.17 requires that the control state of every process i, for $1 \leq i \leq \mathsf{n}$, in the state s is an initial control state, as defined in Definition 2.6. Condition (2) in Definition 2.17 requires that all cells of all neighborhood variables $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}})$ are empty. Finally, conditions (3)-(5) in Definition 2.17 restrict the way in which the environment variables are initialized. That is, the round number is set to zero, there are no processes that crashed, and the receivers lists are empty.

Example 2.6. Let $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ be a synchronous transition system, used to model the algorithm FloodMin for $k = 1$, where $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$. An initial global state $s \in S_0(\mathsf{n},\mathsf{t},\mathsf{f})$ of this system is given below:

$$s: \quad \mathbf{best} \begin{bmatrix}1\\1\\0\\1\\1\\0\end{bmatrix} \quad \mathbf{dec} \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \quad \mathbf{fld} \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \quad \mathbf{Msg} \begin{bmatrix}\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\end{bmatrix} \quad r \; 0 \quad \mathbf{cr} \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \quad \mathbf{Rcv} \begin{bmatrix}\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot&\bot\end{bmatrix}$$

Initially, all processes are in some initial control state, $\langle 0,\bot,\bot\rangle \in C$ or $\langle 1,\bot,\bot\rangle \in C$. The message array $s.\mathbf{Msg}$ is empty. The round number is 0, there are no crashed processes and the receiver lists are empty. □

2.4.2 Global Transition Relation

We now define the global transition relation $T(\mathsf{n},\mathsf{t},\mathsf{f}) \subseteq S(\mathsf{n},\mathsf{t},\mathsf{f}) \times S(\mathsf{n},\mathsf{t},\mathsf{f})$, which is a subset of the composition of three transition relations $\mathit{Env}(\mathsf{n},\mathsf{t},\mathsf{f})$, $\mathit{Snd}(\mathsf{n},\mathsf{t},\mathsf{f})$, and $\mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$, which are binary relations over the set $S(\mathsf{n},\mathsf{t},\mathsf{f})$ of states, and are formally defined below, in Definitions 2.19, 2.20, and 2.21, respectively. The global transition relation $T(\mathsf{n},\mathsf{t},\mathsf{f})$ is formally defined as follows.

Definition 2.18 (Global transition relation $T(\mathsf{n},\mathsf{t},\mathsf{f})$)**.** The *global transition relation* $T(\mathsf{n},\mathsf{t},\mathsf{f})$ is a binary relation $T(\mathsf{n},\mathsf{t},\mathsf{f}) \subseteq S(\mathsf{n},\mathsf{t},\mathsf{f}) \times S(\mathsf{n},\mathsf{t},\mathsf{f})$, where a pair $(s,s''') \in S(\mathsf{n},\mathsf{t},\mathsf{f}) \times S(\mathsf{n},\mathsf{t},\mathsf{f})$ of states is a *transition* in $T(\mathsf{n},\mathsf{t},\mathsf{f})$, i.e., $(s,s''') \in T(\mathsf{n},\mathsf{t},\mathsf{f})$ iff there exist $s',s'' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ such that:

- $(s, s') \in Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$,
- $(s', s'') \in Snd(\mathsf{n}, \mathsf{t}, \mathsf{f})$,
- $(s'', s''') \in Upd(\mathsf{n}, \mathsf{t}, \mathsf{f})$. □

A transition $(s, s''') \in T(\mathsf{n}, \mathsf{t}, \mathsf{f})$ encodes one round in the execution of the distributed algorithm. Recall that, in a round, processes send messages, receive messages, and update their variables based on the received messages. The sending and receiving of messages are encoded by the transition relation $Snd(\mathsf{n}, \mathsf{t}, \mathsf{f})$, while the update of process variables by the transition relation $Upd(\mathsf{n}, \mathsf{t}, \mathsf{f})$. In addition, the transition relation $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$ encodes the behavior of the faulty processes in the system. We now proceed by formally defining the three transition relations.

Transition Relation $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$. The first transition relation, $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$, is used to update the environment variables V_{env}.

Definition 2.19 (Transition relation $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$)**.** The transition relation $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$ is a binary relation $Env(\mathsf{n}, \mathsf{t}, \mathsf{f}) \subseteq S(\mathsf{n}, \mathsf{t}, \mathsf{f}) \times S(\mathsf{n}, \mathsf{t}, \mathsf{f})$, such that two states $s, s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ are in relation $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$, i.e., $(s, s') \in Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$, iff:

1. $s'.r = s.r + 1$,
2. $s'.\mathbf{cr}[i] = \bot$, if $s.\mathbf{fld}[i] = \top$, for $1 \leq i \leq \mathsf{n}$,
3. $|\{i \mid 1 \leq i \leq \mathsf{n} \text{ and } s.\mathbf{fld}[i] \vee s'.\mathbf{cr}[i]\}| \leq f$,
4. $s'.\mathbf{Rcv}[i, j] = \bot$, if $s.\mathbf{fld}[j] = \top$, for $1 \leq i, j \leq \mathsf{n}$,
5. $s'.\mathbf{Rcv}[i, j] = \top$, if $s.\mathbf{fld}[j] = \bot$ and $s'.\mathbf{cr}[j] = \bot$, for $1 \leq i, j \leq \mathsf{n}$,
6. $s'.\mathbf{x} = s.\mathbf{x}$, for $\mathbf{x} \in \mathsf{cntl}(V_{\mathsf{sys}})$,
7. $s'.\mathbf{Y} = s.\mathbf{Y}$, for $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}})$. □

Transitions from a state $s \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ to a state $s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ using the relation $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$ update the environment variables as follows. The condition (1) in Definition 2.19 expresses that the round number in the state s' is incremented. The environment non-deterministically chooses which processes will crash in the current round, while keeping the number of crashed processes below the parameter f. That is, $s'.\mathbf{cr}$ is updated such that the value $s'.\mathbf{cr}[i]$ for a process i that has already failed before is set to $\bot$ (condition (2) in Definition 2.19), and the value for the other processes that have not failed before is chosen non-deterministically. The condition (3) in Definition 2.19 ensures that there are at most f faults in every execution (by taking into account the already failed and the newly crashed processes). Next, the receiver lists for the next round are updated by flagging that no message is received from processes that failed in some previous round

(condition (4) in Definition 2.19), all messages are received from the correct processes (condition (5) in Definition 2.19), and by non-deterministically choosing which processes receive messages from the processes that crash in the current round. Conditions (6, 7) in Definition 2.19 ensure that the control and neighborhood variables are not updated by the transition $(s, s') \in \mathit{Env}(\mathsf{n}, \mathsf{t}, \mathsf{f})$.

Example 2.7. Consider again the synchronous transition system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ used to model the algorithm FloodMin for $k = 1$, where $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$. The state $s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ below is in relation $\mathit{Env}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ with the state s, presented in Example 2.6. The gray variables denote the variables that were not updated by the relation $\mathit{Env}(\mathsf{n}, \mathsf{t}, \mathsf{f})$.

$$
\begin{array}{lccccc}
s': & \textbf{best} & \textbf{dec} & \textbf{fld} & & \textbf{Msg} \\
 & \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} & \begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix} & \begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix} & \begin{bmatrix} \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \end{bmatrix}
\end{array}
\qquad
\begin{array}{ccc}
r & \textbf{cr} & \textbf{Rcv} \\
1 & \begin{bmatrix} \bot \\ \bot \\ \top \\ \bot \\ \bot \\ \top \end{bmatrix} & \begin{bmatrix} \top & \top & \top & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \top \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \end{bmatrix}
\end{array}
$$

In s', the environment increments the round number, picks processes 3 and 6 to crash in the current round. The environment also adds process 1 to the receiver list of process 3, and process 4 to the receiver list of process 6. This means that messages are sent, the message from process 3 will only be delivered to process 1, and the message from process 6 will only be delivered to process 4. □

Transition Relation $\mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$. The second transition relation, $\mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, is used to update the message array **Msg** with messages sent by the processes. In this transition relation, the message generation function *send_msg* is used to determine the message that the processes send. The environment determines which messages are delivered.

Definition 2.20 (Transition relation $\mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$)**.** The transition relation $\mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ is a binary relation $\mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \subseteq S(\mathsf{n}, \mathsf{t}, \mathsf{f}) \times S(\mathsf{n}, \mathsf{t}, \mathsf{f})$, such that two states $s, s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ are in relation $\mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, i.e., $(s, s') \in \mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, iff:

1. $s'.\mathbf{Msg}[i, j] = \begin{cases} \mathit{send\_msg}(s.\mathit{control}_j) & \text{if } s.\mathbf{Rcv}[i, j] = \top \\ \bot & \text{otherwise} \end{cases}$ for $1 \leq i, j \leq \mathsf{n}$,
2. $s'.\mathbf{x} = s.\mathbf{x}$, for $\mathbf{x} \in \mathsf{cntl}(V_{\mathsf{sys}})$,
3. $s'.\mathbf{Y} = s.\mathbf{Y}$, for $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}}) \setminus \{\mathbf{Msg}\}$,
4. $s'.r = s.r$, $s'.\mathbf{cr} = s.\mathbf{cr}$, and $s'.\mathbf{Rcv} = s.\mathbf{Rcv}$. □

The only variable updated in a transition $(s, s') \in \mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ is the message array **Msg**, captured by condition (1) in Definition 2.20. For two processes i, j, with $1 \leq i, j \leq \mathsf{n}$, the cell $s'.\mathbf{Msg}[i, j]$ of the message array in state s' is assigned the message sent from process j to process i, if i is in the receiver list of j in the state s, that is, if $s.\mathbf{Rcv}[i, j] = \top$. Otherwise, if process i is not in the receiver list of process j, then no message from j is delivered to i. The message that process j sends to process i is generated by the process function $\mathit{send\_msg} \in F_{\mathsf{proc}}$. The control variables, neighborhood variables (except **Msg**), and the environment variables remain unchanged, captured by conditions (2)–(4) in Definition 2.20.

Example 2.8. Consider again the synchronous transition system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ used to model the algorithm FloodMin for $k = 1$, where $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$. The state $s'' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ below is in relation $\mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ with the state s', presented in Example 2.7. The gray variables again denote the variables that were not updated by the relation $\mathit{Snd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$.

$$
s'' : \quad
\begin{array}{c|c|c|c|c|c}
\text{best} & \text{dec} & \text{fld} & & r & \text{cr} \\
\begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} &
\begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix} &
\begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix} &
\begin{bmatrix} 1 & 1 & 0 & 1 & 1 & \bot \\ 1 & 1 & \bot & 1 & 1 & \bot \\ 1 & 1 & \bot & 1 & 1 & \bot \\ 1 & 1 & \bot & 1 & 1 & 0 \\ 1 & 1 & \bot & 1 & 1 & \bot \\ 1 & 1 & \bot & 1 & 1 & \bot \end{bmatrix} &
1 &
\begin{bmatrix} \bot \\ \bot \\ \top \\ \bot \\ \bot \\ \top \end{bmatrix}
\end{array}
\quad
\begin{array}{c}
\text{Msg} \\
\hline
\text{Rcv} \\
\begin{bmatrix} \top & \top & \top & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \top \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \end{bmatrix}
\end{array}
$$

Observe that the message $\mathit{send\_msg}(s'.\mathit{control}_3) = 0$ from process 3 is only delivered to process 1, captured by setting $s''.\mathbf{Msg}[1, 3] = 0$. Similarly, the value 0 from process 6 is only delivered to process 4, as $s''.\mathbf{Msg}[4, 6] = 0$. Processes 3 and 6 do not send messages to the other processes, captured by setting $s''.\mathbf{Msg}[i, 3] = \bot$, for $i \neq 1$, and $s''.\mathbf{Msg}[j, 6] = \bot$, for $j \neq 4$. □

Transition Relation $\mathit{Upd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$. The third transition relation, $\mathit{Upd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, is used to update the process control and neighborhood variables, for each process.

Definition 2.21 (Transition relation $\mathit{Upd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$)**.** The transition relation $\mathit{Upd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ is a binary relation $\mathit{Upd}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \subseteq S(\mathsf{n}, \mathsf{t}, \mathsf{f}) \times S(\mathsf{n}, \mathsf{t}, \mathsf{f})$, such that two states $s, s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ are in relation $\mathit{Upd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, i.e., $(s, s') \in \mathit{Upd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, iff:

1. $s'.\mathbf{fld}[i] = s.\mathbf{fld}[i] \vee s.\mathbf{cr}[i]$, for $1 \leq i \leq \mathsf{n}$,
2. $s'.\mathit{control}_i = \begin{cases} \mathit{update}_{\mathsf{n},\mathsf{t},s.r}(s.\mathit{local}_i) & \text{if } s'.\mathbf{fld}[i] = \bot \\ s.\mathit{control}_i & \text{otherwise} \end{cases}$, for $1 \leq i \leq \mathsf{n}$,

3. $s'.\mathbf{Y}[i,j] = \begin{cases} translate_{\mathbf{y}}(s.\mathbf{Msg}[i,j]) & \text{if } s'.\mathbf{fld}[i] = \bot \\ s.\mathbf{Y}[i,j] & \text{otherwise} \end{cases}$, for $1 \leq i,j \leq \mathsf{n}$, and $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}}) \setminus \{\mathbf{Msg}\}$,
4. $s'.\mathbf{Msg}[i,j] = \bot$, for $1 \leq i,j \leq \mathsf{n}$,
5. $s'.r = s.r$, $s'.\mathbf{cr} = s.\mathbf{cr}$, and $s'.\mathbf{Rcv} = s.\mathbf{Rcv}$. □

A transition $(s,s') \in \mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$ updates the values of the control and neighborhood variables $\mathsf{cntl}(V_{\mathsf{sys}}) \cup \mathsf{nbhd}(V_{\mathsf{sys}})$. First, the failure flags stored in **fld** are updated, such that a process i is flagged as failed in s' if it was flagged as failed in s, or if it was chosen by the environment to crash in state s, captured by condition (1) in Definition 2.21. Then, the control state of every *correct* process i, for $1 \leq i \leq \mathsf{n}$, in the state s' is obtained using the process function $\mathit{update}_{\mathsf{n},\mathsf{t},s.r}$, as stated in condition (2) in Definition 2.21. That is, if the process i has not been flagged as failed in the state s', it calls the function $\mathit{update}_{\mathsf{n},\mathsf{t},s.r}$ with its local state $s.\mathit{local}_i$, and obtains its new control state, $s'.\mathit{control}_i$. Otherwise, if the process i has either crashed before, or in the current round, its control state is not updated. Next, all the neighborhood variables $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}})$, except the message array **Msg**, are updated using the corresponding process function $\mathit{translate}_{\mathbf{y}}$, which defines how the received messages are translated into values from the set $D_{\mathbf{y}}$. Again, the neighborhood variables are updated only for processes that are flagged as correct (condition (3) in Definition 2.21). The message array **Msg** is emptied, by setting the value of every cell to $\bot$ (condition (4) in Definition 2.21). Finally, condition (5) in Definition 2.21 ensures that the environment variables are not updated.

Example 2.9. We depict the state $s''' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ from the synchronous transition system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ used to model the algorithm FloodMin for $k = 1$, where $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$. The state $s''' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ below is in relation $\mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$ with the state s'', presented in Example 2.8. The gray variables again denote the variables that were not updated by the relation $\mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$.

s''' :

	best	**dec**	**fld**	**Msg**						r	**cr**	**Rcv**					
1	0	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	1	⊥	⊤	⊤	⊤	⊤	⊤	⊥
2	1	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥		⊥	⊤	⊤	⊥	⊤	⊤	⊥
3	0	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥		⊤	⊤	⊤	⊥	⊤	⊤	⊥
4	0	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥		⊥	⊤	⊤	⊥	⊤	⊤	⊤
5	1	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥		⊥	⊤	⊤	⊥	⊤	⊤	⊥
6	0	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥		⊤	⊤	⊤	⊥	⊤	⊤	⊥

For processes 1 and 4, the guard $g_2.\varphi$ of the guarded assignment g_2 from Example 2.4 on page 45 holds, as the current round number $s''.r = 1$ is not greater than $\mathsf{t}+1 = 4$, their value $s''.\mathbf{best}[i]$ is 1, and they both have received a message with value 0. Thus, processes 1 and 4 update their value to 0, that is, $s'''.\mathbf{best}[1] = s'''.\mathbf{best}[4] = 0$. Processes 3 and 6 were flagged as crashed, hence they update their failure flags $s'''.\mathbf{fld}[3]$ and $s'''.\mathbf{fld}[6]$ to

$\perp$, and do not update their control state. The other processes, 2 and 5, update their control state based on the guarded assignment g_3 from Example 2.4, as their value is 1, and they have not received any message with value 0.

By combining this state, and the states presented in Examples 2.6, 2.7, and 2.8, we obtain one transition $(s, s''') \in T(\mathsf{n}, \mathsf{t}, \mathsf{f})$ in the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ that models FloodMin, for $k = 1$, and for $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$. □

2.5 Temporal Logic for Specifying Properties

Given a synchronous fault-tolerant distributed algorithm, modeled using a synchronous system specification characterized by system variables and process functions, we will use a fragment of *indexed linear temporal logic (indexed-LTL)* [BCG89, EN95] to encode its properties. We define the syntax of the atomic propositions, the fragment $\mathcal{F}(k)$ of indexed-LTL that we will use, and how we will use indexed-LTL formulas to express properties of synchronous transition systems $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$.

2.5.1 Syntax and Semantics of a Fragment of Indexed-LTL

To define the syntax of the fragment $\mathcal{F}(k)$ of indexed-LTL formulas that we will use, we fix a set Ind of *indices*, a set Vars of *variables*, and a set Prop of *Boolean propositions*. Each variable $\mathbf{z} \in \mathsf{Vars}$ is indexed by a single index $i \in \mathsf{Ind}$, and $\mathbf{z}_i$ ranges over a set D_z of values. We define two kinds of atomic propositions:

1. $\mathbf{z}_i = v$, where $\mathbf{z} \in \mathsf{Vars}$ is a variable, $i \in \mathsf{Ind}$ is an index, and $v \in D_z$ is a value,
2. p, where $p \in \mathsf{Prop}$.

We use the following fragment of indexed-LTL:

Definition 2.22 (The fragment $\mathcal{F}(k)$)**.** For $\mathsf{k} \in \mathbb{N}$, we write $\mathcal{F}(\mathsf{k})$ for the set of indexed-LTL formulas of the form $\forall i_1 : \rho_1. \forall i_2 : \rho_2. \ldots \forall i_\mathsf{k} : \rho_\mathsf{k}.\ \psi(i_1, i_2, \ldots i_\mathsf{k})$, where:

1. $\rho_1, \rho_2, \ldots, \rho_\mathsf{k}$ are the *ranges* of the universally quantified indices $i_1, i_2, \ldots, i_\mathsf{k}$, respectively,
2. $\rho_1 = \rho_2 = \cdots = \rho_\mathsf{k} = \mathsf{Ind}$,
3. no universal quantifier occurs in $\psi(i_1, i_2, \ldots i_\mathsf{k})$,
4. existential quantifiers only occur in subformulas of the form $\exists j : \rho.\ (\mathbf{z}_j = v)$ or $\exists j : \rho.\ (\mathbf{z}_j \neq v)$, where:
   - ρ is the *range* of the existentially quantified index j,
   - $\rho = \mathsf{Ind}$,

- j is *fresh*, i.e., not bound by a universal quantifier.

We denote by $\mathcal{F}(k) = \bigcup_{\mathsf{k}\in\mathbb{N}} \mathcal{F}(\mathsf{k})$ the fragment of indexed-LTL formulas which we will use to express properties of synchronous fault-tolerant distributed algorithms. □

By the above definition, the universal and existential quantifiers in the formulas from the indexed-LTL fragment $\mathcal{F}(k)$ are bounded, and they range over the set Ind of indices. The existential quantifiers are only used to quantify over atomic propositions of the form $\mathbf{z}_j = v$, by introducing fresh indices, not bound by the leading universal quantifiers.

The formulas ϕ in $\mathcal{F}(k)$ are evaluated over tuples ς, that represent valuations of the variables Vars and the propositions Prop. That is, $\varsigma.\mathbf{z}[i] \in D_{\mathbf{z}}$ is the value assigned to a variable $\mathbf{z} \in \mathsf{Vars}$ and an index $i \in \mathsf{Ind}$ in the valuation ς. Similarly, $\varsigma.p \in \{\bot, \top\}$ is the value assigned to a Boolean proposition $p \in \mathsf{Prop}$ in the valuation ς.

We now define the semantics of the atomic propositions. An atomic proposition $\mathbf{z}_i = v$ holds in a valuation ς, that is, $\varsigma \models \mathbf{z}_i = v$ iff $i \in \rho$, where ρ is the range of the quantifier that bounds the index i, and $\varsigma.\mathbf{z}[i] = v$. Similarly, a Boolean proposition $p \in \mathsf{Prop}$ holds in a valuation ς, i.e., $\varsigma \models p$ iff $\varsigma.p = \top$. The semantics of the logical connectives, quantifiers and temporal operators is standard.

2.5.2 Expressing Properties of $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ in $\mathcal{F}(k)$

The fragments $\mathcal{F}(\mathsf{k})$, for $\mathsf{k} \in \mathbb{N}$, are rich enough to capture properties of fault-tolerant agreement algorithms.

To express properties of a given synchronous transition system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, for $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, we fix:

- $\mathsf{Ind} = \{1,\ldots,\mathsf{n}\}$ as the set of indices,
- $\mathsf{Vars} = \mathsf{cntl}(V_{\mathsf{sys}}) \cup \{\mathbf{cr}\}$ as the set of variables,
- $\mathsf{Prop} = \emptyset$ as the set of Boolean propositions.

The set Ind of indices contains an index for each process, while the set Vars contains the control variables $\mathsf{cntl}(V_{\mathsf{sys}})$, as well as the environment variable **cr**. These variables are indexed by one index in the global states of the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$. The set Prop of propositions is empty in this case since $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ does not have any Boolean variables, and no Boolean propositions are needed to express the properties of synchronous agreement algorithms whose executions are modeled by $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$. In Chapter 3, we will apply abstractions to $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, which will result in transition systems that have Boolean variables. Hence, there we will state formulas that use Boolean propositions.

The atomic propositions $\mathbf{z}_i = v$ are evaluated in global states $s \in S(\mathsf{n},\mathsf{t},\mathsf{f})$. That is, for an atomic proposition $\mathbf{z}_i = v$ and a global state $s \in S(\mathsf{n},\mathsf{t},\mathsf{f})$, we have $s \models \mathbf{z}_i = v$ iff $i \in \mathsf{Ind}$ and $s.\mathbf{z}[i] = v$.

Example 2.10. Below, we formalize the properties of FloodMin, for $k = 1$. Observe that these properties coincide with the properties for consensus algorithms (described in Section 1.6.1). The properties for k-set agreement, for $k = 2$, (Section 1.6.2) and non-blocking atomic commit (Section 1.6.3) can be encoded analogously.

- *Validity:* A value that is not an initial value of any process is not a value that is decided on. We express this using two formulas: one that checks whether all processes have an initial value different than 0, and an analogous formula that checks the same condition for the value 1.

$$\forall i : \mathsf{Ind}.\ (\exists j : \mathsf{Ind}.\ \mathbf{best}_j \neq 0) \vee \mathsf{G}((\mathbf{fld}_i = \bot \wedge \mathbf{dec}_i \neq \bot) \rightarrow \mathbf{dec}_i = 0)$$
$$\forall i : \mathsf{Ind}.\ (\exists j : \mathsf{Ind}.\ \mathbf{best}_j \neq 1) \vee \mathsf{G}((\mathbf{fld}_i = \bot \wedge \mathbf{dec}_i \neq \bot) \rightarrow \mathbf{dec}_i = 1)$$

- *Agreement:* No two correct processes decide on different values.

$$\forall i : \mathsf{Ind}.\, \forall j : \mathsf{Ind}.\, \mathsf{G}((\mathbf{fld}_i = \bot \wedge \mathbf{fld}_j = \bot \wedge \mathbf{dec}_i \neq \bot \wedge \mathbf{dec}_j \neq \bot) \rightarrow ((\mathbf{dec}_i = 0 \wedge \mathbf{dec}_j = 0) \vee (\mathbf{dec}_i = 1 \wedge \mathbf{dec}_j = 1)))$$

- *Termination*: Every correct process eventually decides.

$$\forall i : \mathsf{Ind}.\ \mathsf{F}(\mathbf{fld}_i = \bot \rightarrow \mathbf{dec}_i \neq \bot)$$

Observe that *Validity* and *Termination* are encoded with formulas from the fragment $\mathcal{F}(1)$, and the formula for *Agreement* is from the fragment $\mathcal{F}(2)$. □

2.6 Discussion

In this chapter, we addressed the formalization challenge for synchronous fault-tolerant distributed algorithms. We introduced a process specification that describes the behavior of a process running a synchronous distributed algorithm. Originally described using pseudocode, we introduce process variables and process functions to model this behavior. We defined a language of guarded assignments that allowed us to give a finite characterization to a parameterized process function, which represents an infinite family of finite process functions.

The environment specification using environment variables, as defined in this chapter, is tailored to encode how the processes behave under the crash fault model. We conjecture that different fault models can be encoded with analogous environment specifications, that is, by defining appropriate environment variables.

By decoupling the environment variables from the process variables, and by having the environment influence the values of the process variables only in the global transitions, our approach is in line with the modeling principles of the distributed systems community:

(1) the process specification describes how a process works in a "perfect world", where no faults occur, and (2) the reasoning about faults, and how they influence the execution of the algorithm, is applied when analyzing distributed computations in unreliable environments.

Moreover, given a system specification and values n, t, and f for the parameters n, t, and f, it is straightforward to formalize the transition system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ using formal specification languages. In this thesis, we produced TLA+ [Lam02] specifications of the transition systems for the algorithms EDAC, ESC, FairCons, FloodMin, FloodSet, and NBAC. The TLA+ specifications can be found in [Stob].

CHAPTER 3

Parameterized Model Checking by Abstraction

In this chapter, we propose an *abstraction technique* for solving the parameterized model checking problem for a given parameterized synchronous transition system $\mathsf{STS}(n,t,f)$ and an indexed-LTL formula $\phi \in \mathcal{F}(k)$ as input, both defined in Chapter 2.

This abstraction technique adapts and combines several existing verification methods [CTV06,Krs05,McM01,JKS+13,PXZ02], originally designed for asynchronous systems, to the synchronous setting. The main idea here is to reduce the problem of verifying infinitely many finite-state *concrete* synchronous transition systems $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ to the problem of model checking a single *abstract* synchronous transition system $\widehat{\mathsf{STS}}$, that simulates the behavior of every concrete system.

Figure 3.1 gives an overview of the abstraction technique. We start with a parameterized system $\mathsf{STS}(n,t,f)$ that we obtain as a result of the modeling presented in Chapter 2. We note that the obtained system is *symmetric* by design, as all the processes follow the same process specification $\mathsf{proc}(n,t,f)$. The abstraction we introduce is applied in two steps: first, the parameters t and f are eliminated using *pattern-based predicate abstraction*, and then n is eliminated using *data and counter abstraction*.

Symmetry. The systems $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ that we analyze are *symmetric* [EN96]. This means that instead of checking if a property expressed in the fragment $\mathcal{F}(k)$ of indexed-LTL holds for all processes, it suffices to check the given property for a small, fixed number of processes. The choice of the number of fixed processes depends on the properties we are interested in verifying. We discuss symmetry in Section 3.1.

Pattern-Based Predicate Abstraction. Recall that the set G of guarded assignments that we used to characterize the parameterized process function $update_{n,t,r}$ can

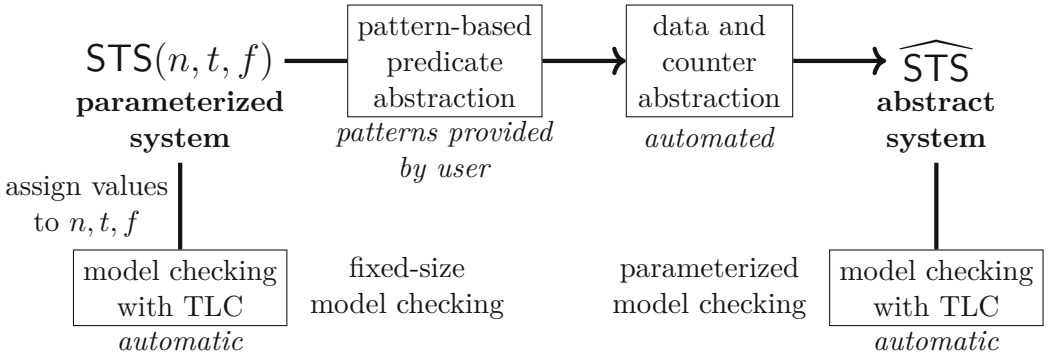


Figure 3.1: Overview of our abstraction-based technique

contain termination guard propositions, that feature the parameters n and t, as well as the round number r (Definition 2.12). To abstract away from the parameter t and the round number r, we introduce *pattern-based predicate abstraction* for termination guard propositions. Recall that the termination guard propositions (Definition 2.8) were used to check if the round number passes a given threshold, defined by the parameters. Many algorithms are designed in a way that they follow the same pattern, that is, they have the same termination conditions, captured by the same termination guard proposition.

The idea of pattern-based predicate abstraction is to identify these patterns and introduce predicates, that can be reused across different algorithms. For each termination guard proposition, we introduce a Boolean predicate, which is true when the termination guard proposition is satisfied. Further, for every newly defined predicate in this abstraction step, a constraint that ensures that the predicate is eventually satisfied is introduced. This eliminates the parameter t and the round number r, which is an environment variable.

The parameter f is used by the environment to ensure that there are no more than f faulty processes in the system. To eliminate the parameter f, we introduce a constraint which states that the crashes eventually stop appearing. The predicate abstraction step is described in more detail in Section 3.2.

Data and counter abstraction. Using ideas from [McM01, CMP04, Krs05], we fix a small number of processes (two or three), whose behaviors we keep concrete, and abstract the remaining processes depending on the current values of their variables. The number of processes whose behaviors we keep concrete is the same number of processes that we fixed in the symmetry step. Using data and counter abstraction [PXZ02, JKS+13], we reduce the size of the array variables in the global state from n to a fixed number, which depends on the number of fixed processes and the control states of the remaining processes, but is independent of n. The main idea is to store whether there are no processes (*zero*) or at least one process (*many*) that is in some particular control state. Sections 3.3 and 3.4 formally describe the zero-many data and counter abstractions, respectively.

Effective construction of $\widehat{\mathsf{STS}}$. The abstraction defined in this way does not give an effective method for building an abstract system $\widehat{\mathsf{STS}}$ from a given parameterized system $\mathsf{STS}(n,t,f)$. The challenge lies in the fact that we need to abstract the transition relation of every finite-state system in the infinite family $\{\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) \mid \mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ satisfy the resilience condition$\}$. In Section 3.5, we propose a constructive definition of the abstract initial states and transition relation. The key step in this method is the abstraction of the guarded assignments that we introduced in Chapter 2, and which were used to characterize the parameterized process function $update_{n,t,r}$.

Experiments. For several synchronous algorithms, we encoded both the parameterized system $\mathsf{STS}(n,t,f)$ and the abstract system $\widehat{\mathsf{STS}}$ using the specification language TLA+ [Lam02]. All of these algorithms were not automatically verified before, with the exception of FloodMin for $k=1$. We instantiated the parameterized system with small values for the parameters and ran a model checker on these fixed-size instances. Our experiments show that model checking fixed-size systems quickly runs into combinatorial state space explosion. We also ran a model checker on the abstract system, and concluded that parameterized model checking performs better than fixed-size model checking already for few (typically 5) processes. We discuss the experimental results in Section 5.6.

3.1 Symmetry

We observe that for all values $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ of the parameters n,t,f, that satisfy the resilience condition, the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ is a *symmetric transition system*. We adapt the definition of symmetric transition system from [ES96], which is based on the notion of a *permuted state*.

Let $\mathsf{Ind} = \{1,\ldots,\mathsf{n}\}$ denote a set of indices, and let $\mathsf{Sys}(\mathsf{n}) = \langle \Sigma(\mathsf{n}), \Sigma_0(\mathsf{n}), \mathcal{R}(\mathsf{n})\rangle$, for $\mathsf{n} \in \mathbb{N}$ be a transition system composed of n identical components. Given a permutation $\theta : \mathsf{Ind} \to \mathsf{Ind}$ of the set Ind of component indices and a state $\sigma \in \Sigma(n)$, a *permuted state* $\theta(\sigma)$ is the state obtained from σ by replacing every occurrence of the index $i \in \mathsf{Ind}$ with $\theta(i)$.

Definition 3.1 (Symmetric transition system (adapted from [ES96]))**.** A transition system $\mathsf{Sys}(\mathsf{n}) = \langle \Sigma(\mathsf{n}), \Sigma_0(\mathsf{n}), \mathcal{R}(\mathsf{n})\rangle$ consisting of $\mathsf{n} \in \mathbb{N}$ components is *symmetric*, if for every permutation $\theta : \mathsf{Ind} \to \mathsf{Ind}$ of the set $\mathsf{Ind} = \{1,\ldots,\mathsf{n}\}$ of component indices it holds that:

1. for every state $\sigma \in \Sigma(\mathsf{n})$, its permuted state $\theta(\sigma)$ is also a state in $\Sigma(\mathsf{n})$,
2. for every initial state $\sigma_0 \in \Sigma_0(\mathsf{n})$, its permuted state $\theta(\sigma_0)$ is also an initial state in $\Sigma_0(\mathsf{n})$,
3. for every transition $(\sigma,\sigma') \in \mathcal{R}(\mathsf{n})$, the pair $(\theta(\sigma),\theta(\sigma'))$ of permuted states is a transition in $\mathcal{R}(\mathsf{n})$. □

We now define how we build the permuted state in the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, for $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition.

Let $P = \{1,\ldots,\mathsf{n}\}$ denote the set of n processes in the synchronous transition system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) = \langle S(\mathsf{n},\mathsf{t},\mathsf{f}), S_0(\mathsf{n},\mathsf{t},\mathsf{f}), T(\mathsf{n},\mathsf{t},\mathsf{f})\rangle$. Recall that the system variables of the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ are indexed by indices from the set P.

Definition 3.2 (Permuted state in $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$)**.** Let $\theta : P \to P$ be a permutation of the set of processes and let $s \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ be a state. The *permuted state* $\theta(s)$ is obtained by replacing all occurrences of the index i in the state s by the index $\theta(i)$, as follows:

1. $\theta(s).\mathbf{x}[i] = s.\mathbf{x}[\theta(i)]$, for $\mathbf{x} \in \mathsf{cntl}(V_{\mathsf{sys}}) \cup \{\mathbf{cr}\}$,
2. $\theta(s).\mathbf{Y}[i,j] = s.\mathbf{Y}[\theta(i),\theta(j)]$, for $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}}) \cup \{\mathbf{Rcv}\}$. □

The following proposition follows from Definitions 3.1 and 3.2.

Proposition 3.1. *For every* $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition, the system* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *is a symmetric transition system.* □

Example 3.1. Recall Example 2.6 on page 50, where we presented an initial state $s \in S_0(\mathsf{n},\mathsf{t},\mathsf{f})$ in a system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ for the algorithm FloodMin, for $k = 1$, where $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$. Also, recall Example 2.9 on page 54, where we presented the state $s''' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$, such that $(s,s''') \in T(\mathsf{n},\mathsf{t},\mathsf{f})$. For $P = \{1,\ldots,6\}$, consider the permutation $\theta : P \to P$, where $\theta(i) = (i\%\mathsf{n}) + 1$, for $i \in P$. Below, we show each of the states $s, s''' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ from Examples 2.6 and 2.9, as well as their respective permuted states $\theta(s), \theta(s''')$.

We start with the initial state $s \in S_0(\mathsf{n},\mathsf{t},\mathsf{f})$:

$$
\begin{array}{lcccccc}
s: & \textbf{best} & \textbf{dec} & \textbf{fld} & & \textbf{Msg} & & r \quad \textbf{cr} \quad \textbf{Rcv} \\
& \begin{bmatrix} 1\\1\\0\\1\\1\\0 \end{bmatrix} & \begin{bmatrix} \bot\\\bot\\\bot\\\bot\\\bot\\\bot \end{bmatrix} & \begin{bmatrix} \bot\\\bot\\\bot\\\bot\\\bot\\\bot \end{bmatrix} & \multicolumn{3}{c}{\begin{bmatrix} \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot \end{bmatrix}} & 0 \quad \begin{bmatrix} \bot\\\bot\\\bot\\\bot\\\bot\\\bot \end{bmatrix} \quad \begin{bmatrix} \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot \end{bmatrix}
\end{array}
$$

$$
\begin{array}{lcccccc}
\theta(s): & \textbf{best} & \textbf{dec} & \textbf{fld} & & \textbf{Msg} & & r \quad \textbf{cr} \quad \textbf{Rcv} \\
& \begin{bmatrix} 0\\1\\1\\0\\1\\1 \end{bmatrix} & \begin{bmatrix} \bot\\\bot\\\bot\\\bot\\\bot\\\bot \end{bmatrix} & \begin{bmatrix} \bot\\\bot\\\bot\\\bot\\\bot\\\bot \end{bmatrix} & \multicolumn{3}{c}{\begin{bmatrix} \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot \end{bmatrix}} & 0 \quad \begin{bmatrix} \bot\\\bot\\\bot\\\bot\\\bot\\\bot \end{bmatrix} \quad \begin{bmatrix} \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot\\ \bot&\bot&\bot&\bot&\bot&\bot \end{bmatrix}
\end{array}
$$

In the permuted state $\theta(s)$ of the state s, we note that only the value of the variable **best** has changed. Namely, in the state s, processes 3 and 6 have value 0, while in the permuted state $\theta(s)$, processes $\theta(3) = 4$ and $\theta(6) = 1$ have value 0; the other processes have value 1. The permutation does not change the value of the other variables, as all cells in all arrays are set to $\bot$. We note that the permuted $\theta(s)$ is an initial state in $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, that is, $\theta(s) \in S_0(\mathsf{n},\mathsf{t},\mathsf{f})$, since it fulfills the conditions of Definition 2.17.

We now look at the state $s''' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$, which is in relation $T(\mathsf{n},\mathsf{t},\mathsf{f})$ with s, that is, $(s, s''') \in T(\mathsf{n},\mathsf{t},\mathsf{f})$, and its respective permuted state $\theta(s''')$.

s''' :

best	**dec**	**fld**	**Msg**	r	**cr**	**Rcv**
0	$\bot$	$\bot$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\bot$	$\top\ \top\ \top\ \top\ \top\ \bot$
1	$\bot$	$\bot$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\bot$	$\top\ \top\ \bot\ \top\ \top\ \bot$
0	$\bot$	$\top$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\top$	$\top\ \top\ \bot\ \top\ \top\ \bot$
0	$\bot$	$\bot$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$	1	$\bot$	$\top\ \top\ \bot\ \top\ \top\ \top$
1	$\bot$	$\bot$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\bot$	$\top\ \top\ \bot\ \top\ \top\ \bot$
0	$\bot$	$\top$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\top$	$\top\ \top\ \bot\ \top\ \top\ \bot$

$\theta(s''')$:

best	**dec**	**fld**	**Msg**	r	**cr**	**Rcv**
0	$\bot$	$\top$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\top$	$\bot\ \top\ \top\ \bot\ \top\ \top$
0	$\bot$	$\bot$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\bot$	$\bot\ \top\ \top\ \top\ \top\ \top$
1	$\bot$	$\bot$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\bot$	$\bot\ \top\ \top\ \bot\ \top\ \top$
0	$\bot$	$\top$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$	1	$\top$	$\bot\ \top\ \top\ \bot\ \top\ \top$
0	$\bot$	$\bot$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\bot$	$\top\ \top\ \top\ \bot\ \top\ \top$
1	$\bot$	$\bot$	$\bot\ \bot\ \bot\ \bot\ \bot\ \bot$		$\bot$	$\bot\ \top\ \top\ \bot\ \top\ \top$

In the state s''', we have that processes 1, 3, 4, and 6 have value 0, and the rest have value 1. Processes 3 and 6 have crashed and updated their failure flags. Process 1 is in the receiver list of process 3, and process 4 is in the receiver list of process 6. In the permuted state $\theta(s''')$, we have that processes $\theta(1) = 2$, $\theta(3) = 4$, $\theta(4) = 5$, and $\theta(6) = 1$ have value 0. Processes $\theta(3) = 4$ and $\theta(6) = 1$ have crashed and updated their failure flag. Process $\theta(1) = 2$ is in the receiver list of process $\theta(3) = 4$, and process $\theta(4) = 5$ is in the receiver list of process $\theta(6) = 1$. Thus, we conclude that the permuted state $\theta(s''')$ is a state in the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, for $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$, that is, $\theta(s''') \in S(\mathsf{n},\mathsf{t},\mathsf{f})$.

As for the states $s, s''' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ we have that $(s, s''') \in T(\mathsf{n},\mathsf{t},\mathsf{f})$, by Definition 2.18, there exist states s' and s'' such that $(s, s') \in \mathit{Env}(\mathsf{n},\mathsf{t},\mathsf{f})$, $(s', s'') \in \mathit{Snd}(\mathsf{n},\mathsf{t},\mathsf{f})$, and $(s'', s''') \in \mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$. In Examples 2.7 and 2.8 we showed the existence of the states s' and s'', respectively. It is easy to check that their respective permuted states, $\theta(s'), \theta(s'')$ are also states in the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, and that the following holds: $(\theta(s), \theta(s')) \in \mathit{Env}(\mathsf{n},\mathsf{t},\mathsf{f})$, $(\theta(s'), \theta(s'')) \in \mathit{Snd}(\mathsf{n},\mathsf{t},\mathsf{f})$, and $(\theta(s''), \theta(s''')) \in \mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$. Thus, we can conclude that $(\theta(s), \theta(s''')) \in T(\mathsf{n},\mathsf{t},\mathsf{f})$. □

Due to the symmetry of $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, we can fix a small number $\mathsf{m} \leq \mathsf{n}$ of processes that represent any m processes among the n processes in the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$. To determine m, we look at the indexed-LTL formulas that encode the properties of $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ that we are interested in verifying. As defined in Chapter 2, these properties were expressed in the fragment $\mathcal{F}(k)$ of indexed-LTL (Definition 2.22 on page 55).

Let $\Phi \subseteq \{\phi \mid \phi \in \mathcal{F}(k)\}$ denote the set of indexed-LTL formulas that encode the properties of $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$. To determine m, we take the maximal number of universal quantifiers that appear in the formulas $\phi \in \Phi$. That is, $\mathsf{m} = \max\{\mathsf{k} \in \mathbb{N} \mid \phi \in \Phi \text{ and } \phi \in \mathcal{F}(\mathsf{k})\}$.

Example 3.2. The *Validity* and *Termination* properties of consensus (see Example 2.10 on page 57) have a single universal quantifier, that is, they are expressed using formulas in $\mathcal{F}(1)$, while *Agreement* has two universal quantifiers, and is thus expressed using formulas in the fragment $\mathcal{F}(2)$. Therefore, for systems $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ that model consensus algorithms, we set $\mathsf{m} = 2$. □

The number m is small and fixed. For systems $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, such that $\mathsf{m} \leq \mathsf{n}$, we can exploit symmetry, where it suffices to check that the properties of the $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ hold only for the m processes, rather than for all n processes. Recall that the formulas in the indexed-LTL fragment $\phi \in \mathcal{F}(\mathsf{k})$ are of the shape $\forall i_1 : \mathsf{Ind}.\, \forall i_2 : \mathsf{Ind}.\, \ldots \forall i_{\mathsf{k}} : \mathsf{Ind}.\, \psi(i_1, i_2, \ldots i_{\mathsf{k}})$, where $\psi(i_1, i_2, \ldots i_{\mathsf{k}})$ is a formula with no universal quantifiers, and where the k universal quantifiers range over the set $\mathsf{Ind} = \{1, \ldots, \mathsf{n}\}$ of indices (Definition 2.22 on page 55). Intuitively, the formula ϕ is used to express a property that should be satisfied in the system; more precisely, the formula $\psi(i_1, i_2, \ldots i_{\mathsf{k}})$ is a property of each k-tuple of the n processes.

Example 3.3. We recall the indexed-LTL formula ϕ from the fragment $\mathcal{F}(2)$ that encodes the property *Agreement* in Example 2.10 on page 57.

$$\phi \equiv \forall i : \mathsf{Ind}.\, \forall j : \mathsf{Ind}.\, \psi(i,j), \text{ where}$$
$$\psi(i,j) \equiv \mathsf{G}((\mathbf{fld}_i = \bot \wedge \mathbf{fld}_j = \bot \wedge \mathbf{dec}_i \neq \bot \wedge \mathbf{dec}_j \neq \bot) \rightarrow ((\mathbf{dec}_i = 0 \wedge \mathbf{dec}_j = 0) \vee (\mathbf{dec}_i = 1 \wedge \mathbf{dec}_j = 1)))$$

The two universal quantifiers in the formula $\phi \in \mathcal{F}(2)$ range over the set $\mathsf{Ind} = \{1, \ldots \mathsf{n}\}$ of indices, for $\mathsf{n} \in \mathbb{N}$. The formula ϕ holds if for each pair $(\mathsf{i},\mathsf{j}) \in \{1,\ldots,\mathsf{n}\} \times \{1,\ldots,\mathsf{n}\}$ of processes, the formula $\psi(\mathsf{i},\mathsf{j})$ holds. The formula $\psi(\mathsf{i},\mathsf{j})$ expresses that it is always the case that if processes i and j have not failed and have decided, then their decision values are the same.

As we saw in Example 3.2, we set $\mathsf{m} = 2$. Assume that the underlying system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ for which we want to check the property ϕ is symmetric and that $\mathsf{m} \leq \mathsf{n}$. Due to the symmetry of the system, instead of checking $\psi(i,j)$ for each pair $(\mathsf{i},\mathsf{j}) \in \{1,\ldots,\mathsf{n}\} \times \{1,\ldots,\mathsf{n}\}$ of processes, it suffices to check $\psi(i,j)$ only for the pairs $(\mathsf{i}',\mathsf{j}') \in \{1,\ldots,\mathsf{m}\} \times \{1,\ldots,\mathsf{m}\} \subseteq \{1,\ldots,\mathsf{n}\} \times \{1,\ldots,\mathsf{n}\}$. □

To capture this formally, we introduce the restriction $\mathcal{F}^{\mathsf{m}}(k)$ of the indexed-LTL fragment $\mathcal{F}(k)$, where the universal quantifiers are used to quantify only over the fixed m processes. The syntax of $\mathcal{F}^{\mathsf{m}}(k)$ is expressed over a set Ind of indices, a set Vars of variables, a set Prop of Boolean propositions, and a value $\mathsf{m} \in \mathbb{N}$.

Definition 3.3 (The fragment $\mathcal{F}^{\mathsf{m}}(k)$)**.** For $\mathsf{k} \in \mathbb{N}$, we write $\mathcal{F}^{\mathsf{m}}(k)$ for the set of indexed-LTL formulas of the form $\forall i_1 : \rho_1. \forall i_2 : \rho_2. \ldots \forall i_{\mathsf{k}} : \rho_{\mathsf{k}}.\ \psi(i_1, i_2, \ldots i_{\mathsf{k}})$, where:

1. $\rho_1, \rho_2, \ldots, \rho_{\mathsf{k}}$ are the *ranges* of the universally quantified indices $i_1, i_2, \ldots, i_{\mathsf{k}}$, respectively,
2. $\rho_1 = \rho_2 = \cdots = \rho_{\mathsf{k}} = \{1, \ldots, \mathsf{m}\}$,
3. no universal quantifier occurs in $\psi(i_1, i_2, \ldots i_{\mathsf{k}})$,
4. existential quantifiers only occur in subformulas of the form $\exists j : \rho.\ (\mathbf{z}_j = v)$ or $\exists j : \rho.\ (\mathbf{z}_j \neq v)$, where:
   - ρ is the *range* of the existentially quantified index j,
   - $\rho = \mathsf{Ind}$,
   - j is *fresh*, i.e., not bound by a universal quantifier.

We denote by $\mathcal{F}^{\mathsf{m}}(k) = \bigcup_{\mathsf{k} \leq \mathsf{m}} \mathcal{F}^{\mathsf{m}}(\mathsf{k})$ the fragment of indexed-LTL formulas with at most m occurrences of universal quantifiers that range over the set $\{1, \ldots, \mathsf{m}\}$. □

As the universal quantifiers in formulas $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$ range over the set $\{1, \ldots, \mathsf{m}\}$, when evaluating the formulas ϕ^{m} in states $s \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ of the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, we need to distinguish whether the indices in the atomic propositions $\mathbf{z}_i = v$ are bound by universal or existential quantifiers. More precisely, given a state $s \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ and an atomic proposition $\mathbf{z}_i = v$, we say that $s \models \mathbf{z}_i = v$ iff either the index i is bound by an universal quantifier and $i \in \{1, \ldots, \mathsf{m}\}$, or the index i is bound by an existential quantifier and $i \in \mathsf{Ind}$; and $s.\mathbf{z}[i] = v$.

We define the translation of an indexed-LTL formula from $\mathcal{F}(\mathsf{k})$ to an indexed-LTL formula from $\mathcal{F}^{\mathsf{m}}(\mathsf{k})$, by changing the ranges of the universal quantifiers.

Definition 3.4 ($\mathcal{F}(\mathsf{k})$ to $\mathcal{F}^{\mathsf{m}}(\mathsf{k})$)**.** Let $\phi = \forall i_1 : \mathsf{Ind}. \forall i_2 : \mathsf{Ind}. \ldots \forall i_{\mathsf{k}} : \mathsf{Ind}.\ \psi(i_1, i_2, \ldots i_{\mathsf{k}})$ be an indexed-LTL formula from $\mathcal{F}(\mathsf{k})$ and let $\mathsf{m} \in \mathbb{N}$ be a fixed number such that $\mathsf{k} \leq \mathsf{m}$. We denote by $\mathsf{sym}(\phi, \mathsf{m})$ the formula:

$$\mathsf{sym}(\phi, \mathsf{m}) \equiv \forall i_1 : \{1, \ldots, \mathsf{m}\}. \forall i_2 : \{1, \ldots, \mathsf{m}\}. \ldots \forall i_{\mathsf{k}} : \{1, \ldots, \mathsf{m}\}.\ \psi(i_1, i_2, \ldots i_{\mathsf{k}})$$

from $\mathcal{F}^{\mathsf{m}}(\mathsf{k})$ corresponding to $\phi \in \mathcal{F}(\mathsf{k})$, where the universal quantifiers range over the set $\{1, \ldots, \mathsf{m}\}$. □

Example 3.4. Recall the indexed-LTL formula $\phi \in \mathcal{F}(2)$ that encodes the property *Agreement* in Example 2.10 on page 57, also given in Example 3.3. Both universal quantifiers in the formula ϕ range over the set Ind of indices. As we set $\mathsf{m} = 2$ in this case (see Example 3.2), the following formula from $\mathcal{F}^{\mathsf{m}}(2)$ is obtained by changing the ranges of the universal quantifiers in the formula ϕ:

$$\begin{aligned}\mathsf{sym}(\phi, \mathsf{m}) \equiv \forall i : \{1, \ldots, \mathsf{m}\}.\, \forall j : \{1, \ldots, \mathsf{m}\}.\\ \mathsf{G}((\mathbf{fld}_i = \bot \wedge \mathbf{fld}_j = \bot \wedge \mathbf{dec}_i \neq \bot \wedge \mathbf{dec}_j \neq \bot) \rightarrow\\ ((\mathbf{dec}_i = 0 \wedge \mathbf{dec}_j = 0) \vee (\mathbf{dec}_i = 1 \wedge \mathbf{dec}_j = 1)))\end{aligned}$$ □

The following proposition is a consequence of the symmetry of the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$.

Proposition 3.2 (Symmetry)**.** *Let* $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ *be a synchronous transition system, for* $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition and let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number. For every indexed-LTL formula* $\phi \in \mathcal{F}(\mathsf{k})$ *such that* $\mathsf{k} \leq \mathsf{m}$ *and* $\mathsf{m} \leq \mathsf{n}$*, we have:*

$$\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \models \mathsf{sym}(\phi, \mathsf{m}) \quad \textit{implies} \quad \mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \models \phi$$

where $\mathsf{sym}(\phi, \mathsf{m}) \in \mathcal{F}^{\mathsf{m}}(\mathsf{k})$. □

Observe that in the above proposition holds for systems $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ such that $\mathsf{m} \leq \mathsf{n}$. For a system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ where $\mathsf{m} > \mathsf{n}$, we do not need to use symmetry, as the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ is already small and the properties can be checked directly. In the following, we will consider systems where $\mathsf{m} \leq \mathsf{n}$, and where the safety and liveness properties are expressed in the restriction $\mathcal{F}^{\mathsf{m}}(k)$ of the indexed-LTL fragment $\mathcal{F}(k)$, defined in Definition 3.3.

3.2 Pattern-Based Predicate Abstraction

In this section, given a synchronous transition system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, for $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, we will show how we can obtain a synchronous transition system $\mathsf{STS}(\mathsf{n})$, whose states do not depend on the values t and f of the parameters t and f, respectively. We define *pattern-based predicate abstraction* and *verification conditions*, which will allow us to abstract the environment variable $r \in V_{\mathsf{env}}$ that stores the round number, as well as the parameters t and f. Recall that in a transition of the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, the round number is incremented by the environment and it is used when updating the process control states, where it is compared against an expression over the parameters n and t in termination guard propositions (see Definition 2.8), and the parameter f is used by the environment to control the number of new faults (see Definition 2.19).

To abstract the round number, in Section 3.2.1, we will introduce predicates for every termination guard proposition, occurring in the guards from the set G of guarded assignments, introduced in Definition 2.12 on page 45. The predicates we introduce are

based on *patterns*, which are common design features that reappear in various benchmarks that we are interested in verifying. One such pattern is that many algorithms have the same termination conditions, stating that the algorithm runs for $t+1$ rounds. For example, recall Figure 1.1 on page 6, which shows the pseudocode of the algorithm FloodSet and Figure 2.1 on page 40, which shows the pseudocode of the algorithm FloodMin, for $k = 1$. Both of these algorithms have a main loop that follows the same pattern, i.e., both loops run for $t + 1$ rounds, and thus we can use the same predicates in the pattern-based predicate abstraction.

By abstracting the round number using pattern-based predicate abstraction, we will also abstract the parameter t, which occurs only in the termination guard propositions. To abstract the parameter f, we will introduce a verification condition that ensures that at some point, new faults stop occurring. In addition to this verification condition, in Section 3.2.2 we introduce verification conditions, based on patterns occurring in our benchmarks, which will allow us to verify their safety and liveness properties.

3.2.1 Predicates

We now introduce *predicates* that capture the truth value of the termination guard propositions $r \geq \phi(n, t)$ in the states of the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, for $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition.

Recall the set G of guarded assignments, used to characterize the parameterized process function $update_{n,t,r}$, and introduced in Definition 2.12 on page 45. Let Term_G denote the set of termination guard propositions $r \geq \phi(n, t)$ that occur in the guards of the guarded assignments in the set G. We define a set Pred of *predicates*, containing $|\mathsf{Term}_G|$ predicates, one for each termination guard proposition in the set Term_G, such that a predicate $pr_{r \geq \phi(n,t)} \in \mathsf{Pred}$ is true if the termination guard proposition $r \geq \phi(n, t)$ that it represents is satisfied.

To define the system $\mathsf{STS}(\mathsf{n})$, we start by defining the system variables of the parameterized system $\mathsf{STS}(n) = \{\mathsf{STS}(\mathsf{n}) \mid \mathsf{n} \in \mathbb{N}\}$, which are derived from the system variables of the parameterized system $\mathsf{STS}(n, t, f)$, introduced in Definition 2.14 on page 47. Given the set V_{sys} of variables of the parameterized system $\mathsf{STS}(n, t, f)$, we define the set $V_{\mathsf{sys}}^{\mathsf{Pred}}$ of variables of the parameterized system $\mathsf{STS}(n)$.

Definition 3.5 (System Variables $V_{\mathsf{sys}}^{\mathsf{Pred}}$)**.** Given a set V_{sys} of variables of the parameterized system $\mathsf{STS}(n, t, f)$ and the set Pred of predicates, we define the set $V_{\mathsf{sys}}^{\mathsf{Pred}}$ of variables of the parameterized system $\mathsf{STS}(n)$ as the union of the sets:

- *control variables* $\mathsf{cntl}(V_{\mathsf{sys}}^{\mathsf{Pred}}) = \mathsf{cntl}(V_{\mathsf{sys}})$,
- *neighborhood variables* $\mathsf{nbhd}(V_{\mathsf{sys}}^{\mathsf{Pred}}) = \mathsf{nbhd}(V_{\mathsf{sys}})$,
- *environment variables* $V_{\mathsf{env}}^{\mathsf{Pred}} = \{\mathbf{cr}, \mathbf{Rcv}\}$,
- *predicates* Pred. □

Observe that the set $V_{\mathsf{env}}^{\mathsf{Pred}}$ of environment variables does not contain the environment variable $r \in V_{\mathsf{env}}$, which in $\mathsf{STS}(n,t,f)$ occurs only in the termination guard propositions from the set Term_G. By introducing the predicates Pred, which replace the termination guard propositions, we eliminate the variable r and the parameter t.

Example 3.5. Recall Example 2.4 on page 45, where we introduced the set G of guarded assignments for the algorithm $\mathsf{FloodMin}$, for $k = 1$. As the only termination guard proposition that occurs in the five guarded assignments in the set G is the expression $r > t + 1$, we have that the set Term_G contains a single termination guard proposition, that is, $\mathsf{Term}_G = \{r > t + 1\}$. Thus, the set Pred of predicates contains $|\mathsf{Term}_G| = 1$ predicate, that is $\mathsf{Pred} = \{pr_{r>t+1}\}$, where $pr_{r>t+1}$ is a predicate abstracting the expression $r > t + 1$. □

Global states $S(\mathsf{n})$. For a given $\mathsf{n} \in \mathbb{N}$, the set $S(\mathsf{n})$ of global states of the system $\mathsf{STS}(\mathsf{n})$, for $\mathsf{n} \in \mathbb{N}$, is the set of all valuations of $V_{\mathsf{sys}}^{\mathsf{Pred}}$, where the variables $\mathsf{cntl}(V_{\mathsf{sys}}^{\mathsf{Pred}}) \cup \{\mathbf{cr}\}$ are one-dimensional arrays of size n, the variables $\mathsf{nbhd}(V_{\mathsf{sys}}^{\mathsf{Pred}}) \cup \{\mathbf{Rcv}\}$ are two-dimensional arrays of size $\mathsf{n} \cdot \mathsf{n}$, and each predicate $pr \in \mathsf{Pred}$ is a Boolean variable, i.e., it ranges over $\{\bot, \top\}$. To define the set $S_0(\mathsf{n})$ of initial states and the transition relation $T(\mathsf{n})$ of $\mathsf{STS}(\mathsf{n})$, we will need the following abstraction mapping $\alpha_{\mathsf{n,t,f}}$.

Definition 3.6 (Abstraction mapping $\alpha_{\mathsf{n,t,f}}$)**.** Given parameter values $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, the abstraction mapping $\alpha_{\mathsf{n,t,f}} : S(\mathsf{n}, \mathsf{t}, \mathsf{f}) \to S(\mathsf{n})$ maps a state $s \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ to a state $\sigma \in S(\mathsf{n})$ as follows:

- $\sigma.\mathbf{x} = s.\mathbf{x}$, for every $\mathbf{x} \in \mathsf{cntl}(V_{\mathsf{sys}}^{\mathsf{Pred}})$,
- $\sigma.\mathbf{Y} = s.\mathbf{Y}$, for every $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}}^{\mathsf{Pred}})$,
- $\sigma.\mathbf{cr} = s.\mathbf{cr}$ and $\sigma.\mathbf{Rcv} = s.\mathbf{Rcv}$,
- $\sigma.pr_{r \geq \phi(n,t)} = \top$, for $pr_{r \geq \phi(n,t)} \in \mathsf{Pred}$, if $s.r \geq \phi(\mathsf{n}, \mathsf{t})$. □

Overapproximation. Given the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, and the abstraction mapping $\alpha_{\mathsf{n,t,f}}$, we define the system $\mathsf{STS}(\mathsf{n})$ as the *overapproximation* [CGL94] of $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ induced by $\alpha_{\mathsf{n,t,f}}$. We adapt the definition of overapproximation from [CGL94] to our setting, and use it to define the system $\mathsf{STS}(\mathsf{n})$.

Definition 3.7 (Overapproximation $\mathsf{STS}(\mathsf{n})$)**.** Given parameter values $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, a system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) = \langle S(\mathsf{n}, \mathsf{t}, \mathsf{f}), S_0(\mathsf{n}, \mathsf{t}, \mathsf{f}), T(\mathsf{n}, \mathsf{t}, \mathsf{f}) \rangle$, a set $S(\mathsf{n})$ of abstract states, and an abstraction mapping $\alpha_{\mathsf{n,t,f}}$, the abstract system $\mathsf{STS}(\mathsf{n}) = \langle S(\mathsf{n}), S_0(\mathsf{n}), T(\mathsf{n}) \rangle$ is the *overapproximation* of $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ induced by $\alpha_{\mathsf{n,t,f}}$ where:

- $S_0(\mathsf{n}) \subseteq S(\mathsf{n})$ is the set of *initial states*, such that $\sigma \in S_0(\mathsf{n})$ iff there exists $s \in S_0(\mathsf{n}, \mathsf{t}, \mathsf{f})$ with $\sigma = \alpha_{\mathsf{n,t,f}}(s)$,

- $T(\mathsf{n}) \subseteq S(\mathsf{n}) \times S(\mathsf{n})$ is the *transition relation*, such that $(\sigma, \sigma') \in T(\mathsf{n})$ iff there exist $s, s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$, with $\sigma = \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$ and $\sigma' = \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$, where $(s, s') \in T(\mathsf{n}, \mathsf{t}, \mathsf{f})$. □

Example 3.6. Recall Example 2.6 on page 50, where we presented one initial state $s \in S_0(\mathsf{n}, \mathsf{t}, \mathsf{f})$ in a system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ for the algorithm FloodMin, for $k = 1$, where $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$. Also recall Example 2.9 on page 54, which presented a state $s''' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$, which is in relation $T(\mathsf{n}, \mathsf{t}, \mathsf{f})$ with the state s. As we saw in Example 3.5, the set Pred of predicates for the FloodMin, for $k = 1$, contains the single predicate $pr_{r>t+1}$.

The state $\sigma \in S_0(\mathsf{n})$, given below, is the state obtained by applying the abstraction mapping $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$ to the state $s \in S_0(\mathsf{n}, \mathsf{t}, \mathsf{f})$ from Example 2.6, i.e., $\sigma = \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$:

$$\sigma: \begin{array}{c} \textbf{best} \\ \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{dec} \\ \begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{fld} \\ \begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{Msg} \\ \begin{bmatrix} \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{cr} \\ \begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{Rcv} \\ \begin{bmatrix} \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \end{bmatrix} \end{array} \quad \begin{array}{c} pr_{r>t+1} \\ \\ \bot \end{array}$$

According to Definition 3.6, the values of the control variables **best**, **dec**, **fld**, the neighborhood variable **Msg**, and the environment variables **cr** and **Rcv** are copied from the state $s \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ to the state $\sigma \in S(\mathsf{n})$. The predicate $pr_{r>t+1} \in \mathsf{Pred}$ has the value $\bot$ in the state σ, as in the state s we have $s.r = 0$, and since $0 < \mathsf{t} + 1 = 4$. By Definition 3.7, the state σ is an initial state of the system $\mathsf{STS}(\mathsf{n})$, that is, $\sigma \in S_0(\mathsf{n})$.

The state $\sigma''' \in S(\mathsf{n})$ given below is the result of applying the abstraction mapping $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$ to the state $s''' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ from Example 2.9, i.e., $\sigma''' = \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s''')$:

$$\sigma''': \begin{array}{c} \textbf{best} \\ \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{dec} \\ \begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{fld} \\ \begin{bmatrix} \bot \\ \bot \\ \top \\ \bot \\ \bot \\ \top \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{Msg} \\ \begin{bmatrix} \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot & \bot \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{cr} \\ \begin{bmatrix} \bot \\ \bot \\ \top \\ \bot \\ \bot \\ \top \end{bmatrix} \end{array} \quad \begin{array}{c} \textbf{Rcv} \\ \begin{bmatrix} \top & \top & \top & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \top \\ \top & \top & \bot & \top & \top & \bot \\ \top & \top & \bot & \top & \top & \bot \end{bmatrix} \end{array} \quad \begin{array}{c} pr_{r>t+1} \\ \\ \bot \end{array}$$

In the state $\sigma''' \in S(\mathsf{n})$, we still have that the value $\sigma'''.pr_{r>t+1}$ is set to $\bot$, as in the state $s \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$, we have $s.r = 1$, and as $1 < \mathsf{t} + 1 = 4$. By Definition 3.7, we have that the pair (σ, σ''') is a transition in the system $\mathsf{STS}(\mathsf{n})$, that is, $(\sigma, \sigma''') \in T(\mathsf{n})$. □

To be able to evaluate indexed-LTL formulas from the fragment $\mathcal{F}^{\mathsf{m}}(k)$ in $\mathsf{STS}(\mathsf{n})$, where $\mathsf{m} \leq \mathsf{n}$, we fix:

- $\mathsf{Ind} = \{1, \ldots, \mathsf{n}\}$ as the set of indices,
- $\mathsf{Vars} = \mathsf{cntl}(V_{\mathsf{sys}}^{\mathsf{Pred}}) \cup \{\mathbf{cr}\}$ as the set of variables,
- $\mathsf{Prop} = \mathsf{Pred}$ as the set of Boolean propositions.

The atomic propositions $\mathbf{z}_i = v$ are evaluated in the states $\sigma \in S(\mathsf{n})$ in the same way as they are evaluated in the states $s \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$, that is, we have $\sigma \models \mathbf{z}_i = v$ iff either the index i is bound by an universal quantifier and $i \in \{1, \ldots, \mathsf{m}\}$, or the index i is bound by an existential quantifier and $i \in \mathsf{Ind}$; and $\sigma.\mathbf{z}[i] = v$. For the Boolean propositions from the set Pred, we have $\sigma \models pr$ iff $\sigma.pr = \top$.

The following proposition is a consequence of overapproximation.

Proposition 3.3. *Let* $\mathsf{STS}(\mathsf{n})$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ *induced by the abstraction mapping* $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$*, for* $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition. Let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number such that* $\mathsf{m} \leq \mathsf{n}$*. For every formula* $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$*, we have:*

$$\mathsf{STS}(\mathsf{n}) \models \phi^{\mathsf{m}} \quad \textit{implies} \quad \mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \models \phi^{\mathsf{m}} \qquad \square$$

3.2.2 Pattern-Based Verification Conditions

We now identify a set of *pattern-based verification conditions*, containing indexed-LTL formulas that are satisfied by the executions of all synchronous fault-tolerant agreement algorithms that tolerate crash faults, and which we use as benchmarks. We will use these verification conditions to filter out executions in the abstract system $\mathsf{STS}(\mathsf{n})$ (as well as the executions in the abstract systems we will define in the remainder of this chapter), which are not reproducible in the concrete system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$.

The reason behind introducing these verification conditions is to ensure that the way in which the crash flags, stored in the environment variable $\mathbf{cr}$, and the newly introduced predicates from the set Pred are assigned values, corresponds to the way in which they are assigned values in underlying concrete system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$. For the predicates $pr \in \mathsf{Pred}$, the verification conditions ensure that each predicate $pr \in \mathsf{Pred}$ is eventually satisfied. For the environment variable $\mathbf{cr}$, the verification conditions ensure that faults eventually stop occurring.

We now formally introduce the pattern-based verification conditions, and sketch why it is sound to consider only executions that satisfy these conditions.

Definition 3.8 (Verification conditions)**.** The set VC of *verification conditions* imposed on the system $\mathsf{STS}(\mathsf{n})$ contains the following indexed-LTL formulas from the fragment $\mathcal{F}^{\mathsf{m}}(k)$:

1. $\mathsf{FG}\,\phi_{\mathsf{clean}}$, where $\phi_{\mathsf{clean}} \equiv \neg(\exists i : \mathsf{Ind}.\ \mathbf{cr}[i] = \top)$,
2. $\mathsf{FG}\,(pr)$, for every $pr \in \mathsf{Pred}$, and
3. $(\bigwedge_{pr \in \mathsf{Pred}} \neg pr)\ \mathsf{U}\ \phi_{\mathsf{clean}}$

We denote by $\phi_{\mathsf{VC}} = \bigwedge_{\phi \in \mathsf{VC}} \phi$ the conjunction of all verification conditions from the set VC. □

The formula $\phi_{\mathsf{VC}} \in \mathcal{F}^{\mathsf{m}}(0)$ is a formula in $\mathcal{F}^{\mathsf{m}}(k)$, as it is a conjunction of formulas in $\mathcal{F}^{\mathsf{m}}(k)$ with no occurrences of universal quantifiers. The formula ϕ_{clean} encodes the *clean round* condition. In the distributed systems literature, a clean round is a round in which no process exhibits faulty behavior. For our crash-tolerant benchmarks, we express the clean round condition using the formula ϕ_{clean}, which encodes that there is no process i that is flagged as crashed by the environment.

The pattern-based verification conditions we introduce in Definition 3.8 are tailored to system where the environment captures crash faults. Identifying verification conditions for environments that model other kinds of faults is a possible direction for future work.

In the following, we sketch why these conditions hold in a system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ and thus can be imposed on the overapproximation $\mathsf{STS}(\mathsf{n})$. Let $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ be a system that models the executions of a synchronous crash-tolerant agreement algorithm, for $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, i.e., $\mathsf{n} > \mathsf{t} \geq \mathsf{f}$.

1. The first condition, $\mathsf{FG}\, \phi_{\mathsf{clean}}$, where $\phi_{\mathsf{clean}} \equiv \neg(\exists i : \mathsf{Ind}.\ \mathbf{cr}[i] = \top)$, ensures that from some time on, there are no more processes that are flagged as crashed. It holds in $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ since the value f is fixed a priori in $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, and once the number of crashed processes reaches f, no more new processes are flagged as crashed by the environment.

2. The second condition, $\mathsf{FG}\,(pr)$, ensures that from some time on, the predicate $pr \in \mathsf{Pred}$ that abstracts some termination guard proposition $r \geq \phi(n, t) \in \mathsf{Term}_G$ becomes and remains true. As the round number, stored in the environment variable r, does not decrease with each transition, and as the values n and t are fixed a priori in $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, the truth value of the termination guard proposition $r \geq \phi(n, t)$ changes its truth value only once in each execution of $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$.

3. The third condition, $(\bigwedge_{pr \in \mathsf{Pred}} \neg pr)\, \mathsf{U}\, \phi_{\mathsf{clean}}$, ensures that the predicates $pr \in \mathsf{Pred}$, which abstract the termination guard propositions from the set Term_G, become true only after a clean round has occurred. This is typical for agreement algorithms that follow the same pattern, namely, algorithms which are designed to tolerate $\mathsf{t} \geq \mathsf{f}$ faults and whose main loops run for $\mathsf{t} + 1$ rounds. In this case, it holds that no process crashes in at least one among the $\mathsf{t} + 1$ rounds, that is, at least one of the $\mathsf{t} + 1$ rounds is a clean round.

Thus, for a system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ that is used to encode a synchronous fault-tolerant distributed algorithm that tolerates crash faults, i.e., where $\mathsf{n} > \mathsf{t} \geq \mathsf{f}$, we have $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \models \phi_{\mathsf{VC}}$, where ϕ_{VC} is the conjunction of the verification conditions from the set VC, introduced in Definition 3.8. We now state an auxiliary soundness result for our benchmarks that tolerate crash faults.

Proposition 3.4 (Soundness of $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$ for crash-tolerant benchmarks)**.** *Let* $\mathsf{STS}(\mathsf{n})$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *induced by the abstraction mapping* $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$*, for every* $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition. Suppose that* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *models a synchronous fault-tolerant distributed algorithm that tolerates crash faults, i.e., that* $\mathsf{n} > \mathsf{t} \geq \mathsf{f}$*. Let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number such that* $\mathsf{m} \leq \mathsf{n}$*. For every formula* $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$*, we have:*

$$\mathsf{STS}(\mathsf{n}) \models \phi_{VC} \rightarrow \phi^{\mathsf{m}} \quad \textit{implies} \quad \mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) \models \phi^{\mathsf{m}} \qquad \square$$

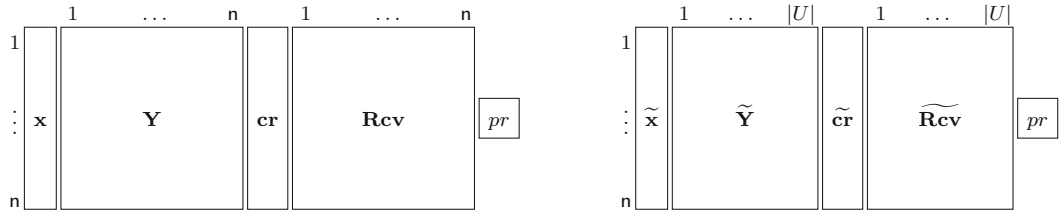
Proof. Suppose $\mathsf{STS}(\mathsf{n}) \models \phi_{\mathsf{VC}} \rightarrow \phi^{\mathsf{m}}$. From this and Proposition 3.3, we have $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) \models \phi_{\mathsf{VC}} \rightarrow \phi^{\mathsf{m}}$. By our assumption, $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) \models \phi_{\mathsf{VC}}$, and hence $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) \models \phi^{\mathsf{m}}$. $\square$

3.3 Zero-many Data Abstraction

We now define zero-many data abstraction, which is the first step towards eliminating the parameter n from the parameterized system $\mathsf{STS}(n)$. Recall that the parameter n defines the size of the array variables from the set $V_{\mathsf{sys}}^{\mathsf{Pred}}$. To build a finite-state system independent of n, we will fix the size of the array variables. Given a system $\mathsf{STS}(\mathsf{n})$, for every $\mathsf{n} \in \mathbb{N}$, we will fix a small value $\mathsf{m} \in \mathbb{N}$ of processes, such that $\mathsf{m} \leq \mathsf{n}$, whose behaviors we keep as in $\mathsf{STS}(\mathsf{n})$, and abstract the remaining $\mathsf{n} - \mathsf{m}$ processes based on their process control state. The number m of fixed processes is the same number that we fixed to define the indexed-LTL fragment $\mathcal{F}^{\mathsf{m}}(k)$. In the following, we will assume that a process i is one of the fixed processes if $1 \leq i \leq \mathsf{m}$. For the other processes j, with $\mathsf{m} < j \leq \mathsf{n}$, we store information about the control state of j. That is, for all process control states, we store whether no process from the $\mathsf{n} - \mathsf{m}$ processes is in a control state $\mathit{control} \in C$ (zero), or whether at least one process from the $\mathsf{n} - \mathsf{m}$ processes is in a control state $\mathit{control} \in C$ (many).

In the zero-many data abstraction step, we define an abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$ that maps states $S(\mathsf{n})$ of $\mathsf{STS}(\mathsf{n})$ to states $\widetilde{S}(\mathsf{n})$ of $\widetilde{\mathsf{STS}}(\mathsf{n})$, by fixing the size of the two-dimensional array variables from the set $\mathsf{nbhd}(V_{\mathsf{sys}}^{\mathsf{Pred}}) \cup \{\mathbf{Rcv}\}$. The resulting two-dimensional array variables $\widetilde{\mathbf{Y}} \in \mathsf{nbhd}(\widetilde{V}_{\mathsf{sys}})$ and $\widetilde{\mathbf{Rcv}}$ have a finite number of *columns*, which depends on the number m of fixed processes and the set C of process control states. The system $\widetilde{\mathsf{STS}}(\mathsf{n})$, obtained as the overapproximation of $\mathsf{STS}(\mathsf{n})$ induced by $\widetilde{\alpha}_{\mathsf{n}}$, still depends on n.

Figure 3.2 shows how the states $\sigma \in S(\mathsf{n})$ of a parameterized system $\mathsf{STS}(\mathsf{n})$ are mapped to states $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ of a parameterized system $\widetilde{\mathsf{STS}}(\mathsf{n})$ using the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$. For illustration purposes, we assume that the system $\mathsf{STS}(\mathsf{n})$ has a single control and neighborhood variable, $\mathbf{x}$ and $\mathbf{Y}$, respectively. The state $\sigma \in S(\mathsf{n})$ is shown in Figure 3.2a. Its two-dimensional array variables $\mathbf{Y}$ and $\mathbf{Rcv}$ are of size $\mathsf{n} \cdot \mathsf{n}$. Figure 3.2b illustrates the state $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$, which is a result of applying the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$ to the state σ. Its two-dimensional array variables $\widetilde{\mathbf{Y}}$ and $\widetilde{\mathbf{Rcv}}$ are of size $\mathsf{n} \cdot |U|$, where U is a finite set of abstract indices, defined below. Since the set U is of fixed size, the number of columns of the two-dimensional array variables $\mathbf{Y}$ and $\mathbf{Rcv}$ in the state $\widetilde{\sigma}$ is fixed for different



(a) $\sigma \in S(\mathsf{n})$ (b) $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$

Figure 3.2: Zero-many data abstraction with m fixed processes, where $U = \{1, \ldots, \mathsf{m}\} \cup C$.

values of n. This is not case for the two-dimensional array variables in the state σ, whose number of columns varies with the value of n.

We now proceed by formally defining the parameterized system $\widetilde{\mathsf{STS}}(\mathsf{n})$, and start by introducing its system variables. Their size depends on the set U of abstract indices, defined below.

Definition 3.9 (Abstract indices)**.** Let $\mathsf{m} \in \mathbb{N}$ be a fixed number and let C be the set of process control states. The set $U = \{1, \ldots, \mathsf{m}\} \cup C$ denotes the set of *abstract indices*. □

We will use the abstract indices from the set U as indices for the columns of the two-dimensional arrays variables of the system $\widetilde{\mathsf{STS}}(\mathsf{n})$. As the set U of abstract indices is finite, the two-dimensional array variables have a fixed number $|U|$ of columns, that does not depend on the parameter n.

Definition 3.10 (System variables $\widetilde{V}_{\mathsf{sys}}$)**.** Given a set $V_{\mathsf{sys}}^{\mathsf{Pred}}$ of variables of the parameterized system $\mathsf{STS}(n)$, we define the set $\widetilde{V}_{\mathsf{sys}}$ of variables of the parameterized system $\widetilde{\mathsf{STS}}(n)$ as the union of the sets of:

- *control variables* $\mathsf{cntl}(\widetilde{V}_{\mathsf{sys}}) = \mathsf{cntl}(V_{\mathsf{sys}}^{\mathsf{Pred}})$, containing one-dimensional array variables $\widetilde{\mathbf{x}}$ of size n, corresponding to $\mathbf{x} \in \mathsf{cntl}(V_{\mathsf{sys}}^{\mathsf{Pred}})$, and ranging over D_x^n,
- *neighborhood variables* $\mathsf{nbhd}(\widetilde{V}_{\mathsf{sys}})$, containing two-dimensional array variables $\widetilde{\mathbf{Y}}$ of size $n \cdot |U|$, corresponding to $\mathbf{Y} \in \mathsf{nbhd}(V_{\mathsf{sys}}^{\mathsf{Pred}})$, and ranging over $(2^{D_{\mathbf{y}}})^{n \cdot |U|}$,
- *environment variables* $\widetilde{V}_{\mathsf{env}} = \{\widetilde{\mathbf{cr}}, \widetilde{\mathbf{Rcv}}\}$, where
  - $\widetilde{\mathbf{cr}}$ is a one-dimensional array variable of size n, ranging over $\{\bot, \top\}^n$, and
  - $\widetilde{\mathbf{Rcv}}$ is a two-dimensional array variable of size $n \cdot |U|$, ranging over $(2^{\{\bot,\top\}})^{n \cdot |U|}$,
- *predicates* Pred. □

Global states $\widetilde{S}(\mathsf{n})$. The set $\widetilde{S}(\mathsf{n})$ of states of $\widetilde{\mathsf{STS}}(\mathsf{n})$, for $\mathsf{n} \in \mathbb{N}$, is the set of all valuations of $\widetilde{V}_{\mathsf{sys}}$, where the one-dimensional array variables $\mathsf{cntl}(\widetilde{V}_{\mathsf{sys}}) \cup \{\widetilde{\mathbf{cr}}\}$ are of size n, and the two-dimensional array variables $\mathsf{nbhd}(\widetilde{V}_{\mathsf{sys}}) \cup \{\widetilde{\mathbf{Rcv}}\}$ are of size $\mathsf{n} \cdot |U|$.

We introduce the notion of *witness*, which relates processes to abstract indices in a state.

Definition 3.11 (Witness)**.** An abstract index $u \in U$ is a *witness* of a process i in a state $\sigma \in S(\mathsf{n})$, for $1 \leq i \leq \mathsf{n}$ and $\mathsf{n} \in \mathbb{N}$, iff:

- $u \in \{1, \ldots, \mathsf{m}\}$ and $u = i$, or
- $u \in C$ and $u = \sigma.control_i$

The mapping $witness_{\mathsf{n}} : S(\mathsf{n}) \times U \to 2^{\{1,\ldots,\mathsf{n}\}}$ maps a state $\sigma \in S(\mathsf{n})$ and an abstract index $u \in U$, to a set of processes $witness_{\mathsf{n}}(\sigma, u) \subseteq \{1, \ldots, \mathsf{n}\}$, where:

$$witness_{\mathsf{n}}(\sigma, u) = \begin{cases} \{u\} & \text{if } u \in \{1, \ldots, \mathsf{m}\} \\ \{i \mid \mathsf{m} < i \leq \mathsf{n} \text{ and } \sigma.control_i = u\} & \text{if } u \in C \end{cases}$$

The notion of witness and the mapping $witness_{\mathsf{n}}$ are defined analogously for the states $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$. □

That is, an abstract index $u \in U$ witnesses a process in a state $\sigma \in S(\mathsf{n})$ (resp. $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$) iff it corresponds to one of the m fixed processes, or if there is a process whose current control state in σ (resp. $\widetilde{\sigma}$) is $u \in C$.

We now define an abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$, which maps states $S(\mathsf{n})$ of $\mathsf{STS}(\mathsf{n})$ to states $\widetilde{S}(\mathsf{n})$ of $\widetilde{\mathsf{STS}}(\mathsf{n})$, for $\mathsf{n} \in \mathbb{N}$.

Example 3.7. We now define the set U of abstract indices for the algorithm FloodMin for $k = 1$, and determine the sets of processes witnessed by different abstract indices in the states σ, σ''' of the system $\mathsf{STS}(\mathsf{n})$ for $\mathsf{n} = 6$, depicted in Example 3.6.

By Definition 3.9, the set U of abstract indices is the set $U = \{1, \ldots, \mathsf{m}\} \cup C$, where m is the number of fixed processes, and C is the set of process control states. In Example 3.2, we saw that for the algorithm FloodMin for $k = 1$, we set m to 2. Also, in Example 2.1 on page 40, we saw that the set C is the set $C = D_{best} \times D_{dec} \times D_{fld}$, where $D_{best} = \{0, 1\}$, $D_{dec} = \{\bot, 0, 1\}$, and $D_{fld} = \{\bot, \top\}$. That is, there are 12 control states in the set C of control states. For illustration purposes, we enumerate the control states in the set C, such that cnt_1 denotes the control state $\langle 0, \bot, \bot\rangle$, and cnt_{12} denotes the control state $\langle 1, 1, \top\rangle$.

For the state $\sigma \in S(\mathsf{n})$, we have that processes 3 and 6 are in control state $cnt_1 = \langle 0, \bot, \bot\rangle$, and processes 4 and 5 are in control state $cnt_7 = \langle 1, \bot, \bot\rangle$. Thus, for the state $\sigma \in S(\mathsf{n})$, we have the following witness sets:

$$witness_{\mathsf{n}}(\sigma, 1) = \{1\} \qquad witness_{\mathsf{n}}(\sigma, cnt_1) = \{3, 6\}$$
$$witness_{\mathsf{n}}(\sigma, 2) = \{2\} \qquad witness_{\mathsf{n}}(\sigma, cnt_7) = \{4, 5\}$$

In Example 3.5, we observe that processes 3 and 6 have updated their control state: in the state σ, they are both in the control state $cnt_1 = \langle 0, \bot, \bot \rangle$, while in the state σ''', they are both in the control state $cnt_2 = \langle 0, \bot, \top \rangle$. Further, process 4 updates its control state from $cnt_7 = \langle 1, \bot, \bot \rangle$ to $cnt_1 = \langle 0, \bot, \bot \rangle$. Since the processes 5 remains in the control state $cnt_7 = \langle 1, \bot, \bot \rangle$, we obtain the following witness sets for the state σ''':

$$witness_{\mathsf{n}}(\sigma''', 1) = \{1\} \qquad witness_{\mathsf{n}}(\sigma''', cnt_1) = \{4\}$$
$$witness_{\mathsf{n}}(\sigma''', 2) = \{2\} \qquad witness_{\mathsf{n}}(\sigma''', cnt_2) = \{3, 6\}$$
$$witness_{\mathsf{n}}(\sigma''', cnt_7) = \{5\}$$

The witness sets for the other control states in both cases are empty. □

Definition 3.12 (Abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$)**.** Given $\mathsf{n} \in \mathbb{N}$, the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}} : S(\mathsf{n}) \to \widetilde{S}(\mathsf{n})$ maps a state $\sigma \in S(\mathsf{n})$ to a state $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ as follows:

- $\widetilde{\sigma}.\widetilde{\mathbf{x}} = \sigma.\mathbf{x}$, for every $\widetilde{\mathbf{x}} \in \mathsf{cntl}(\widetilde{V}_{\mathsf{sys}})$,
- $\widetilde{\sigma}.\widetilde{\mathbf{Y}}[i, v] = \bigcup\{\sigma.\mathbf{Y}[i, j] \mid j \in witness_{\mathsf{n}}(\sigma, v)\}$, for $1 \leq i \leq \mathsf{n}$ and $v \in U$, and $\widetilde{\mathbf{Y}} \in \mathsf{nbhd}(\widetilde{V}_{\mathsf{sys}}) \cup \{\widetilde{\mathbf{Rcv}}\}$,
- $\widetilde{\sigma}.\widetilde{\mathbf{cr}} = \sigma.\mathbf{cr}$ and $\widetilde{\sigma}.pr = \sigma.pr$, for every $pr \in \mathsf{Pred}$. □

The abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$ fixes the number of columns of the two-dimensional array variables $\mathsf{nbhd}(\widetilde{V}_{\mathsf{sys}}) \cup \{\widetilde{\mathbf{Rcv}}\}$, such that each cell in a column indexed by $u \in U$ in $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ is a union of the cells in the column indexed by the processes witnessed by u in $\sigma \in S(\mathsf{n})$.

Example 3.8. Recall the two states $\sigma, \sigma''' \in S(\mathsf{n})$ given in Example 3.6, which are states of the system $\mathsf{STS}(\mathsf{n})$, where $\mathsf{n} = 6$, for the algorithm FloodMin for $k = 1$. Also, recall Example 3.7, where we defined the set U of abstract indices for the algorithm FloodMin for $k = 1$, as well as the witness sets for the two states $\sigma, \sigma''' \in S(\mathsf{n})$.

The state $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ (given below) is the result of applying the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$ to the state $\sigma \in S(\mathsf{n})$. We assume that the processes $\{1, \ldots, \mathsf{m}\}$ are fixed, that is, in our example, we will assume that processes 1 and 2 are fixed. We will abstract the other processes in our example, that is, processes 3, 4, 5, and 6, using their control state. We observe that in the state $\widetilde{\sigma}$, the control variables $\widetilde{\mathbf{best}}$, $\widetilde{\mathbf{dec}}$, and $\widetilde{\mathbf{fld}}$, as well as the environment variable $\widetilde{\mathbf{cr}}$ and the predicate $pr_{r>t+1}$, have the same values as in the state σ. The variables $\widetilde{\mathbf{Msg}}$ and $\widetilde{\mathbf{Rcv}}$ are transformed: they are two-dimensional arrays of size $\mathsf{n} \cdot |U|$, where the columns are indexed using the abstract indices from the set U. For the abstract indices for which we have non-empty witness sets, namely the indices 1, 2, cnt_1, and cnt_7, the columns of $\widetilde{\mathbf{Msg}}$ and $\widetilde{\mathbf{Rcv}}$ indexed by them store sets of values which are non-empty.

$$\widetilde{\sigma}: \quad \widetilde{\mathbf{best}} \quad \widetilde{\mathbf{dec}} \quad \widetilde{\mathbf{fld}} \quad \widetilde{\mathbf{Msg}}$$

$$\begin{bmatrix}1\\1\\0\\1\\1\\0\end{bmatrix} \quad \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \quad \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \quad \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & cnt_1 & cnt_2 & \cdots & cnt_7 & \cdots & cnt_{12} \end{array} \\ \begin{bmatrix} \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \end{bmatrix} \end{array}$$

$$\widetilde{\mathbf{cr}} \quad \widetilde{\mathbf{Rcv}} \quad pr_{r>t+1}$$

$$\begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \quad \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & cnt_1 & cnt_2 & \cdots & cnt_7 & \cdots & cnt_{12} \end{array} \\ \begin{bmatrix} \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset \end{bmatrix} \end{array} \quad \bot$$

We now show the result of applying the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$ to the state $\sigma''' \in S(\mathsf{n})$. The result of applying the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$ to the state σ''' is the state $\widetilde{\sigma}'''$, given below:

$$\widetilde{\sigma}''': \quad \widetilde{\mathbf{best}} \quad \widetilde{\mathbf{dec}} \quad \widetilde{\mathbf{fld}} \quad \widetilde{\mathbf{Msg}}$$

$$\begin{bmatrix}0\\1\\0\\0\\1\\0\end{bmatrix} \quad \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \quad \begin{bmatrix}\bot\\\bot\\\top\\\bot\\\bot\\\top\end{bmatrix} \quad \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & cnt_1 & cnt_2 & \cdots & cnt_7 & \cdots & cnt_{12} \end{array} \\ \begin{bmatrix} \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \end{bmatrix} \end{array}$$

$$\widetilde{\mathbf{cr}} \quad \widetilde{\mathbf{Rcv}} \quad pr_{r>t+1}$$

$$\begin{bmatrix}\bot\\\bot\\\top\\\bot\\\bot\\\top\end{bmatrix} \quad \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & cnt_1 & cnt_2 & \cdots & cnt_7 & \cdots & cnt_{12} \end{array} \\ \begin{bmatrix} \{\top\} & \{\top\} & \{\top\} & \{\bot,\top\} & \cdots & \{\top\} & \cdots & \emptyset \\ \{\top\} & \{\top\} & \{\top\} & \{\bot\} & \cdots & \{\top\} & \cdots & \emptyset \\ \{\top\} & \{\top\} & \{\top\} & \{\bot\} & \cdots & \{\top\} & \cdots & \emptyset \\ \{\top\} & \{\top\} & \{\top\} & \{\bot,\top\} & \cdots & \{\top\} & \cdots & \emptyset \\ \{\top\} & \{\top\} & \{\top\} & \{\bot\} & \cdots & \{\top\} & \cdots & \emptyset \\ \{\top\} & \{\top\} & \{\top\} & \{\bot\} & \cdots & \{\top\} & \cdots & \emptyset \end{bmatrix} \end{array} \quad \bot$$

Observe that the difference between the witness sets w.r.t. the states σ and σ''' induces a difference between the values of the variables $\widetilde{\mathbf{Msg}}$ and $\widetilde{\mathbf{Rcv}}$ in the states $\widetilde{\sigma} = \widetilde{\alpha}_{\mathsf{n}}(\sigma)$ and $\widetilde{\sigma}''' = \widetilde{\alpha}_{\mathsf{n}}(\sigma''')$: since no processes are witnessed by the control state cnt_2 in the state σ, and there are processes witnessed by the control state cnt_2 in the state σ''', the column indexed by cnt_2 in both $\widetilde{\mathbf{Msg}}$ and $\widetilde{\mathbf{Rcv}}$ in the state $\widetilde{\sigma}''' = \widetilde{\alpha}_{\mathsf{n}}(\sigma''')$ contains information

about the processes 3 and 6, which are witnessed by cnt_2. By the definition of the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$, the column of $\widetilde{\mathbf{Rcv}}$ indexed by cnt_2 in the state $\widetilde{\sigma}'''$ is a union of the columns of $\mathbf{Rcv}$ indexed by 3 and 6 in the state σ. Thus, $\widetilde{\sigma}'''.\widetilde{\mathbf{Rcv}}[1, cnt_2] = \{\bot, \top\}$ because $\sigma.\mathbf{Rcv}[1,3] = \bot$ and $\sigma.\mathbf{Rcv}[1,6] = \top$. Similarly, $\widetilde{\sigma}'''.\widetilde{\mathbf{Rcv}}[4, cnt_2] = \{\bot, \top\}$ because $\sigma.\mathbf{Rcv}[4,3] = \top$ and $\sigma.\mathbf{Rcv}[4,6] = \bot$.

Finally, note that in our example, where we set $\mathsf{n} = 6$, the states $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ have two-dimensional array variables with $|U| = 14$ columns, while in the states $\sigma \in S(\mathsf{n})$, the two-dimensional array variables have $\mathsf{n} = 6$ columns. While the number of columns is larger in the states obtained by applying the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$, this number is fixed for any value of n. On the contrary, the number of columns of the two-dimensional array variables for the states $\sigma \in S(\mathsf{n})$ changes depending on the value of n. □

Given the system $\mathsf{STS}(\mathsf{n})$ and the set $\widetilde{S}(\mathsf{n})$ of states, we define a system $\widetilde{\mathsf{STS}}(\mathsf{n})$ as the *overapproximation* of $\mathsf{STS}(\mathsf{n})$ induced by $\widetilde{\alpha}_{\mathsf{n}}$. We can express indexed-LTL formulas from the fragment $\mathcal{F}^{\mathsf{m}}(k)$ in the system $\widetilde{\mathsf{STS}}(\mathsf{n})$ in the same way as we did for the system $\mathsf{STS}(\mathsf{n})$. The following proposition is a consequence of overapproximation.

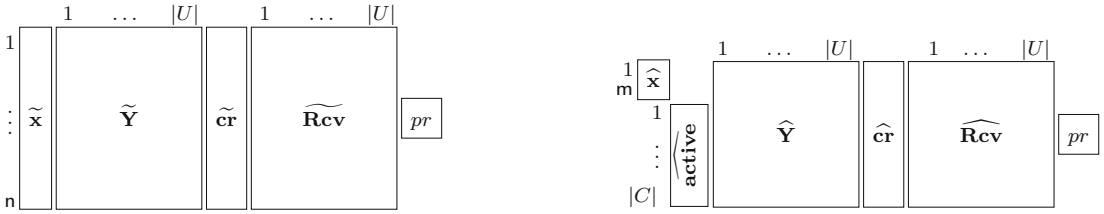
Proposition 3.5 (Soundness of $\widetilde{\alpha}_{\mathsf{n}}$)**.** *Let* $\widetilde{\mathsf{STS}}(\mathsf{n})$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n})$ *induced by the abstraction mapping* $\widetilde{\alpha}_{\mathsf{n}}$*, for* $\mathsf{n} \in \mathbb{N}$*. Let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number such that* $\mathsf{m} \leq \mathsf{n}$*. For every formula* $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$*, we have:*

$$\widetilde{\mathsf{STS}}(\mathsf{n}) \models \phi^{\mathsf{m}} \quad \textit{implies} \quad \mathsf{STS}(\mathsf{n}) \models \phi^{\mathsf{m}}$$ □

3.4 Zero-many Counter Abstraction

In this abstraction step, we define an abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$, which fixes the size of the one-dimensional array variables and the number of rows of the two-dimensional array variables using the abstract indices $u \in U$. For the fixed m processes, we store the values of the control variables in one-dimensional array variables of size m. For the remaining $\mathsf{n} - \mathsf{m}$ processes, we keep information whether there exists some process i, with $\mathsf{m} < i \leq \mathsf{n}$, in some control state $control \in C$ in a newly introduced variable control $\widehat{\mathbf{active}}$. For the two-dimensional array variables, we use the abstract indices to index their rows, in a way similar to the previous abstraction step, where we used the abstract indices to index their columns. This results in a finite abstract system $\widehat{\mathsf{STS}}$, which is not parameterized.

Figure 3.3 shows the effect of the abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$. In Figure 3.3a, we have the state $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ obtained as a result of the abstraction mapping $\widetilde{\alpha}_{\mathsf{n}}$, which we depicted in Figure 3.2. In Figure 3.3b, we have the state $\widehat{\sigma} \in \widehat{S}$, which is the result of applying the abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$ to the state $\widetilde{\sigma}$. In the state $\widehat{\sigma}$, the control variable $\widehat{\mathbf{x}} \in \mathsf{cntl}(\widehat{V}_{\mathsf{sys}})$ is a one-dimensional array of size m, storing the values of the process control variable $x \in \mathsf{cntl}(V_{\mathsf{proc}})$ for each of the m fixed processes. We abstract the control variables of the remaining processes by introducing the one-dimensional array variable $\widehat{\mathbf{active}}$. This variable stores if there are zero or many processes in a given control state,



(a) $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ (b) $\widehat{\sigma} \in \widehat{\Sigma}$

Figure 3.3: Zero-many counter abstraction with m fixed processes.

for each control state *control* $\in C$. That is, if the set of processes witnessed by a control state *control* $\in C$ in the state $\widetilde{\sigma}$ is empty, the variable $\widehat{\textbf{active}}$ in the state $\widehat{\sigma}$ stores the value 0 at index *control*. Otherwise, if there are processes witnessed by the control state *control* $\in C$ in the state $\widetilde{\sigma}$, the variable $\widehat{\textbf{active}}$ in the state $\widehat{\sigma}$ stores the value many at index *control*. The two-dimensional array variables $\widehat{\mathbf{Y}}$ and $\widehat{\textbf{Rcv}}$ in the state $\widehat{\sigma}$ are of size $|U| \cdot |U|$, and the one-dimensional array variable $\widehat{\textbf{cr}}$ is of size $|U|$. Thus, all array variables in $\widehat{\sigma} \in \widehat{S}$ are of size which is finite and independent of the parameters.

We now define the system variables of the system $\widehat{\mathsf{STS}}$.

Definition 3.13 (System variables $\widehat{V}_{\mathsf{sys}}$)**.** The set $\widehat{V}_{\mathsf{sys}}$ of variables of the system $\widehat{\mathsf{STS}}$ is the union of the sets of:

- *control variables* $\mathsf{cntl}(\widehat{V}_{\mathsf{sys}})$, containing one-dimensional array variables:
  - $\widehat{\mathbf{x}}$ of size m, ranging over D_x^{m},
  - $\widehat{\textbf{active}}$ of size $|C|$, ranging over $\{0, \mathsf{many}\}^{|C|}$,
- *neighborhood variables* $\mathsf{nbhd}(\widehat{V}_{\mathsf{sys}})$, containing two-dimensional array variables $\widehat{\mathbf{Y}}$ of size $|U| \cdot |U|$, ranging over $(2^{D_{\mathbf{y}}})^{|U| \cdot |U|}$,
- *environment variables* $\widehat{V}_{\mathsf{env}} = \{\widehat{\textbf{cr}}, \widehat{\textbf{Rcv}}\}$, where:
  - $\widehat{\textbf{cr}}$ is a one-dimensional array variable of size $|U|$, ranging over $(2^{\{\bot,\top\}})^{|U|}$, and
  - $\widehat{\textbf{Rcv}}$ is a two-dimensional array variable of size $|U| \cdot |U|$, ranging over $(2^{\{\bot,\top\}})^{|U| \cdot |U|}$,
- *predicates* Pred. □

For every control state *control* $\in C$, the control variable $\widehat{\textbf{active}} \in \mathsf{cntl}(\widehat{V}_{\mathsf{sys}})$ is used to store whether there is no process whose control state is *control*, in which case $\widehat{\textbf{active}}[\mathit{control}] = 0$, or if there is at least one process whose control state is *control*, in which case $\widehat{\textbf{active}}[\mathit{control}] = \mathsf{many}$.

Global states $\widehat{S}$. The set $\widehat{S}$ of states of $\widehat{\mathsf{STS}}$ is the set of all valuations of $\widehat{V}_{\mathsf{sys}}$, which is finite and does not depend on the value of the parameter n. This is because there are finitely many variables in $\widehat{V}_{\mathsf{sys}}$, and all of them range over a finite set of values and have a finite size.

Definition 3.14 (Abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$)**.** The abstraction mapping $\widehat{\alpha}_{\mathsf{n}} : \widetilde{S}(\mathsf{n}) \to \widehat{S}$ maps a state $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ to a state $\widehat{\sigma} \in \widehat{S}$, for $\mathsf{n} \in \mathbb{N}$, as follows:

- $\widehat{\sigma}.\widehat{\mathbf{x}}[u] = \widetilde{\sigma}.\widetilde{\mathbf{x}}[u]$, for $\widehat{\mathbf{x}} \in \mathsf{cntl}(\widehat{V}_{\mathsf{sys}})$ and $u \in \{1, \ldots, \mathsf{m}\}$,
- $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \begin{cases} \mathsf{many} & \text{if } \exists i \text{ with } \mathsf{m} < i \leq \mathsf{n} \text{ and } i \in \mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, u) \\ 0 & \text{otherwise} \end{cases}$, for $u \in C$,
- $\widehat{\sigma}.\widehat{\mathbf{Y}}[u, v] = \bigcup\{\widetilde{\sigma}.\widetilde{\mathbf{Y}}[i, v] \mid i \in \mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, u)\}$, for $u, v \in U$, and $\widehat{\mathbf{Y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{sys}}) \cup \{\widehat{\mathbf{Rcv}}\}$,
- $\widehat{\sigma}.\widehat{\mathbf{cr}}[u] = \bigcup\{\widetilde{\sigma}.\widetilde{\mathbf{cr}}[i] \mid i \in \mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, u)\}$, for $u \in U$,
- $\widehat{\sigma}.pr = \widetilde{\sigma}.pr$, for every $pr \in \mathsf{Pred}$. □

Example 3.9. We now apply the abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$ to the states $\widetilde{\sigma}, \widetilde{\sigma}''' \in \widetilde{S}(\mathsf{n})$, given in Example 3.8, which are states of the system $\widetilde{\mathsf{STS}}(\mathsf{n})$, where $\mathsf{n} = 6$, for the algorithm FloodMin for $k = 1$. First, we determine the sets of processes witnessed by different abstract indices in the state $\widetilde{\sigma}$, defined in Definition 3.11.

$$\mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, 1) = \{1\} \qquad \mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, \mathit{cnt}_1) = \{3, 6\}$$
$$\mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, 2) = \{2\} \qquad \mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, \mathit{cnt}_7) = \{4, 5\}$$

The result of applying the abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$ to the state $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ is the state $\widehat{\sigma} \in \widehat{S}$, given below.

$$
\widehat{\sigma}:\quad
\widehat{\mathbf{best}}=\begin{bmatrix}1\\1\end{bmatrix}\qquad
\widehat{\mathbf{dec}}=\begin{bmatrix}\bot\\\bot\end{bmatrix}\qquad
\widehat{\mathbf{fld}}=\begin{bmatrix}\bot\\\bot\end{bmatrix}
$$

$$
\widehat{\mathbf{active}}=\begin{array}{c}\\ cnt_1\\ cnt_2\\ \vdots\\ cnt_7\\ \vdots\\ cnt_{12}\end{array}\!\!\begin{bmatrix}\\ \mathsf{many}\\ 0\\ \vdots\\ \mathsf{many}\\ \vdots\\ 0\end{bmatrix}
$$

$$
\widehat{\mathbf{Msg}}=\begin{array}{c}\\ 1\\ 2\\ cnt_1\\ cnt_2\\ \vdots\\ cnt_7\\ \vdots\\ cnt_{12}\end{array}\!\!\begin{array}{c}\begin{array}{cccccccc}1 & 2 & cnt_1 & cnt_2 & \cdots & cnt_7 & \cdots & cnt_{12}\end{array}\\ \begin{bmatrix}\{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset\\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset\\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset\\ \emptyset & \emptyset & \emptyset & \emptyset & \cdots & \emptyset & \cdots & \emptyset\\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots\\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset\\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots\\ \emptyset & \emptyset & \emptyset & \emptyset & \cdots & \emptyset & \cdots & \emptyset\end{bmatrix}\end{array}
$$

$$
pr_{r>t+1}\qquad
\widehat{\mathbf{cr}}=\begin{array}{c}\\ 1\\ 2\\ cnt_1\\ cnt_2\\ \vdots\\ cnt_7\\ \vdots\\ cnt_{12}\end{array}\!\!\begin{bmatrix}\\ \{\bot\}\\ \{\bot\}\\ \{\bot\}\\ \emptyset\\ \vdots\\ \{\bot\}\\ \vdots\\ \emptyset\end{bmatrix}
$$

$$
\bot\qquad
\widehat{\mathbf{Rcv}}=\begin{array}{c}\\ 1\\ 2\\ cnt_1\\ cnt_2\\ \vdots\\ cnt_7\\ \vdots\\ cnt_{12}\end{array}\!\!\begin{array}{c}\begin{array}{cccccccc}1 & 2 & cnt_1 & cnt_2 & \cdots & cnt_7 & \cdots & cnt_{12}\end{array}\\ \begin{bmatrix}\{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset\\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset\\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset\\ \emptyset & \emptyset & \emptyset & \emptyset & \cdots & \emptyset & \cdots & \emptyset\\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots\\ \{\bot\} & \{\bot\} & \{\bot\} & \emptyset & \cdots & \{\bot\} & \cdots & \emptyset\\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots\\ \emptyset & \emptyset & \emptyset & \emptyset & \cdots & \emptyset & \cdots & \emptyset\end{bmatrix}\end{array}
$$

The control variables $\widehat{\mathbf{best}}$, $\widehat{\mathbf{dec}}$, and $\widehat{\mathbf{fld}}$ are one-dimensional arrays of size $\mathsf{m} = 2$, storing the values of the control variables for the processes 1 and 2. The values of the control variables of processes 3, 4, 5, and 6 are abstracted in the one-dimensional array variable $\widehat{\mathbf{active}}$, which is indexed by the control states from the set $C = D_{best} \times D_{dec} \times D_{fld}$. Since processes 3 and 6 are in control state $cnt_1 = \langle 0, \bot, \bot \rangle$ in state $\widetilde{\sigma}$, in the state $\widehat{\sigma}$, we set $\widehat{\mathbf{active}}[cnt_1] = \mathsf{many}$. Similarly, we have $\widehat{\mathbf{active}}[cnt_7] = \mathsf{many}$, because processes 4 and 5 are in control state $cnt_7 = \langle 1, \bot, \bot \rangle$ in state $\widetilde{\sigma}$. For the remaining control states $cnt \in C$, where $cnt \neq cnt_1$ and $cnt \neq cnt_7$, we have $\widehat{\mathbf{active}}[cnt] = 0$. The variables $\widehat{\mathbf{cr}}$, $\widehat{\mathbf{Msg}}$, and $\widehat{\mathbf{Rcv}}$ are indexed by the abstract indices $U = \{1, \ldots, \mathsf{m}\} \cup C$. For the abstract indices for which we have non-empty witness sets, namely the indices 1, 2, cnt_1, and cnt_7, the cells of $\widehat{\mathbf{cr}}$, $\widehat{\mathbf{Msg}}$, and $\widehat{\mathbf{Rcv}}$ indexed by them store sets of values which are non-empty.

We now determine the sets of processes witnessed by different abstract indices in the state $\widetilde{\sigma}'''$, defined in Definition 3.11.

$$
\begin{aligned}
&witness_{\mathsf{n}}(\widetilde{\sigma}''', 1) = \{1\} && witness_{\mathsf{n}}(\widetilde{\sigma}''', cnt_1) = \{4\}\\
&witness_{\mathsf{n}}(\widetilde{\sigma}''', 2) = \{2\} && witness_{\mathsf{n}}(\widetilde{\sigma}''', cnt_2) = \{3, 6\}\\
& && witness_{\mathsf{n}}(\widetilde{\sigma}''', cnt_7) = \{5\}
\end{aligned}
$$

The result of applying the abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$ to the state $\widetilde{\sigma}''' \in \widetilde{S}(\mathsf{n})$, that is, the state $\widehat{\sigma}''' = \widehat{\alpha}_{\mathsf{n}}(\widetilde{\sigma}''')$ is obtained in a similar way, and is given below.

$$
\widehat{\sigma}''' :\ \widehat{\mathbf{best}} = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\quad \widehat{\mathbf{dec}} = \begin{bmatrix} \bot \\ \bot \end{bmatrix},\quad \widehat{\mathbf{fld}} = \begin{bmatrix} \bot \\ \bot \end{bmatrix}
$$

$$
\widehat{\mathbf{active}} = \begin{array}{c} \\ cnt_1 \\ cnt_2 \\ \vdots \\ cnt_7 \\ \vdots \\ cnt_{12} \end{array} \begin{bmatrix} \\ \mathsf{many} \\ \mathsf{many} \\ \vdots \\ \mathsf{many} \\ \vdots \\ 0 \end{bmatrix}
$$

$$
\widehat{\mathbf{Msg}} = \begin{array}{c} \\ 1 \\ 2 \\ cnt_1 \\ cnt_2 \\ \vdots \\ cnt_7 \\ \vdots \\ cnt_{12} \end{array} \begin{bmatrix} 1 & 2 & cnt_1 & cnt_2 & \cdots & cnt_7 & \cdots & cnt_{12} \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \{\bot\} & \{\bot\} & \{\bot\} & \{\bot\} & \cdots & \{\bot\} & \cdots & \emptyset \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \emptyset & \emptyset & \emptyset & \emptyset & \cdots & \emptyset & \cdots & \emptyset \end{bmatrix}
$$

$$
pr_{r>t+1} \qquad \widehat{\mathbf{cr}} = \begin{array}{c} \\ 1 \\ 2 \\ cnt_1 \\ cnt_2 \\ \vdots \\ cnt_7 \\ \vdots \\ cnt_{12} \end{array} \begin{bmatrix} \\ \{\bot\} \\ \{\bot\} \\ \{\bot\} \\ \{\top\} \\ \vdots \\ \{\bot\} \\ \vdots \\ \emptyset \end{bmatrix}
$$

$$
\widehat{\mathbf{Rcv}} = \begin{array}{c} \\ 1 \\ 2 \\ cnt_1 \\ cnt_2 \\ \vdots \\ cnt_7 \\ \vdots \\ cnt_{12} \end{array} \begin{bmatrix} 1 & 2 & cnt_1 & cnt_2 & \cdots & cnt_7 & \cdots & cnt_{12} \\ \{\top\} & \{\top\} & \{\top\} & \{\bot,\top\} & \cdots & \{\top\} & \cdots & \emptyset \\ \{\top\} & \{\top\} & \{\top\} & \{\bot\} & \cdots & \{\top\} & \cdots & \emptyset \\ \{\top\} & \{\top\} & \{\top\} & \{\bot,\top\} & \cdots & \{\top\} & \cdots & \emptyset \\ \{\top\} & \{\top\} & \{\top\} & \{\bot\} & \cdots & \{\top\} & \cdots & \emptyset \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \{\top\} & \{\top\} & \{\top\} & \{\bot\} & \cdots & \{\top\} & \cdots & \emptyset \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \emptyset & \emptyset & \emptyset & \emptyset & \cdots & \emptyset & \cdots & \emptyset \end{bmatrix}
$$

Note that as processes 3 and 6 update their control state from cnt_1 to cnt_2 in the transition from $\widetilde{\sigma}$ to $\widetilde{\sigma}'''$, in the state $\widehat{\sigma}'''$ there are many processes in control state cnt_2, i.e., $\widehat{\sigma}'''.\widehat{\mathbf{active}}[cnt_2] = \mathsf{many}$. Moreover, there are many processes in control state cnt_1 as well, since process 4 updates its control state from cnt_7 to cnt_1, i.e., $\widehat{\sigma}'''.\widehat{\mathbf{active}}[cnt_1] = \mathsf{many}$. The variables that are indexed by control states, namely $\widehat{\mathbf{cr}}, \widehat{\mathbf{Msg}}$, and $\widehat{\mathbf{Rcv}}$, the cells indexed by the abstract indices witnessing processes, that is, indices 1, 2, cnt_1, cnt_2, and cnt_7, store sets of values which are non-empty.

Thus we see that all variables in the states $\widehat{\sigma}, \widehat{\sigma}''' \in \widehat{S}$ have fixed size, which does not depend on the value of n, but rather on the small number m and the number of control states $|C|$, which are both fixed and finite. □

Given the system $\widetilde{\mathsf{STS}}(\mathsf{n})$ and the set $\widehat{S}$ of states, we define $\widehat{\mathsf{STS}}$ as the *overapproximation* of $\widetilde{\mathsf{STS}}(\mathsf{n})$ induced by the abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$. To be able to state the soundness of the abstraction mapping $\widehat{\alpha}_{\mathsf{n}}$, which follows from the overapproximation, we need to define

the semantics of the indexed-LTL formulas from the fragment $\mathcal{F}^{\mathsf{m}}(k)$ in states $\widehat{\sigma} \in \widehat{S}$. This is necessary, as until now, we expressed indexed-LTL formulas from the fragment $\mathcal{F}^{\mathsf{m}}(k)$ where we fixed the set $\mathsf{Ind} = \{1, \ldots, \mathsf{n}\}$ as the set of indices, where $\mathsf{n} \in \mathbb{N}$ is the value of the parameter n. As we have abstracted the parameter n in the system $\widehat{\mathsf{STS}}$, and replaced the indices of the array variables with the set U of abstract indices, to evaluate indexed-LTL formulas from the fragment $\mathcal{F}^{\mathsf{m}}(k)$ in abstract states $\widehat{\sigma} \in \widehat{S}$, we fix:

- $\mathsf{Ind} = U$ as the set of indices,
- $\mathsf{Vars} = \mathsf{cntl}(\widehat{V}_{\mathsf{sys}}) \cup \{\widehat{\mathbf{cr}}\}$ as the set of variables,
- $\mathsf{Prop} = \mathsf{Pred}$ as the set of Boolean propositions.

By Definition 3.3, for a formula $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$, we have that the universal quantifiers range over the set $\{1, \ldots, \mathsf{m}\}$ and the existential quantifiers range over the set $\mathsf{Ind} = U$ of abstract indices.

The semantics of the atomic propositions $\mathbf{z}_i = v$ in $\widehat{\sigma}$ is defined as follows: $\widehat{\sigma} \models \mathbf{z}_i = v$ iff:

1. either i is bound by a universal quantifier and $i \in \{1, \ldots, \mathsf{m}\}$ or i is bound by an existential quantifier and $i \in U$, and
2. either:
   - $\mathbf{z} \neq \widehat{\mathbf{cr}}$ and $i \in \{1, \ldots, \mathsf{m}\}$ and $\widehat{\sigma}.\widehat{\mathbf{z}}[i] = v$, or
   - $\mathbf{z} \neq \widehat{\mathbf{cr}}$ and $i \in C$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[i] = \mathsf{many}$ and $i.z = v$, or
   - $\mathbf{z} = \widehat{\mathbf{cr}}$ and $v \in \widehat{\sigma}.\widehat{\mathbf{cr}}[i]$.

For the Boolean propositions from the set Pred we have $\widehat{\sigma} \models pr$ iff $\widehat{\sigma}.pr = \top$.

Proposition 3.6 (Soundness of $\widehat{\alpha}_{\mathsf{n}}$)**.** *Let* $\widehat{\mathsf{STS}}$ *be the overapproximation of* $\widetilde{\mathsf{STS}}(\mathsf{n})$ *induced by the abstraction mapping* $\widehat{\alpha}_{\mathsf{n}}$*, for* $\mathsf{n} \in \mathbb{N}$*. Let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number such that* $\mathsf{m} \leq \mathsf{n}$*. For every formula* $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$*, we have:*

$$\widehat{\mathsf{STS}} \models \phi^{\mathsf{m}} \quad \textit{implies} \quad \widetilde{\mathsf{STS}}(\mathsf{n}) \models \phi^{\mathsf{m}} \qquad \square$$

Proof. The semantics of the indexed-LTL formulas from the fragment $\mathcal{F}^{\mathsf{m}}(k)$ differs in $\widetilde{\mathsf{STS}}(\mathsf{n})$ and $\widehat{\mathsf{STS}}$. We show that for every $\mathsf{n} \in \mathbb{N}$, every formula $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$, and every initial abstract state $\widehat{\sigma}_0 \in \widehat{S}_0$, we have $\widehat{\sigma}_0 \models \phi^{\mathsf{m}}$ implies $\widetilde{\sigma}_0 \models \phi^{\mathsf{m}}$, where $\widehat{\sigma}_0 = \widehat{\alpha}_{\mathsf{n}}(\widetilde{\sigma}_0)$ and $\widetilde{\sigma}_0 \in \widetilde{S}_0(\mathsf{n})$.

When evaluating $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$ in states $\widehat{\sigma} \in \widehat{S}$, we fix $\mathsf{Ind} = U$, while in $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$, we fix $\mathsf{Ind} = \{1, \ldots, \mathsf{n}\}$. Since the set Ind is used as a range of the existential quantifiers, and since existential quantifiers only occur in front of the atomic propositions $\exists i : \mathsf{Ind}.\, \mathbf{z}_i = v$

(recall Definition 3.3), the difference between evaluating formulas $\phi^{\mathsf{m}} \in \mathcal{F}^{\mathsf{m}}(k)$ in states $\widehat{\sigma} \in \widehat{S}$ and $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ is in evaluating the formulas of the form $\exists i : \mathsf{Ind}.\mathbf{z}_i = v$. Thus, in the following, we show that $\widehat{\sigma} \models \exists i : \mathsf{Ind}.\,\mathbf{z}_i = v$ implies $\widetilde{\sigma} \models \exists i : \mathsf{Ind}.\,\mathbf{z}_i = v$. The semantics of the atomic propositions $\mathbf{z}_i = v$ when the index i is bound by a universal quantifier, as well as the semantics of the atomic propositions $pr \in \mathsf{Prop}$ is the same in both states $\widehat{\sigma} \in \widehat{S}$ and $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$. Further, the semantics of the logical connectives, quantifiers, and temporal operators is standard.

Let $\widehat{\sigma} \in \widehat{S}$ be an arbitrary state in $\widehat{\mathsf{STS}}$, and let $\widetilde{\sigma} \in \widetilde{S}(\mathsf{n})$ be a state in $\widetilde{\mathsf{STS}}\,(\mathsf{n})$ such that $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}}(\widetilde{\sigma})$. Suppose $\widehat{\sigma} \models \exists i : \mathsf{Ind}.\,\mathbf{z}_i = v$. We consider the following cases:

1. $\mathbf{z} \neq \widehat{\mathbf{cr}}$ and $i \in \{1, \ldots, \mathsf{m}\}$. By the semantics of the atomic propositions in the states $\widehat{S}$, we have $\widehat{\sigma}.\widehat{\mathbf{z}}[i] = v$. By Definition 3.14, we have $\widehat{\sigma}.\widehat{\mathbf{z}}[i] = \widetilde{\sigma}.\widetilde{\mathbf{z}}[i]$, hence $\widetilde{\sigma}.\widetilde{\mathbf{z}}[i] = v$ and $\widetilde{\sigma} \models \exists i : \mathsf{Ind}.\,\mathbf{z}_i = v$.

2. $\mathbf{z} \neq \widehat{\mathbf{cr}}$ and $i \in C$. By the semantics of the atomic propositions in the states $\widehat{S}$, we have $\widehat{\sigma}.\widehat{\mathbf{active}}[i] = \mathsf{many}$ and $i.z = v$. By Definition 3.14 and $\widehat{\sigma}.\widehat{\mathbf{active}}[i] = \mathsf{many}$ we have that there exists a process j, with $\mathsf{m} < j \leq \mathsf{n}$, such that $j \in \mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, i)$. By the definition of the mapping $\mathit{witness}_{\mathsf{n}}$, we get $\widetilde{\sigma}.\mathit{control}_j = i$, and from $i.z = v$, we obtain $\widetilde{\sigma}.\mathit{control}_j.z = \widetilde{\sigma}.\widetilde{\mathbf{z}}[j] = v$. Thus, $\widetilde{\sigma} \models \exists i : \mathsf{Ind}.\,\mathbf{z}_i = v$.

3. $\mathbf{z} = \widehat{\mathbf{cr}}$. By the semantics of the atomic propositions in the states $\widehat{S}$, we have $v \in \widehat{\sigma}.\widehat{\mathbf{cr}}[i]$. By Definition 3.14, $\widehat{\sigma}.\widehat{\mathbf{cr}}[i] = \bigcup\{\widetilde{\sigma}.\widetilde{\mathbf{cr}}[j] \mid j \in \mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, i)\}$. By the definition of the mapping $\mathit{witness}_{\mathsf{n}}$ (Definition 3.11), we have that there exists a process $j \in \mathit{witness}_{\mathsf{n}}(\widetilde{\sigma}, i)$, with $1 \leq j \leq \mathsf{n}$, such that $\widetilde{\sigma}.\widetilde{\mathbf{cr}}[i] = v$. From this we get $\widetilde{\sigma} \models \exists i : \mathsf{Ind}.\,\mathbf{z}_i = v$.

Hence we obtain $\widehat{\sigma} \models \exists i : \mathsf{Ind}.\,\mathbf{z}_i = v$ implies $\widetilde{\sigma} \models \exists i : \mathsf{Ind}.\,\mathbf{z}_i = v$, where $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}}(\widetilde{\sigma})$. □

The overall soundness of the abstraction-based approach is a consequence of Propositions 3.2, 3.3, 3.5, and 3.6, stated in the theorem below.

Theorem 3.1 (Soundness)**.** *Let* $\widehat{\mathsf{STS}}$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ *induced by the abstraction mapping* $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$*, for* $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition. Let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number such that* $\mathsf{m} \leq \mathsf{n}$*. For every formula* $\phi \in \mathcal{F}(k)$*, we have:*

$$\widehat{\mathsf{STS}} \models \mathsf{sym}(\phi, \mathsf{m}) \quad \textit{implies} \quad \mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \models \phi \qquad \square$$

Additionally, as a consequence of Propositions 3.2, 3.4, 3.5, and 3.6, we obtain the following soundness theorem for our crash-tolerant benchmarks.

Theorem 3.2 (Soundness for crash-tolerant benchmarks)**.** *Let* $\widehat{\mathsf{STS}}$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ *induced by the abstraction mapping* $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$*, for* $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition. Suppose that* $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ *models a synchronous*

fault-tolerant distributed algorithm that tolerates crash faults, i.e., that $\mathsf{n} > \mathsf{t} \geq \mathsf{f}$. *Let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number such that* $\mathsf{m} \leq \mathsf{n}$. *For every formula* $\phi \in \mathcal{F}(k)$, *we have:*

$$\widehat{\mathsf{STS}} \models \phi_{VC} \rightarrow \mathsf{sym}(\phi, \mathsf{m}) \quad \textit{implies} \quad \mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \models \phi \qquad \square$$

3.5 Constructive Definition of the Abstract System

So far, in this chapter, we defined the abstract system $\widehat{\mathsf{STS}} = \langle \widehat{S}, \widehat{S}_0, \widehat{T} \rangle$ as the overapproximation of every concrete finite-state system from the infinite family $\{\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f}) \mid \mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ satisfy the resilience condition$\}$. This definition is not constructive. In order to obtain all possible abstract initial states and abstract transitions, the abstraction mapping $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$ should be applied to all initial states and all transitions in every concrete system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$. In this section, we will give a definition of an abstract system $\widehat{\mathsf{STS}}^* = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \widehat{\mathcal{T}} \rangle$, where the set $\widehat{\mathcal{S}}$ of abstract states is equal to the set $\widehat{S}$ of abstract states, and where we constructively define the set $\widehat{\mathcal{S}}_0$ of initial states and the transition relation $\widehat{\mathcal{T}}$. We will also show that the abstract system $\widehat{\mathsf{STS}}^*$ *simulates* the abstract system $\widehat{\mathsf{STS}}$, and thus we can use the abstract system $\widehat{\mathsf{STS}}^*$ to check properties of the abstract system $\widehat{\mathsf{STS}}$ (and, by Theorem 3.1, properties of the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, for $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition).

The remainder of this section is organized as follows. In Section 3.5.1, we define the set $\widehat{\mathcal{S}}$ of states and the set $\widehat{\mathcal{S}}_0$ of initial states of the system $\widehat{\mathsf{STS}}^*$. Section 3.5.2 introduces abstract versions of the guarded assignments, defined in Section 2.1.3 on page 43. The abstract guarded assignments will be used to constructively define the abstract transition relation $\widehat{\mathcal{T}}$ in Section 3.5.3. Similarly to the transition relation $T(\mathsf{n}, \mathsf{t}, \mathsf{f})$ defined in Definition 2.18 on page 50, we will define the abstract transition relation $\widehat{\mathcal{T}}$ as a composition of three abstract transition relations, $\widehat{Env}$, $\widehat{Snd}$, and $\widehat{Upd}$, which are abstract versions of the transition relations $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$, $Snd(\mathsf{n}, \mathsf{t}, \mathsf{f})$, and $Upd(\mathsf{n}, \mathsf{t}, \mathsf{f})$, respectively, used to build the transition relation $T(\mathsf{n}, \mathsf{t}, \mathsf{f})$. Finally, in Section 3.5.4, we show the existence of a simulation relation between the systems $\widehat{\mathsf{STS}}$ and $\widehat{\mathsf{STS}}^*$.

3.5.1 Abstract States

The abstract system $\widehat{\mathsf{STS}}^* = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \widehat{\mathcal{T}} \rangle$ has the same system variables $\widehat{V}_{\mathsf{sys}}$ as the abstract system $\widehat{\mathsf{STS}} = \langle \widehat{S}, \widehat{S}_0, \widehat{T} \rangle$, and thus, both systems have the same set of states, i.e., $\widehat{\mathcal{S}} = \widehat{S}$. Recall that the variables $\widehat{V}_{\mathsf{sys}}$ of the abstract system were defined based on the variables V_{sys} of the concrete system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, for $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, which in turn were defined based on the process variables V_{proc} and environment variables V_{env} (defined in Chapter 2). More precisely, the system control and neighborhood variables, $\mathsf{cntl}(V_{\mathsf{sys}})$ and $\mathsf{nbhd}(V_{\mathsf{sys}})$, are defined w.r.t. the process control and neighborhood variables $\mathsf{cntl}(V_{\mathsf{proc}})$ and $\mathsf{nbhd}(V_{\mathsf{proc}})$, respectively.

A similar correspondence between process and system variables exists in the abstract system $\widehat{\mathsf{STS}}^*$ as well: we distinguish between *abstract process control variables* $\widehat{x} \in \mathsf{cntl}(\widehat{V}_{\mathsf{proc}})$,

whose values define the control states from C, and *abstract process neighborhood variables* $\widehat{\mathbf{y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{proc}})$, which are one-dimensional array variables of size $|U|$, ranging over $(2^{D_{\mathbf{y}}})^{|U|}$, where $\mathbf{y} \in \mathsf{nbhd}(V_{\mathsf{proc}})$ is the concrete process neighborhood variable corresponding to $\widehat{\mathbf{y}}$, and $D_{\mathbf{y}}$ is the set of values stored in $\mathbf{y}$. Given the set of abstract process variables $\widehat{V}_{\mathsf{proc}} = \mathsf{cntl}(\widehat{V}_{\mathsf{proc}}) \cup \mathsf{nbhd}(\widehat{V}_{\mathsf{proc}})$, we define the set $\widehat{L} = \prod_{\widehat{x}} D_x \times \prod_{\widehat{\mathbf{y}}} (2^{D_{\mathbf{y}}})^{|U|}$ of *abstract local states* as the set of valuations of the abstract process variables $\widehat{V}_{\mathsf{proc}}$.

We now proceed by adapting several other notions that we used throughout this chapter and Chapter 2. First, given an abstract state $\widehat{\sigma} \in \widehat{\mathcal{S}}$ and an abstract index $u \in U$, we say that u *witnesses a process* in the state $\widehat{\sigma}$ if $u \in \{1, \dots \mathsf{m}\}$ or if $u \in C$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$. Next, we adapt the notions *control*, *row* and *local*. For an abstract state $\widehat{\sigma} \in \widehat{\mathcal{S}}$ and an abstract index $u \in U$, we denote by:

- $\widehat{\sigma}.control_u$ the tuple
  - $\langle \widehat{\sigma}.\widehat{\mathbf{x}}_1[u], \dots, \widehat{\sigma}.\widehat{\mathbf{x}}_{cv}[u] \rangle \in C$, where $cv = |\mathsf{cntl}(\widehat{V}_{\mathsf{sys}})|$, if $u \in \{1, \dots, \mathsf{m}\}$
  - u, if $u \in C$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$,
- $\widehat{\sigma}.row_u^{\widehat{\mathbf{Y}}}$ the tuple $\langle \widehat{\sigma}.\widehat{\mathbf{Y}}[u, v_1], \dots, \widehat{\sigma}.\widehat{\mathbf{Y}}[u, v_{|U|}] \rangle \in (2^{D_{\mathbf{y}}})^{|U|}$,
- $\widehat{\sigma}.local_u$ the tuple $\langle \widehat{\sigma}.control_u, \widehat{\sigma}.row_u^{\widehat{\mathbf{Y}}_1} \dots, \widehat{\sigma}.row_u^{\widehat{\mathbf{Y}}_{nv}} \rangle \in \widehat{L}$, where $nv = |\mathsf{nbhd}(\widehat{V}_{\mathsf{sys}})|$.

In the following, we give a constructive definition of the set $\widehat{\mathcal{S}}_0$ of abstract initial states, which constrains the values assigned to the abstract system variables $\widehat{V}_{\mathsf{sys}}$.

Definition 3.15 (Abstract initial states $\widehat{\mathcal{S}}_0$)**.** Let $\widehat{\mathcal{S}}_0 \subseteq \widehat{\mathcal{S}}$ denote the set of *abstract initial states*. An abstract state $\widehat{\sigma} \in \widehat{\mathcal{S}}$ is *initial*, that is $\widehat{\sigma} \in \widehat{\mathcal{S}}_0$ if:

1. $\widehat{\sigma}.control_u \in C_0$, where $u \in \{1 \dots \mathsf{m}\}$ and C_0 is the set of initial control states,
2. $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = 0$, where $u \in C \setminus C_0$,
3. $\widehat{\sigma}.\widehat{\mathbf{Y}}[u, v] = \begin{cases} \{\bot\} & \text{if } u, v \in U \text{ witness a process in } \widehat{\sigma} \\ \emptyset & \text{otherwise} \end{cases}$, where $\widehat{\mathbf{Y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{sys}})$,
4. $\widehat{\sigma}.\widehat{\mathbf{cr}}[u] = \begin{cases} \{\bot\} & \text{if } u \in U \text{ witnesses a process in } \widehat{\sigma} \\ \emptyset & \text{otherwise} \end{cases}$,
5. $\widehat{\sigma}.\widehat{\mathbf{Rcv}}[u, v] = \begin{cases} \{\bot\} & \text{if } u, v \in U \text{ witness a process in } \widehat{\sigma} \\ \emptyset & \text{otherwise} \end{cases}$,
6. $\widehat{\sigma}.pr = \bot$, where $pr \in \mathsf{Pred}$. □

The above definition constrains the values assigned to the abstract variables $\widehat{V}_{\mathsf{sys}}$. These constraints ensure that concrete initial states $S_0(\mathsf{n},\mathsf{t},\mathsf{f})$, defined in Definition 2.17 on page 49, are mapped by the abstraction mapping $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$ to initial states which satisfy the constraints in Definition 3.15. We formalize this using the lemma below.

Lemma 3.1. *Let* $\widehat{\mathsf{STS}} = \langle \widehat{S}, \widehat{S}_0, \widehat{T} \rangle$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *induced by the abstraction mapping* $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$*, for* $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition. Let* $\widehat{\mathcal{S}}_0 \subseteq \widehat{\mathcal{S}}$ *be the set of abstract initial states. For every abstract state* $\widehat{\sigma} \in \widehat{S}$*, we have* $\widehat{\sigma} \in \widehat{S}_0$ *implies* $\widehat{\sigma} \in \widehat{\mathcal{S}}_0$*.*

Proof. Follows from the definition of the concrete initial states $S_0(\mathsf{n},\mathsf{t},\mathsf{f})$ (Definition 2.17 on page 49), the definitions of the abstraction mappings $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}, \widetilde{\alpha}_{\mathsf{n}}$, and $\widehat{\alpha}_{\mathsf{n}}$, (Definitions 3.6, 3.12, and 3.14, respectively), and the definition of the abstract initial states $\widehat{\mathcal{S}}_0$ (Definition 3.15). □

Example 3.10. Recall Example 3.9, where we depicted the abstract initial state $\widehat{\sigma} \in \widehat{S}_0$ of the abstract system $\widehat{\mathsf{STS}}$ for the algorithm FloodMin for $k = 1$. Recall that for the algorithm FloodMin for $k = 1$ we have $\mathsf{m} = 2$, and the set $C_0 \subseteq C$ of initial control states contains the two control states $cnt_1 = \langle 0, \bot, \bot \rangle$ and $cnt_7 = \langle 1, \bot, \bot \rangle$.

Consider the state $\widehat{\sigma}$, depicted on page 81. We show that the state $\widehat{\sigma}$ satisfies the conditions of Definition 3.15, and thus we have $\widehat{\sigma} \in \widehat{\mathcal{S}}_0$.

1. Both processes 1 and 2 are in control state $cnt_7 = \langle 1, \bot, \bot \rangle$. Thus $\widehat{\sigma}.control_u \in C_0$, for $u \in \{1, \ldots, \mathsf{m}\}$, i.e., condition 1 from Definition 3.15 holds;
2. In the state $\widehat{\sigma}$, we have $\widehat{\sigma}.\widehat{\mathbf{active}}[cnt_1] = \widehat{\sigma}.\widehat{\mathbf{active}}[cnt_7] = \mathsf{many}$, and $\widehat{\sigma}.\widehat{\mathbf{active}}[cnt] = 0$, for $cnt \notin \{cnt_1, cnt_7\}$. As $C_0 = \{cnt_1, cnt_7\}$, we have that $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = 0$, for $u \in C \setminus C_0$, hence condition 2 from Definition 3.15 holds;
3. The abstract indices $1, 2, cnt_1, cnt_7 \in U$ witness a process in the state $\widehat{\sigma}$. For the variables $\widehat{\mathbf{Msg}}, \widehat{\mathbf{cr}}$, and $\widehat{\mathbf{Rcv}}$, the cells indexed by these indices store the value $\{\bot\}$, while the remaining ones store the value $\emptyset$. Hence, conditions 3, 4, and 5 from Definition 3.15 hold.

From this, we can conclude that $\widehat{\sigma} \in \widehat{\mathcal{S}}_0$. □

3.5.2 Abstract Guarded Assignments

To be able to give a constructive definition of the abstract transition relation $\widehat{\mathcal{T}}$, we need to define how the control states are updated in the abstract system $\widehat{\mathsf{STS}}^*$. To this end, we define two abstract versions of the parameterized process function $update_{n,t,r}$: one that we will use to update the control states of the fixed m processes, and which we will denote by $\widehat{update}^{\mathsf{m}}_{\mathsf{Pred}}$, and a second one, denoted by $\widehat{update}^{C}_{\mathsf{Pred}}$, which we will use to update the

control states of the processes for which we keep information about their control state in the abstract variable $\widehat{\mathbf{active}}$. As the function $update_{n,t,r}$ was characterized using a set of guarded assignments (recall Definition 2.12 on page 45), in the following, we define abstract guarded assignments, which we will use to characterize the functions $\widehat{update}^{\mathsf{m}}_{\mathsf{Pred}}$ and $\widehat{update}^{C}_{\mathsf{Pred}}$.

The syntax of the *abstract guard propositions*, which are an abstract version of the guard propositions from Definition 2.10 on page 44 is given below:

empty	$\top$	
control	$\widehat{x} = \mathsf{v}$	where $\widehat{x} \in \mathsf{cntl}(\widehat{V}_{\mathsf{proc}})$ and $\mathsf{v} \in D_x$
neighborhood	$\exists u \bigwedge_{\widehat{\psi} \in \widehat{\Psi}(u)} \widehat{\psi}$	where $\widehat{\Psi}(u) \subseteq \{in(\mathsf{v}, \widehat{\mathbf{y}}[u]) \mid \widehat{\mathbf{y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{proc}}), \mathsf{v} \in D_{\mathbf{y}}\} \cup \{not\_in(\mathsf{v}, \widehat{\mathbf{y}}[u]) \mid \widehat{\mathbf{y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{proc}}), \mathsf{v} \in D_{\mathbf{y}}\}$
termination	pr	where pr abstracts the termination guard proposition $r \geq \phi(n,t)$.

Example 3.11. Recall Example 2.4 on page 45, where we defined the set G of guarded assignments that characterizes the control state update function $update_{n,t,r}$ for the algorithm FloodMin for $k = 1$. We listed four atomic guard propositions, which we used in the guards of the guarded assignments from the set G. We now show the abstract versions of these atomic guard propositions:

- $pr_{r>t+1}$, which is the abstract version of the termination guard proposition $r > t+1$,
- $\widehat{best} = 0$, which is the abstract version of the control guard proposition $best = 0$,
- $\widehat{best} = 1$, which is the abstract version of the control guard proposition $best = 1$,
- $\exists u\, in(0, \widehat{\mathbf{msg}}[u])$, which is the abstract version of the neighborhood guard proposition $\exists j\, \mathbf{msg}[j] = 0$. □

The semantics of the abstract guard propositions is defined over tuples $(\widehat{local}, \mathsf{Pred})$, where $\widehat{local} \in \widehat{L}$ is an abstract local state, and Pred is a set of predicates, as follows:

$(\widehat{local}, \mathsf{Pred}) \models \top$	holds true
$(\widehat{local}, \mathsf{Pred}) \models \widehat{x} = \mathsf{v}$	if $\widehat{local}.\widehat{x} = \mathsf{v}$
$(\widehat{local}, \mathsf{Pred}) \models \exists u \bigwedge_{\widehat{\psi} \in \widehat{\Psi}(u)} \widehat{\psi}$	if there is $u \in U$ such that for every $\widehat{\psi} \in \widehat{\Psi}(u)$ we have either $\mathsf{v} \in \widehat{local}.\widehat{\mathbf{y}}[u]$, if $\widehat{\psi} \equiv in(\mathsf{v}, \widehat{\mathbf{y}}[u])$, or $\mathsf{v} \notin \widehat{local}.\widehat{\mathbf{y}}[u]$, if $\widehat{\psi} \equiv not\_in(\mathsf{v}, \widehat{\mathbf{y}}[u])$.
$(\widehat{local}, \mathsf{Pred}) \models pr$	if $pr \in \mathsf{Pred}$ and $pr = \top$.

An *abstract guarded assignment* is an expression of the form $\widehat{\varphi} \to \widehat{assig}$, where $\widehat{\varphi}$ is an *abstract guard* and $\widehat{assig}$ is an *abstract assignment*, defined analogously to Definition 2.10. Given a set Pred of predicates, a control state *control* $\in C$ is the *result of applying*

a guarded assignment $\widehat{\varphi} \to \widehat{assig}$ to an abstract local state $\widehat{local} \in L(n)$ if for every $\widehat{x} \in \mathsf{cntl}(\widehat{V}_{\mathsf{proc}})$, we have:

$$control.\widehat{x} = \begin{cases} \widehat{assig}(\widehat{x}) & \text{if } (\widehat{local}, \mathsf{Pred}) \models \widehat{\varphi} \text{ and } \widehat{assig}(\widehat{x}) \text{ is defined} \\ \widehat{local}.\widehat{x} & \text{otherwise} \end{cases}$$

To characterize the function $\widehat{update}^{\mathsf{m}}_{\mathsf{Pred}}$, which we will apply to update the control states of the fixed m processes, we use a finite set $\widehat{G}^{\mathsf{m}}$ of abstract guarded assignments with pairwise mutually exclusive guards (as in the concrete case).

Definition 3.16 (Characterization of $\widehat{update}^{\mathsf{m}}_{\mathsf{Pred}}$)**.** Let Pred be a set of predicates and $\widehat{G}^{\mathsf{m}}$ a set of abstract guarded assignments with pairwise mutually disjoint guards. The function $\widehat{update}^{\mathsf{m}}_{\mathsf{Pred}} : \widehat{L} \to C$ maps an abstract local state $\widehat{local} \in \widehat{L}$ to a control state $control \in C$, such that $\widehat{update}^{\mathsf{m}}_{\mathsf{Pred}}(\widehat{local}) = control$, iff there exists an abstract guarded assignment $\widehat{\varphi} \to \widehat{assig} \in \widehat{G}^{\mathsf{m}}$, such that $control$ is the result of applying $\widehat{\varphi} \to \widehat{assig}$ to the local state $\widehat{local}$. □

Example 3.12. We now define the set $\widehat{G}^{\mathsf{m}}$ for the algorithm $\mathsf{FloodMin}$ for $k = 1$, which is used to characterize the abstract update function $\widehat{update}^{\mathsf{m}}_{\mathsf{Pred}}$, that updates the control states of the fixed m processes. Since the control stats of the fixed m processes are kept as in the concrete system, the set $\widehat{G}^{\mathsf{m}}$ contains the abstract versions of the guarded assignments presented in Example 2.4 on page 45:

$$\widehat{g}^{\mathsf{m}}_1 \ : \ \neg pr_{r>t+1} \wedge (\widehat{best} = 0) \ \to \ \widehat{best} := 0$$

$$\widehat{g}^{\mathsf{m}}_2 \ : \ \neg pr_{r>t+1} \wedge (\widehat{best} = 1) \wedge (\exists u \ in(0, \widehat{\mathbf{msg}}[u])) \ \to \ \widehat{best} := 0$$

$$\widehat{g}^{\mathsf{m}}_3 \ : \ \neg pr_{r>t+1} \wedge (\widehat{best} = 1) \wedge \neg(\exists u \ in(0, \widehat{\mathbf{msg}}[u])) \ \to \ \widehat{best} := 1$$

$$\widehat{g}^{\mathsf{m}}_4 \ : \ pr_{r>t+1} \wedge (\widehat{best} = 0) \ \to \ \widehat{dec} := 0$$

$$\widehat{g}^{\mathsf{m}}_5 \ : \ pr_{r>t+1} \wedge (\widehat{best} = 1) \ \to \ \widehat{dec} := 1$$

Observe that these abstract guarded assignments are obtained from the concrete guarded assignments in Example 2.4, by directly replacing the guard propositions with their abstract counterparts, which we presented in Example 3.11. □

To characterize the function $\widehat{update}^{C}_{\mathsf{Pred}}$, which we will apply to update the control states of processes witnessed by $u \in C$, we use a set $\widehat{G}^{C}$ of abstract guarded assignments, where the guards are not pairwise mutually exclusive.

Definition 3.17 (Characterization of $\widehat{update}^{C}_{\mathsf{Pred}}$)**.** Let Pred be a set of predicates and $\widehat{G}^{C}$ a set of abstract guarded assignments. The function $\widehat{update}^{C}_{\mathsf{Pred}} : \widehat{L} \to 2^{C}$ maps an abstract local state $\widehat{local} \in \widehat{L}$ to a set of control states such that $control \in \widehat{update}^{C}_{\mathsf{Pred}}(\widehat{local})$, for $control \in C$, iff there exists an abstract guarded assignment $\widehat{\varphi} \to \widehat{assig} \in \widehat{G}^{C}$, such that $control$ is the result of applying $\widehat{\varphi} \to \widehat{assig}$ to $\widehat{local}$. □

To define which guarded assignments are in the set $\widehat{G}^{C}$ of abstract guarded assignments, we start by setting $\widehat{G}^{C} = \widehat{G}^{\mathsf{m}}$. Then, for every control variable $\widehat{x} \in \mathsf{cntl}(\widehat{V}_{\mathsf{proc}})$ and every abstract guarded assignment $\widehat{\varphi} \to \widehat{assig} \in \widehat{G}^{\mathsf{m}}$, we proceed as follows. If the control variable $\widehat{x}$ occurs in the guard $\widehat{\varphi}$ in a control guard proposition $\widehat{x} = \mathsf{v}$ and if $\widehat{assig}(\widehat{x})$ is defined, then if the value $\widehat{assig}(\widehat{x})$ assigned to $\widehat{x}$ by the assignment $\widehat{assig}$ is different than the value v, we add the following guarded assignment to the set $\widehat{G}^{C}$:

$$\widehat{\varphi} \to \widehat{assig'} \quad \text{where} \quad \widehat{assig'}(\widehat{z}) = \begin{cases} \mathsf{v} & \text{if } \widehat{z} = \widehat{x} \\ \widehat{assig}(\widehat{z}) & \text{otherwise} \end{cases}$$

This captures that two different processes are witnessed by the same control state in $\widehat{\sigma}$, depending on the neighborhood and environment variables, it can happen that they update to two different control states. This is why the function $\widehat{update}^{C}_{\mathsf{Pred}}$ returns a set of control states.

Example 3.13. The set $\widehat{G}^{C}$ for the algorithm $\mathsf{FloodMin}$ for $k = 1$ contains the following guarded assignments:

$$\widehat{g}_1^C \ : \ \neg pr_{r>t+1} \wedge (\widehat{best} = 0) \ \to \ \widehat{best} := 0$$

$$\widehat{g}_2^C \ : \ \neg pr_{r>t+1} \wedge (\widehat{best} = 1) \wedge (\exists u \ in(0, \widehat{\mathbf{msg}}[u])) \ \to \ \widehat{best} := 0$$

$$\widehat{g}_3^C \ : \ \neg pr_{r>t+1} \wedge (\widehat{best} = 1) \wedge (\exists u \ in(0, \widehat{\mathbf{msg}}[u])) \ \to \ \widehat{best} := 1$$

$$\widehat{g}_4^C \ : \ \neg pr_{r>t+1} \wedge (\widehat{best} = 1) \wedge \neg(\exists u \ in(0, \widehat{\mathbf{msg}}[u])) \ \to \ \widehat{best} := 1$$

$$\widehat{g}_5^C \ : \ pr_{r>t+1} \wedge (\widehat{best} = 0) \ \to \ \widehat{dec} := 0$$

$$\widehat{g}_6^C \ : \ pr_{r>t+1} \wedge (\widehat{best} = 1) \ \to \ \widehat{dec} := 1$$

Observe that the set $\widehat{G}^{C}$ differs from the set $\widehat{G}^{\mathsf{m}}$ from Example 3.12 in the guarded assignment $\widehat{g}_3^C$. Further, the two guarded assignments $\widehat{g}_2^C$ and $\widehat{g}_3^C$ have the same guard, but the former assigns 0 to $\widehat{best}$, while the latter assigns 1 to $\widehat{best}$. These two guarded assignments are used to capture the cases when two different processes are witnessed by the same abstract index, i.e., the same control state, but one of them receives a value 0, while the other one does not. In this case, have the same value for $\widehat{best}$, and the message array contains the values that both processes received. Thus, for an abstract local state $\widehat{local} \in \widehat{L}$ and a set Pred of predicates, that satisfy the guard

$\neg pr_{r>t+1} \wedge (\widehat{best} = 1) \wedge (\exists u\ in(0, \widehat{\mathbf{msg}}[u]))$, the result of $\widehat{update}^{C}_{\mathsf{Pred}}(\widehat{local})$ is a set of two control states: one where $\widehat{best}$ is 0, and one where $\widehat{best}$ is 1, that is $\widehat{update}^{C}_{\mathsf{Pred}}(\widehat{local}) = \{\langle 0, \bot, \bot\rangle, \langle 1, \bot, \bot\rangle\}$. □

3.5.3 Abstract Transition Relations $\widehat{Env}$, $\widehat{Snd}$, $\widehat{Upd}$

We now give a constructive definition of the transition relation $\widehat{\mathcal{T}}$. Similarly to the way we defined the transition relation $T(\mathsf{n}, \mathsf{t}, \mathsf{f})$ of the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ in Definition 2.18 on page 50, the abstract transition relation $\widehat{\mathcal{T}} \subseteq \widehat{\mathcal{S}} \times \widehat{\mathcal{S}}$ is a subset of the composition of three abstract transition relations, $\widehat{Env}$, $\widehat{Snd}$, and $\widehat{Upd}$, which are abstract encodings of the three transition relations $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$, $Snd(\mathsf{n}, \mathsf{t}, \mathsf{f})$, and $Upd(\mathsf{n}, \mathsf{t}, \mathsf{f})$, defined in Definitions 2.19, 2.20, and 2.21, respectively.

Definition 3.18 (Abstract transition relation $\widehat{\mathcal{T}}$)**.** The *abstract transition relation* $\widehat{\mathcal{T}}$ is a binary relation $\widehat{\mathcal{T}} \subseteq \widehat{\mathcal{S}} \times \widehat{\mathcal{S}}$, where a pair $(\widehat{\sigma}, \widehat{\sigma}''') \in \widehat{\mathcal{S}} \times \widehat{\mathcal{S}}$ of abstract states is an *abstract transition* in $\widehat{\mathcal{T}}$, i.e., $(\widehat{\sigma}, \widehat{\sigma}''') \in \widehat{\mathcal{T}}$, iff there exist $\widehat{\sigma}', \widehat{\sigma}'' \in \widehat{\mathcal{S}}$ such that:

- $(\widehat{\sigma}, \widehat{\sigma}') \in \widehat{Env}$,
- $(\widehat{\sigma}', \widehat{\sigma}'') \in \widehat{Snd}$,
- $(\widehat{\sigma}'', \widehat{\sigma}''') \in \widehat{Upd}$. □

In the remainder of this section, we define the three abstract transition relations $\widehat{Env}$, $\widehat{Snd}$, and $\widehat{Upd}$. The relations $\widehat{Env}$ and $\widehat{Snd}$ are direct abstract encodings of their concrete counterparts, namely the relations $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$ and $Snd(\mathsf{n}, \mathsf{t}, \mathsf{f})$, respectively. While $\widehat{Upd}$ is also an abstract encoding of the concrete relation $Upd(\mathsf{n}, \mathsf{t}, \mathsf{f})$, its definition is more involved. Since the relation $Upd(\mathsf{n}, \mathsf{t}, \mathsf{f})$ updates control states of processes, $\widehat{Upd}$ should define how the control states of processes witnessed by abstract indices $u \in U$ are updated. To do this, we use the abstract update functions, characterized using abstract guarded assignments, which we defined in Section 3.5.2. Further, as we use control states as indices of the array variables in $\widehat{V}_{\mathsf{sys}}$, an update of the control states implies an update of the array variables. When updating the array variables we do not need to define abstract versions of the message translation functions $translate_{\mathbf{y}}$. Rather, for an abstract index $u \in U$, we have to decode the control state that corresponds to that index, compute the possible successor control states, and reshuffle the contents stored in the array variables such that only cells indexed by indices that witness processes in the abstract state store a value different than $\emptyset$.

Abstract Transition Relation $\widehat{Env}$. The first abstract transition relation, $\widehat{Env}$, is used to update the predicates $\widehat{\mathsf{Pred}}$ and the environment variables $\widehat{\mathbf{cr}}$ and $\widehat{\mathbf{Rcv}}$.

Definition 3.19 (Abstract transition relation $\widehat{Env}$)**.** The abstract transition relation $\widehat{Env}$ is a binary relation $\widehat{Env} \subseteq \widehat{\mathcal{S}} \times \widehat{\mathcal{S}}$, such that two abstract states $\widehat{\sigma}, \widehat{\sigma}' \in \widehat{S}$ are in relation $\widehat{Env}$, i.e., $(\widehat{\sigma}, \widehat{\sigma}') \in \widehat{Env}$, iff:

1. $\widehat{\sigma}'.pr$ is assigned a value from $\{\bot, \top\}$ non-deterministically, for $pr \in \mathsf{Pred}$,
2. for $u \in U$:
   - $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u] = \emptyset$, if u does not witness a process in $\widehat{\sigma}$,
   - $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u] = \{\bot\}$, if u witnesses a process and $\widehat{\sigma}.control_u.fld = \top$,
   - $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ is assigned a value $\{\bot\}$ or $\{\top\}$ non-deterministically, if $u \in \{1, \ldots, \mathsf{m}\}$ and $\widehat{\sigma}.control_u.fld = \bot$,
   - $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ is assigned a value $\{\bot\}$, $\{\top\}$, or $\{\bot, \top\}$ non-deterministically, if $u \in C$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$, and $\widehat{\sigma}.control_u.fld = \bot$,
3. for $u, v \in U$:
   - $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u, v] = \emptyset$, if u or v do not witness a process in $\widehat{\sigma}$,
   - $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u, v] = \{\bot\}$, if both u and v witness a process, and $\widehat{\sigma}.control_v.fld = \top$,
   - $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u, v]$ is assigned a value $\{\bot\}$, $\{\top\}$, or $\{\bot, \top\}$ non-deterministically, if both u and v witness a process, and $\top \in \widehat{\sigma}'.\widehat{\mathbf{cr}}[v]$,
   - $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u, v] = \{\top\}$, if both u and v witness a process, $\widehat{\sigma}.control_v.fld = \bot$, and $\widehat{\sigma}'.\widehat{\mathbf{cr}}[v] = \{\bot\}$,
4. $\widehat{\sigma}'.\widehat{\mathbf{x}} = \widehat{\sigma}.\widehat{\mathbf{x}}$, for $\widehat{\mathbf{x}} \in \mathsf{cntl}(\widehat{V}_{\mathsf{sys}})$,
5. $\widehat{\sigma}'.\widehat{\mathbf{Y}} = \widehat{\sigma}.\widehat{\mathbf{Y}}$, for $\widehat{\mathbf{Y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{sys}})$. □

Transitions from a state $\widehat{\sigma} \in \widehat{S}$ to a state $\widehat{\sigma}' \in \widehat{S}$ using the relation $\widehat{Env}$ update the environment variables as follows. First, the predicates from the set Pred are assigned values non-deterministically. The control and neighborhood variables $\widehat{\mathbf{x}} \in \mathsf{cntl}(\widehat{V}_{\mathsf{sys}})$ and $\widehat{\mathbf{Y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{sys}})$ are not updated.

For $u \in U$, the value of $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ is set to $\emptyset$ if u does not witness any process, and to $\{\bot\}$ if u witnesses a failed process. To define the new crashes, $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ it assigned either $\{\bot\}$ or $\{\top\}$ non-deterministically, if u witnesses a non-failed process from the fixed m processes. If $u \in C$ witnesses a non-failed process, $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ is assigned one of the values $\{\bot\}$, $\{\top\}$, or $\{\bot, \top\}$ non-deterministically.

To build the new receiver lists, for every $u, v \in U$ that witness a process, the value of $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u, v]$ is set to $\{\bot\}$, if v witnesses a failed process. If v witnesses a crashed process, that is, if $\top \in \widehat{\sigma}'.\widehat{\mathbf{cr}}[v]$, then $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u, v]$ is assigned one of the values $\{\bot\}$, $\{\top\}$ or $\{\bot, \top\}$ non-deterministically. Otherwise, if v witnesses correct processes, that is, if $\widehat{\sigma}.control_v.fld = \bot$, and $\widehat{\sigma}'.\widehat{\mathbf{cr}}[v] = \{\bot\}$, we have that $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u, v] = \{\top\}$. For abstract indices from $u, v \in U$ that do not witness a process, we have $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u, v] = \emptyset$.

The lemma below shows the correspondence between the two states $s, s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ of a concrete system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, which are in relation $Env(\mathsf{n}, \mathsf{t}, \mathsf{f})$, and their corresponding abstract states $\widehat{\sigma}, \widehat{\sigma}' \in \widehat{\mathcal{S}}$, where $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$ and $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$.

Lemma 3.2. *Let* $\widehat{\mathsf{STS}} = \langle \widehat{S}, \widehat{S}_0, \widehat{T} \rangle$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *induced by the abstraction mapping* $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$*, for* $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition. For two states* $s, s' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$*, we have* $(s,s') \in \mathit{Env}(\mathsf{n},\mathsf{t},\mathsf{f})$ *implies* $(\widehat{\sigma},\widehat{\sigma}') \in \widehat{\mathit{Env}}$*, where* $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$ *and* $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$*.*

Proof. Follows from the definition of the transition relation $\mathit{Env}(\mathsf{n},\mathsf{t},\mathsf{f})$ (Definition 2.19 on page 51), the definitions of the abstraction mappings $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$, $\widetilde{\alpha}_{\mathsf{n}}$, and $\widehat{\alpha}_{\mathsf{n}}$, (Definitions 3.6, 3.12, and 3.14, respectively), and the definition of the abstract transition relation $\widehat{\mathit{Env}}$ (Definition 3.19). □

Example 3.14. Recall Examples 2.6 and 2.7 on pages 50 and 52 where we depicted the states s and s', respectively, such that $(s,s') \in \mathit{Env}(\mathsf{n},\mathsf{t},\mathsf{f})$ in the system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ for the algorithm FloodMin for $k = 1$, where $\mathsf{n} = 6$, $\mathsf{t} = 3$, and $\mathsf{f} = 2$. We showed the abstraction of the state s in Example 3.9, with the state $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$ depicted on page 81. Below, we show the abstract state $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$. We omit the valuations of the variables $\widehat{\mathbf{best}}, \widehat{\mathbf{dec}}, \widehat{\mathbf{fld}}, \widehat{\mathbf{active}}$, and $\widehat{\mathbf{Msg}}$, which coincide with those in the state $\widehat{\sigma}$.

$\widehat{\sigma}'$: $\widehat{\mathbf{best}}$ $\widehat{\mathbf{dec}}$ $\widehat{\mathbf{fld}}$ $\widehat{\mathbf{Msg}}$ $\widehat{\mathbf{active}}$

$\cdots$ $\cdots$ $\cdots$ $\cdots$ $\cdots$

$pr_{r>t+1}$ $\widehat{\mathbf{cr}}$ $\widehat{\mathbf{Rcv}}$

		$\widehat{\mathbf{cr}}$	$\widehat{\mathbf{Rcv}}$							
				1	2	cnt_1	cnt_2	$\cdots$	cnt_7	$\cdots$	cnt_{12}
	1	$\{\bot\}$	1	$\{\top\}$	$\{\top\}$	$\{\bot,\top\}$	$\emptyset$	$\cdots$	$\{\top\}$	$\cdots$	$\emptyset$
	2	$\{\bot\}$	2	$\{\top\}$	$\{\top\}$	$\{\bot\}$	$\emptyset$	$\cdots$	$\{\top\}$	$\cdots$	$\emptyset$
	cnt_1	$\{\top\}$	cnt_1	$\{\top\}$	$\{\top\}$	$\{\bot\}$	$\emptyset$	$\cdots$	$\{\top\}$	$\cdots$	$\emptyset$
	cnt_2	$\emptyset$	cnt_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\cdots$	$\emptyset$	$\cdots$	$\emptyset$
$\bot$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$
	cnt_7	$\{\bot\}$	cnt_7	$\{\top\}$	$\{\top\}$	$\{\bot,\top\}$	$\emptyset$	$\cdots$	$\{\top\}$	$\cdots$	$\emptyset$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$
	cnt_{12}	$\emptyset$	cnt_{12}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\cdots$	$\emptyset$	$\cdots$	$\emptyset$

The difference between the state $\widehat{\sigma}$, depicted on page 81 and the state $\widehat{\sigma}'$ is the following. In the state $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$, we have $\widehat{\sigma}'.\widehat{\mathbf{cr}}[cnt_1] = \{\top\}$, since in the concrete state s' we have $s'.\mathbf{cr}[3] = s'.\mathbf{cr}[6] = \top$, that is the processes 3 and 6, which are witnessed by the abstract index cnt_1 are flagged as crashed. Further, we have $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[1, cnt_1] = \widehat{\sigma}'.\widehat{\mathbf{Rcv}}[cnt_7, cnt_1] = \{\bot,\top\}$, since in the concrete state s', process 3, witnessed by cnt_1, has process 1 in its receiver list, and process 6, witnessed by cnt_1, has process 4, witnessed by cnt_7 in its receiver list.

It is easy to check that the pair of abstract states $(\widehat{\sigma},\widehat{\sigma}')$ are in relation $\widehat{\mathit{Env}}$ by checking that the state $\widehat{\sigma}'$ satisfies the conditions of Definition 3.19. First, $\widehat{\sigma}'.pr_{r>t+1} = \bot$ is a

value from the set $\{\bot, \top\}$. Second, in the variables $\widehat{\sigma}.\widehat{\mathbf{cr}}$ and $\widehat{\sigma}.\widehat{\mathbf{Rcv}}$, the cells indexed by abstract indices that do not witness a process in $\widehat{\sigma}$ store the value $\emptyset$. The values stored in the cells indexed by abstract indices that witness a process in $\widehat{\sigma}$ follow the constraints listed in Definition 3.19. Finally, the values of the variables $\widehat{\mathbf{best}}, \widehat{\mathbf{dec}}, \widehat{\mathbf{fld}}, \widehat{\mathbf{active}}$, and $\widehat{\mathbf{Msg}}$ are the same in both states $\widehat{\sigma}$ and $\widehat{\sigma}'$. □

Abstract Transition Relation $\widehat{Snd}$. The second abstract transition relation, $\widehat{Snd}$, updates the abstract message array $\widehat{\mathbf{Msg}}$. To write messages into the abstract message array $\widehat{\mathbf{Msg}}$, we will use the message generation function *send_msg*, defined in Definition 2.7.

Definition 3.20 (Abstract transition relation $\widehat{Snd}$)**.** The abstract transition relation $\widehat{Snd}$ is a binary relation $\widehat{Snd} \subseteq \widehat{\mathcal{S}} \times \widehat{\mathcal{S}}$, such that two abstract states $\widehat{\sigma}, \widehat{\sigma}' \in \widehat{\mathcal{S}}$ are in relation $\widehat{Snd}$, i.e., $(\widehat{\sigma}, \widehat{\sigma}') \in \widehat{Snd}$, iff:

1. $send\_msg(\widehat{\sigma}.control_v) \in \widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v]$, for $u, v \in U$, such that $\top \in \widehat{\sigma}.\widehat{\mathbf{Rcv}}[u, v]$,
2. $\bot \in \widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v]$, for $u, v \in U$, such that $\bot \in \widehat{\sigma}.\widehat{\mathbf{Rcv}}[u, v]$,
3. $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v] = \emptyset$ otherwise,
4. $\widehat{\sigma}'.\widehat{\mathbf{x}} = \widehat{\sigma}.\widehat{\mathbf{x}}$, for $\widehat{\mathbf{x}} \in \mathsf{cntl}(\widehat{V}_{\mathsf{sys}})$,
5. $\widehat{\sigma}'.\widehat{\mathbf{Y}} = \widehat{\sigma}.\widehat{\mathbf{Y}}$, for $\widehat{\mathbf{Y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{sys}}) \setminus \{\widehat{\mathbf{Msg}}\}$,
6. $\widehat{\sigma}'.pr = \widehat{\sigma}.pr$, for $pr \in \mathsf{Pred}$, $\widehat{\sigma}'.\widehat{\mathbf{cr}} = \widehat{\sigma}.\widehat{\mathbf{cr}}$, and $\widehat{\sigma}'.\widehat{\mathbf{Rcv}} = \widehat{\sigma}.\widehat{\mathbf{Rcv}}$. □

When taking a step by applying $\widehat{Snd}$, a message $send\_msg(\widehat{\sigma}.control_v)$ is written in the abstract message array $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v]$, if the abstract receiver list $\widehat{\sigma}.\widehat{\mathbf{Rcv}}$ stores the value $\top$ in the cell $\widehat{\sigma}.\widehat{\mathbf{Rcv}}[u, v]$, for $u, v \in U$. Similarly, the empty message $\bot$ is written in $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v]$, if $\bot \in \widehat{\sigma}.\widehat{\mathbf{Rcv}}[u, v]$. The remaining variables are not updated using $\widehat{Snd}$.

The following lemma shows that for two states $s, s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$ of a concrete system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, which are in relation $Snd(\mathsf{n}, \mathsf{t}, \mathsf{f})$, their corresponding abstract states $\widehat{\sigma}, \widehat{\sigma}' \in \widehat{\mathcal{S}}$, where $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$ and $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$, are in relation $\widehat{Snd}$.

Lemma 3.3. *Let $\widehat{\mathsf{STS}} = \langle \widehat{S}, \widehat{S}_0, \widehat{T} \rangle$ be the overapproximation of $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ induced by the abstraction mapping $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$, for $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition. For two states $s, s' \in S(\mathsf{n}, \mathsf{t}, \mathsf{f})$, we have $(s, s') \in Snd(\mathsf{n}, \mathsf{t}, \mathsf{f})$ implies $(\widehat{\sigma}, \widehat{\sigma}') \in \widehat{Snd}$, where $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$ and $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$.*

Proof. Follows from the definition of the transition relation $Snd(\mathsf{n}, \mathsf{t}, \mathsf{f})$ (Definition 2.20 on page 52), the definitions of the abstraction mappings $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}, \widetilde{\alpha}_{\mathsf{n}}$, and $\widehat{\alpha}_{\mathsf{n}}$, (Definitions 3.6, 3.12, and 3.14, respectively), and the definition of the abstract transition relation $\widehat{Env}$ (Definition 3.20). □

Example 3.15. Recall Examples 2.7 and 2.8 on pages 52 and 53, respectively, and consider the states $s', s'' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ depicted there, for which we have $(s', s'') \in \mathit{Snd}(\mathsf{n},\mathsf{t},\mathsf{f})$. We showed the abstract state $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$ in Example 3.14. Since in s'', only the value of the variable **Msg** is updated, we show below the abstract state $\widehat{\sigma}'' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s'')$, where we omit the valuations of all variables, except the variable $\widehat{\mathbf{Msg}}$.

$\widehat{\sigma}''$:

$\widehat{\mathbf{best}}$	$\widehat{\mathbf{dec}}$	$\widehat{\mathbf{fld}}$
$\ldots$	$\ldots$	$\ldots$

$\widehat{\mathbf{active}}$: $\ldots$

$\widehat{\mathbf{Msg}}$

	1	2	cnt_1	cnt_2	$\cdots$	cnt_7	$\cdots$	cnt_{12}
1	$\{1\}$	$\{1\}$	$\{\bot, 0\}$	$\emptyset$	$\cdots$	$\{1\}$	$\cdots$	$\emptyset$
2	$\{1\}$	$\{1\}$	$\{\bot\}$	$\emptyset$	$\cdots$	$\{1\}$	$\cdots$	$\emptyset$
cnt_1	$\{1\}$	$\{1\}$	$\{\bot\}$	$\emptyset$	$\cdots$	$\{1\}$	$\cdots$	$\emptyset$
cnt_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\cdots$	$\emptyset$	$\cdots$	$\emptyset$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$
cnt_7	$\{1\}$	$\{1\}$	$\{\bot, 0\}$	$\emptyset$	$\cdots$	$\{1\}$	$\cdots$	$\emptyset$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$
cnt_{12}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\cdots$	$\emptyset$	$\cdots$	$\emptyset$

$pr_{r>t+1}$	$\widehat{\mathbf{cr}}$
$\ldots$	$\ldots$

$\widehat{\mathbf{Rcv}}$: $\ldots$

Recall that the two processes 3 and 6 flagged as crashed in the state s'' have processes 1 and 4, respectively, in their receiver lists, and only send messages to those processes. Both processes 3 and 6 are witnessed by the abstract index cnt_1, and by the definitions of the abstraction mappings, in the state $\widehat{\sigma}'$, presented in Example 3.14, we have $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[1, cnt_1] = \widehat{\sigma}'.\widehat{\mathbf{Rcv}}[cnt_7, cnt_1] = \{\bot, \top\}$.

By the definitions of the three abstraction mappings, we have $\widehat{\sigma}''.\widehat{\mathbf{Msg}}[1, cnt_1] = \{\bot, 0\}$, since process 3 sent a value 0 to process 1, but process 6 did not send a message (i.e., it sent a null message $\bot$). Similarly, we have $\widehat{\sigma}''.\widehat{\mathbf{Msg}}[cnt_7, cnt_1] = \{\bot, 0\}$, since process 6 sent a value 0 to process 4, which is witnessed by the abstract index cnt_7, but process 1 sent a null message $\bot$ to both processes witnessed by the abstract index cnt_7. All other processes sent value 1 to everyone, that is we have $\widehat{\sigma}''.\widehat{\mathbf{Msg}}[u, v] = \{1\}$, where $u, v \in U$ witness a process and $v \neq cnt_1$.

We thus see that the value of the variable $\widehat{\mathbf{Msg}}$ in the abstract state $\widehat{\sigma}''$ satisfies the conditions 1, 2, and 3 of Definition 3.20, since $\mathit{send\_msg}(\widehat{\sigma}'.\mathit{control}_{cnt_1}) = 0$ and $\mathit{send\_msg}(\widehat{\sigma}'.\mathit{control}_1) = \mathit{send\_msg}(\widehat{\sigma}'.\mathit{control}_2) = \mathit{send\_msg}(\widehat{\sigma}'.\mathit{control}_{cnt_7}) = 1$. Further, as the values of the other variables do not change, we get that $(\widehat{\sigma}', \widehat{\sigma}'') \in \widehat{\mathit{Snd}}$. □

Abstract Transition Relation $\widehat{\mathit{Upd}}$. The final abstract transition relation, $\widehat{\mathit{Upd}}$, is used to update the control and neighborhood variables $\widehat{\mathbf{x}} \in \mathsf{cntl}(\widehat{V}_{\mathsf{sys}})$ and $\widehat{\mathbf{Y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{sys}})$. To update the control variables $\widehat{\mathbf{x}} \in \mathsf{cntl}(\widehat{V}_{\mathsf{sys}}) \setminus \{\widehat{\mathbf{active}}\}$, we will use the abstract update

function $\widehat{update}^{\mathsf{m}}_{\mathsf{Pred}}$, defined in Definition 3.16. To update the neighborhood variables, $\widehat{\mathbf{Y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{sys}}) \setminus \{\widehat{\mathbf{Msg}}\}$, we will use the message translation functions $translate_{\mathbf{y}}$, defined in Definition 2.7 on page 41. Observe that we do not need to define abstract versions of the message translation functions $translate_{\mathbf{y}} : \mathcal{M} \to D_{\mathbf{y}}$, as we will apply them directly to the values $m \in \mathcal{M}$, which are stored in sets in the cells of the two-dimensional array variable $\widehat{\mathbf{Msg}}$. Finally, to update the control variable $\widehat{\mathbf{active}}$, for every process control state $u \in C$, that witnesses a process, we proceed as follows.

Let $u \in C$ be an abstract index that witnesses a correct process in an abstract state $\widehat{\sigma}$. Updating the control state $\widehat{\sigma}.control_u$ results in a set of control states, which were computed using the abstract update function $\widehat{update}^{C}_{\mathsf{Pred}}$, defined in Definition 3.17. If u witnesses a crashed process, it is updated to a control state whose failure flag *fld* is set to $\top$. To define the set of all possible successor control states of a given $u \in C$, when taking a step in the transition relation $\widehat{Upd}$, we define the following function, that given an abstract state $\widehat{\sigma} \in \widehat{S}$ and an abstract index $u \in U$, defines a set of abstract indices which correspond to u in the successor state $\widehat{\sigma}'$ of the state $\widehat{\sigma}$.

Definition 3.21 (Successor function $\widehat{succ}$)**.** The *control state successor function* $\widehat{succ} : \widehat{\mathcal{S}} \times U \to 2^U$ maps an abstract state $\widehat{\sigma} \in \widehat{\mathcal{S}}$ and an abstract index $u \in U$ to a set $\widehat{succ}(\widehat{\sigma}, u) \subseteq U$ of abstract indices where:

- if $u \in \{1, \ldots, \mathsf{m}\}$, then $\widehat{succ}(\widehat{\sigma}, u) = \{u\}$,
- if $u \in C$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = 0$, then $\widehat{succ}(\widehat{\sigma}, u) = \emptyset$
- if $u \in C$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$, then for every $u' \in C$, we have $u' \in \widehat{succ}(\widehat{\sigma}, u)$ iff:
  1. $u.fld = \top$ and $u' = u$,
  2. $u.fld = \bot$, $\top \in \widehat{\sigma}.\widehat{\mathbf{cr}}[u]$, $u'.x = u.x$, for $x \neq fld$, and $u'.fld = \top$,
  3. $u.fld = \bot$, $\bot \in \widehat{\sigma}.\widehat{\mathbf{cr}}[u]$, and $u' \in \widehat{update}^{C}_{\widehat{\sigma}.\mathsf{Pred}}(\widehat{\sigma}.local_u)$. □

If the abstract index $u \in U$ witnesses one of the fixed m processes in $\widehat{\sigma}$, then its set of successor abstract indices is the singleton $\{u\}$. If the abstract index $u \in C$ does not witness a process in $\widehat{\sigma}$, that is, if $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = 0$, then its set of successor abstract indices is empty. Otherwise, its set of successor abstract indices contains (i) itself, if u witnesses a failed process, (ii) the control state u' where only the failure flag is updated to $\top$, if u witnesses a crashed process, and (iii) the results of the abstract update function $\widehat{update}^{C}_{\mathsf{Pred}}$, if u witnesses a correct process.

Example 3.16. Consider the abstract state $\widehat{\sigma}''$, presented in Example 3.15. We show the result of applying the mapping $\widehat{succ}$ to the state $\widehat{\sigma}'' \in \widehat{\sigma}$ and the abstract indices $u \in U$:

- for $u \in \{1, \ldots, \mathsf{m}\}$, i.e., in our example, for $u \in \{1, 2\}$, we have $\widehat{succ}(\widehat{\sigma}'', u) = \{u\}$,

- for $u = cnt_1$, since we have $\widehat{\sigma}''.\widehat{\mathbf{active}}[u] = \mathsf{many}$, and as $u.fld = \bot$ and $\widehat{\sigma}''.\widehat{\mathbf{cr}}[u] = \{\top\}$, we have $\widehat{succ}(\widehat{\sigma}'', cnt_1) = \{cnt_2\}$, where $cnt_2.x = cnt_1.x$, for $x \neq fld$, and $cnt_2.fld = \top$,
- for $u = cnt_7$, since we have $\widehat{\sigma}''.\widehat{\mathbf{active}}[u] = \mathsf{many}$, and as $u.fld = \bot$ and $\widehat{\sigma}''.\widehat{\mathbf{cr}}[u] = \{\bot\}$, we have $\widehat{succ}(\widehat{\sigma}'', cnt_7) = \widehat{update}^{C}_{\widehat{\sigma}''.\mathsf{Pred}}(\widehat{\sigma}''.local_{cnt_7})$. The abstract local state $\widehat{\sigma}''.local_{cnt_7}$ is the tuple $\langle cnt_7, \left[\{1\} \;\; \{1\} \;\; \{\bot, 0\} \;\; \emptyset \;\; \ldots \;\; \{1\} \;\; \ldots \;\; \emptyset\right]\rangle$. As we saw in Example 3.13, in this local state, both guarded assignments $\widehat{g}_2^C$ and $\widehat{g}_3^C$ are applied, hence we have $\widehat{update}^{C}_{\widehat{\sigma}''.\mathsf{Pred}}(\widehat{\sigma}''.local_{cnt_7}) = \{\langle 0, \bot, \bot\rangle, \langle 1, \bot, \bot\rangle\} = \{cnt_1, cnt_7\}$. □

Definition 3.22 (Abstract transition relation $\widehat{Upd}$)**.** The abstract transition relation $\widehat{Upd}$ is a binary relation $\widehat{Upd} \subseteq \widehat{\mathcal{S}} \times \widehat{\mathcal{S}}$, such that two abstract states $\widehat{\sigma}, \widehat{\sigma}' \in \widehat{\mathcal{S}}$ are in relation $\widehat{Upd}$, i.e., $(\widehat{\sigma}, \widehat{\sigma}') \in \widehat{Upd}$, iff:

1. $\widehat{\sigma}'.\widehat{\mathbf{fld}}[u'] = \widehat{\sigma}.\widehat{\mathbf{fld}}[u'] \vee (\widehat{\sigma}.\widehat{\mathbf{cr}}[u'] = \{\top\})$, for $u' \in \{1, \ldots, \mathsf{m}\}$
2. $\widehat{\sigma}'.control_{u'} = \begin{cases} \widehat{update}^{\mathsf{m}}_{\widehat{\sigma}.\mathsf{Pred}}(\widehat{\sigma}.local_{u'}) & \text{if } \widehat{\sigma}'.\widehat{\mathbf{fld}}[u'] = \bot \\ \widehat{\sigma}.control_{u'} & \text{otherwise} \end{cases}$, for $u' \in \{1, \ldots, \mathsf{m}\}$
3. $\widehat{\sigma}'.\widehat{\mathbf{active}}[u'] = \begin{cases} \mathsf{many} & \text{if } \exists\, u \in C, \text{ such that } u' \in \widehat{succ}(\widehat{\sigma}, u) \\ 0 & \text{otherwise} \end{cases}$, for $u' \in C$,
4. $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u', v'] = \bigcup_{u,v \in U}\{translate_{\mathbf{y}}(m) \mid u' \in \widehat{succ}(\widehat{\sigma}, u), v' \in \widehat{succ}(\widehat{\sigma}, v), \text{ and } m \in \widehat{\mathbf{Msg}}[u, v]\}$, for $u', v' \in U$ and $\widehat{\mathbf{Y}} \in \mathsf{nbhd}(\widehat{V}_{\mathsf{sys}}) \setminus \widehat{\mathbf{Msg}}$,
5. $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u', v'] = \begin{cases} \{\bot\} & \text{if both } u', v' \text{ witness a process in } \widehat{\sigma}' \\ \emptyset & \text{otherwise} \end{cases}$, for $u', v' \in U$,
6. $\widehat{\sigma}'.pr = \widehat{\sigma}.pr$, for $pr \in \mathsf{Pred}$,
7. $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u'] = \bigcup_{u \in U}\{\widehat{\sigma}.\widehat{\mathbf{cr}}[u] \mid u' \in \widehat{succ}(\widehat{\sigma}, u)\}$ for $u' \in U$,
8. $\widehat{\sigma}'.\widehat{\mathbf{Rcv}}[u', v'] = \bigcup_{u,v \in U}\{\widehat{\sigma}.\widehat{\mathbf{Rcv}}[u, v] \mid u' \in \widehat{succ}(\widehat{\sigma}, u) \text{ and } v' \in \widehat{succ}(\widehat{\sigma}, v)\}$, for $u', v' \in U$. □

Updating the control states cause updates in the indices of the neighborhood variables. Thus, the control state successor function is also used in the update of the neighborhood variables $\mathsf{nbhd}(\widehat{V}_{\mathsf{sys}})$. Further, although the concrete transition relation $Upd(\mathsf{n}, \mathsf{t}, \mathsf{f})$ does not update the environment variables **cr** and **Rcv**, in the abstract transition relation $\widehat{Upd}$ we need to update their abstract counterparts $\widehat{\mathbf{cr}}$ and $\widehat{\mathbf{Rcv}}$, since they are also indexed by abstract indices. Hence, the control state update will induce an update in all the variables that are indexed by abstract indices.

The lemma below shows that for two states $s, s' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ of a concrete system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, which are in relation $\mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$, their corresponding abstract states $\widehat{\sigma}, \widehat{\sigma}' \in \widehat{\mathcal{S}}$, where $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$ and $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$ are in relation $\widehat{\mathit{Upd}}$.

Lemma 3.4. *Let* $\widehat{\mathsf{STS}} = \langle \widehat{S}, \widehat{S}_0, \widehat{T} \rangle$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *induced by the abstraction mapping* $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$*, for* $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition. For two states* $s, s' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$*, we have* $(s, s') \in \mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$ *implies* $(\widehat{\sigma}, \widehat{\sigma}') \in \widehat{\mathit{Upd}}$*, where* $\widehat{\sigma} = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s)$ *and* $\widehat{\sigma}' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s')$*.*

Proof. Follows from the definition of the transition relation $\mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$ (Definition 2.21 on page 51), the definitions of the abstraction mappings $\alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$, $\widetilde{\alpha}_{\mathsf{n}}$, and $\widehat{\alpha}_{\mathsf{n}}$, (Definitions 3.6, 3.12, and 3.14, respectively), and the definition of the abstract transition relation $\widehat{\mathit{Upd}}$ (Definition 3.22). □

Example 3.17. Recall Examples 2.8 and 2.9 on pages 53 and 54, respectively, and consider the states $s'', s''' \in S(\mathsf{n},\mathsf{t},\mathsf{f})$ depicted there, for which we have $(s'', s''') \in \mathit{Upd}(\mathsf{n},\mathsf{t},\mathsf{f})$. We showed the abstract state $\widehat{\sigma}'' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s'')$ in Example 3.15, and we presented the abstract state $\widehat{\sigma}''' = \widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}(s''')$ in Example 3.9.

We show that the states $\widehat{\sigma}'', \widehat{\sigma}''' \in \widehat{\mathcal{S}}$ satisfy the conditions of Definition 3.22.

1. for $u \in \{1, \ldots, \mathsf{m}\}$, i.e., in our case for $u \in \{1, 2\}$, we have $\widehat{\sigma}'''.\widehat{\mathbf{fld}}[u] = \bot$, since the disjunction $\widehat{\sigma}''.\widehat{\mathbf{fld}}[u] \vee (\widehat{\sigma}''.\widehat{\mathbf{cr}}[u] = \{\top\})$ evaluates to $\bot$. Hence, condition 1 of Definition 3.22 holds.

2. since we have $\widehat{\sigma}'''.\widehat{\mathbf{fld}}[u] = \bot$ for both $u \in \{1, 2\}$, we get that $\widehat{\sigma}'''.\mathit{control}_u = \widehat{\mathit{update}}^{\mathsf{m}}_{\widehat{\sigma}''.\mathsf{Pred}}(\widehat{\sigma}''.\mathit{control}_u)$. Recall that in Example 3.12, we characterized the abstract update function $\widehat{\mathit{update}}^{\mathsf{m}}_{\widehat{\sigma}''.\mathsf{Pred}}$ using the abstract guarded assignments $\widehat{g}^{\mathsf{m}}_1$–$\widehat{g}^{\mathsf{m}}_5$. For $u = 1$, we can apply the guarded assignment $\widehat{g}^{\mathsf{m}}_2$, and for $u = 2$, we can apply the guarded assignment $\widehat{g}^{\mathsf{m}}_1$, both defined in Example 3.12. Hence, condition 2 of Definition 3.22 holds.

3. in the state $\widehat{\sigma}'''$, we have $\widehat{\sigma}'''.\widehat{\mathbf{active}}[u] = \mathsf{many}$, for $u \in \{\mathit{cnt}_1, \mathit{cnt}_2, \mathit{cnt}_7\}$. In Example 3.16, we defined $\widehat{\mathit{succ}}(\widehat{\sigma}'', \mathit{cnt}_1) = \{\mathit{cnt}_2\}$ and $\widehat{\mathit{succ}}(\widehat{\sigma}'', \mathit{cnt}_7) = \{\mathit{cnt}_1, \mathit{cnt}_7\}$. Hence, the value of the variable $\widehat{\mathbf{active}}$ satisfies the condition 3 of Definition 3.22.

4. since in the state $\widehat{\sigma}'''$ we do not have neighborhood variables other than $\widehat{\mathbf{Msg}}$, condition 4 of Definition 3.22 trivially holds.

5. condition 5 of Definition 3.22 holds because in the state $\widehat{\sigma}'''$, the abstract indices that witness a process are: 1, 2, cnt_1, cnt_2, and cnt_7. Thus, only the cells of $\widehat{\sigma}'''.\widehat{\mathbf{Msg}}$ whose rows and columns are indexed by these abstract indices contain the value $\{\bot\}$, the others contain the value $\emptyset$.

6. since in the transition $(s'', s''') \in \mathit{Upd}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ in the concrete system the round number is not updated, the predicates $pr \in \mathsf{Pred}$ are also not updated in their corresponding abstract states, that is, $\widehat{\sigma}'''.pr = \widehat{\sigma}''.pr$, for $pr \in \mathsf{Pred}$.

7. in the state $\widehat{\sigma}'''$, we have $\widehat{\sigma}'''.\widehat{\mathbf{cr}}[\widehat{\sigma}_1] = \widehat{\sigma}'''.\widehat{\mathbf{cr}}[\widehat{\sigma}_7] = \{\bot\}$. Since $\widehat{succ}(\widehat{\sigma}'', cnt_7) = \{cnt_1, cnt_7\}$ and $\widehat{\sigma}'''.\widehat{\mathbf{cr}}[\widehat{\sigma}_7] = \{\bot\}$, condition 7 of Definition 3.22 holds for the abstract indices $cnt_1, cnt_7 \in U$. Similar reasoning can be done for the abstract index $cnt_2 \in U$.

8. in the state $\widehat{\sigma}'''$, we have $\widehat{\sigma}'''.\widehat{\mathbf{Rcv}}[cnt_1, cnt_2] = \{\bot, \top\}$. Since we have $cnt_1 \in \widehat{succ}(\widehat{\sigma}'', cnt_7)$ and $cnt_2 \in \widehat{succ}(\widehat{\sigma}'', cnt_1)$, and moreover, as $\widehat{\sigma}''.\widehat{\mathbf{Rcv}}[cnt_7, cnt_1] = \{\bot, \top\}$, we get that condition 8 of Definition 3.22 holds for the abstract indices $cnt_1, cnt_2 \in U$. Similar reasoning can be done for the remaining abstract indices.

Thus, we can conclude that $(\widehat{\sigma}'', \widehat{\sigma}''') \in \widehat{\mathit{Upd}}$. □

3.5.4 Simulation

Let $\widehat{\mathsf{STS}} = \langle \widehat{S}, \widehat{S}_0, \widehat{T} \rangle$ be the abstract system obtained as an overapproximation of the system $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$, for $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ that satisfy the resilience condition, and let $\widehat{\mathsf{STS}}^* = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \widehat{\mathcal{T}} \rangle$ be the abstract system obtained in a constructive way, where $\widehat{\mathcal{S}} = \widehat{S}$, and where $\widehat{\mathcal{S}}_0$ and $\widehat{\mathcal{T}}$ are defined in Definitions 3.15 and 3.18, respectively. We now show that the system $\widehat{\mathsf{STS}}^*$ *simulates* the system $\widehat{\mathsf{STS}}$. We start by adapting the notion of simulation [BK08] to the systems $\widehat{\mathsf{STS}}$ and $\widehat{\mathsf{STS}}^*$.

Definition 3.23 (Simulation [BK08])**.** The system $\widehat{\mathsf{STS}}^* = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \widehat{\mathcal{T}} \rangle$ *simulates* the system $\widehat{\mathsf{STS}} = \langle \widehat{S}, \widehat{S}_0, \widehat{T} \rangle$ iff:

- for every initial state $\widehat{\sigma}_0 \in \widehat{S}_0$, we have $\widehat{\sigma}_0 \in \widehat{\mathcal{S}}_0$,
- for every transition $(\widehat{\sigma}, \widehat{\sigma}') \in \widehat{T}$, we have $(\widehat{\sigma}, \widehat{\sigma}') \in \widehat{\mathcal{T}}$. □

The following theorem follows from Lemmas 3.1, 3.2, 3.3, and 3.4 and the definitions of overapproximation and simulation.

Theorem 3.3 (Simulation)**.** *Let* $\widehat{\mathsf{STS}}$ *be the overapproximation of* $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ *induced by the abstraction mapping* $\widehat{\alpha}_{\mathsf{n}} \circ \widetilde{\alpha}_{\mathsf{n}} \circ \alpha_{\mathsf{n},\mathsf{t},\mathsf{f}}$*. Let* $\widehat{\mathsf{STS}}^* = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \widehat{\mathcal{T}} \rangle$ *be the abstract system obtained in a constructive way. Then, the system* $\widehat{\mathsf{STS}}^*$ *simulates the system* $\widehat{\mathsf{STS}}$*.* □

The fact that $\widehat{\mathsf{STS}}^*$ simulates $\widehat{\mathsf{STS}}$ implies that they satisfy the same ACTL^* formulas [BK08, Corollary 7.80.], where by ACTL^* we denote the universal fragment of CTL^* [BK08, Definition 7.74.]. Our indexed-LTL fragment can be easily translated to LTL

Table 3.1: Experimental results for fixed-size model checking

algorithm	$\mathsf{STS}(3,2,1) \models \phi$		$\mathsf{STS}(4,3,2) \models \phi$		$\mathsf{STS}(5,4,3) \models \phi$	
	states	time	states	time	states	time
EDAC	26 962	5s	242 605	16s	124 183 639	4h1min
ESC	10 543	4s	170 088	12s	71 913 792	1h57min
FairCons	9 118	4s	138 160	11s	53 816 397	1h43min
FloodMin, $k = 2$	13 215	6s	287 001	1min1s	out of memory in 3d11h	
FloodMin, $k = 1$	5 662	4s	81 918	17s	29 848 322	3h41min
FloodSet	6 937	4s	99 783	10s	34 724 276	1h18min
NBAC	256	1s	16 120	7s	3 335 753	10min33s

(by replacing the bounded universal quantifiers with conjunctions, and the bounded existential quantifiers with disjunctions). Since LTL is a fragment of ACTL* [BK08, Lemma 7.75.], as a consequence of this and Theorem 3.1, we obtain the following corollary.

Corollary 3.1. *Let* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *be a system, for* $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition, and let* $\widehat{\mathsf{STS}}^* = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \widehat{\mathcal{T}} \rangle$ *be the abstract system obtained in a constructive way. Let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number such that* $\mathsf{m} \leq \mathsf{n}$*. For every formula* $\phi \in \mathcal{F}(k)$*, we have:*

$$\widehat{\mathsf{STS}}^* \models \mathsf{sym}(\phi, \mathsf{m}) \quad \textit{implies} \quad \mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) \models \phi \qquad \square$$

Further, for our crash-tolerant benchmarks, using Theorem 3.2, we obtain the following corollary.

Corollary 3.2. *Let* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *be a system, for* $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$ *that satisfy the resilience condition, and let* $\widehat{\mathsf{STS}}^* = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}_0, \widehat{\mathcal{T}} \rangle$ *be the abstract system obtained in a constructive way. Suppose that* $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$ *models a synchronous fault-tolerant distributed algorithm that tolerates crash faults. Let* $\mathsf{m} \in \mathbb{N}$ *be a fixed number such that* $\mathsf{m} \leq \mathsf{n}$*. For every formula* $\phi \in \mathcal{F}(k)$*, we have:*

$$\widehat{\mathsf{STS}}^* \models \phi_{VC} \rightarrow \mathsf{sym}(\phi, \mathsf{m}) \quad \textit{implies} \quad \mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f}) \models \phi \qquad \square$$

3.6 Experimental Evaluation

In our experimental evaluation, we encoded several synchronous agreement algorithms, namely EDAC, ESC, FairCons, FloodMin, FloodSet, and NBAC, using the specification language TLA+ [Lam02]. For each algorithm, we encoded both the parameterized system $\mathsf{STS}(n,t,f)$ and the constructively defined abstract system $\widehat{\mathsf{STS}}^*$. The TLA+ specifications of our benchmarks can be found in [Stob]. To understand the tradeoff between fixed-size model checking and parameterized model checking, we ran two sets of experiments using the model checker TLC [TLA]. The experiments were run on a machine with two 12-core Intel(R) Xeon(R) E5-2650 v4 CPUs and 256 GB RAM.

Table 3.2: Experimental results for parameterized model checking

algorithm	m	$\mathsf{STS}(n,t,f) \models \phi^{\mathsf{m}}$		m'	$\mathsf{STS}(n,t,f) \models \phi^{\mathsf{m}'}$	
		states	time		states	time
EDAC	2	416 120	4h 35min	1	35 027	2min 28s
ESC	2	163 772	44min 30s	1	12 784	1min 19s
FairCons	2	160 523	3min	1	26 967	18s
FloodMin, $k = 2$	3	10 116 820	10d 16h	2	512 861	1h 39min
				1	43 601	2min 2s
FloodMin, $k = 1$	2	3 5083	55s	1	4 355	6s
FloodSet	2	210 583	2min 28s	1	17 911	11s
NBAC	2	69 845	40s	1	4 981	5s

Fixed-size model checking. In the first set of experiments, we solved several instances of the fixed-size model checking problem. That is, given the TLA+ encoding of the parameterized system $\mathsf{STS}(n,t,f)$, we assigned fixed values to the parameters n, t, and f, such as, e.g., $\mathsf{n} = 5, \mathsf{t} = 3$, and $\mathsf{f} = 2$. This results in a fixed-size system $\mathsf{STS}(\mathsf{n},\mathsf{t},\mathsf{f})$, which we give as an input to the model checker TLC. Table 3.1 summarizes the experimental results for fixed-size model checking, with system sizes of up to $\mathsf{n} = 5$ processes, where t is set to $\mathsf{n} - 1$, and $\mathsf{f} = \mathsf{t} - 1$. We also ran experiments for other values of t and f, but do not report on them here. We were able to obtain results for fixed-size model checking in the case where $\mathsf{n} = 5$, $\mathsf{t} = 4$, and $\mathsf{f} = 3$ for all our benchmarks, except for FloodMin, for $k = 2$. For larger values of the parameters, e.g., $\mathsf{n} = 5$, $\mathsf{t} = 4$, and $\mathsf{f} = 4$, we were only able to verify the simplest benchmark, NBAC. For the remaining benchmarks, we reached the limitations of the model checker, as TLC was not able to enumerate all possible successor states due to the immense branching.

This suggests that in order to verify the algorithms for systems with $\mathsf{n} > 5$ processes, one needs to rely on techniques such as abstractions, which give verification results for systems of all sizes.

Parameterized model checking. In the second set of experiments, we solved the parameterized model checking problem, by reducing it to the finite-state model checking problem. That is, we ran the model checker TLC with the TLA+ encoding of the abstract system $\widehat{\mathsf{STS}}^*$ as input. The TLA+ encoding of $\widehat{\mathsf{STS}}^*$ is obtained by specifying the abstract transition relations $\widehat{Env}$, $\widehat{Snd}$, and $\widehat{Upd}$, as defined in Section 3.5.

Further, in our experiments, we assume that the fixed m processes are correct, which captures the cases where $\mathsf{n} - \mathsf{m} > \mathsf{t} \geq \mathsf{f}$, for every $\mathsf{n},\mathsf{t},\mathsf{f} \in \mathbb{N}$. The results of these experiments are shown in Table 3.2 on the left. To capture the corner cases $\mathsf{n} > \mathsf{t} > \mathsf{n} - \mathsf{m}$ required by the resilience condition $n > t \geq f$, we also do experiments for abstract system where m' processes are fixed, which we assume are correct. We ran these experiments for every m', with $0 < \mathsf{m}' < \mathsf{m}$, and report on their results in Table 3.2 on the right. We

observe that most of the verification time is spent when checking the specification of $\widehat{\mathsf{STS}}^*$ with m fixed processes.

By comparing the experimental results for parameterized model checking and fixed-size model checking, we observe that parameterized model checking outperforms fixed-size model checking already in the case where $\mathsf{n} = 5$, $\mathsf{t} = 4$, and $\mathsf{f} = 3$.

For both parameterized and fixed-size model checking, FloodMin for $k = 2$ is the most challenging benchmark. In the case of fixed-size model checking, the model checker terminated after three days with an out of memory error when checking $\mathsf{STS}(\mathsf{n}, \mathsf{t}, \mathsf{f})$ for $\mathsf{n} = 5$, $\mathsf{t} = 4$, and $\mathsf{f} = 3$. In the parameterized model checking case, for the abstract system $\widehat{\mathsf{STS}}^*$ of FloodMin, we need to fix $\mathsf{m} = 3$ processes. This is due to the fact that one of its properties, when expressed in the indexed-LTL fragment $\mathcal{F}(k)$, has three leading universal quantifiers, which implies that we have to fix $\mathsf{m} = 3$ processes (in contrast to $\mathsf{m} = 2$ for the other benchmarks). Fixing $\mathsf{m} = 3$ processes in the abstract system introduces a greater amount of non-determinism, and therefore a larger state space that TLC explores.

3.7 Discussion

In this chapter, we proposed a symmetry argument and defined three abstraction mappings, which together allow us to reduce parameterized model checking for a given parameterized system $\mathsf{STS}(n, t, f)$ and an indexed-LTL formula ϕ from the fragment $\mathcal{F}(k)$ to finite-state model checking of an abstract system $\widehat{\mathsf{STS}}$ and an indexed-LTL formula ϕ^{m} from the fragment $\mathcal{F}^{\mathsf{m}}(k)$. We proposed a constructive definition of the abstract system, and applied the model checker TLC, we automatically verified several algorithms from the literature, most of which were not automatically verified before.

While synchronous distributed algorithms are considered "simpler" to design and understand than asynchronous ones, encoding and model checking synchronous algorithms is a challenge. In our experiments, we noticed that synchronously selecting a successor state in the abstract system results in a huge branching factor. This is due to the fact that all processes take steps simultaneously, and each process can transfer into several successor states depending on the received messages, which are subject to non-determinism by the environment. Further, additional non-determinism is introduced through the abstraction. This posed a serious challenge for the explicit-state model checker TLC, and required running our experiments on a machine with a lot of computing power. This inspired us to explore other formalization and model checking approaches, which we will discuss in the subsequent chapters.

In the context of parameterized model checking, abstraction is a well-known technique for dealing with undecidability. Different domain-specific abstractions have been used for mutual exclusion [PXZ02, CTV06, CTV08], cache coherence [CMP04, OTT09, McM01, Krs05], dynamic scheduling [McM99], and recently to asynchronous fault-tolerant al-

gorithms [JKS+13, KVW17, AGOP16]. Most of these parameterized model checking techniques consider asynchronous systems.

Several other approaches exist for verifying round-based distributed algorithms, i.e., algorithms whose executions are organized in rounds. The fixed-size model checking problem was addressed, e.g., in [SRSP04, CCM09, TS11], where (small) instances of an algorithm were verified using model checking, The following two approaches for parameterized verification are most related to our approach, as both target the round-based model from [CS09] and focus on partially synchronous algorithms: [DHV+14] proposes invariant checking using decision procedures, requiring the user to provide invariants manually. [MSB17] gives a cut-off theorem for reducing the parameterized problem to verification of small systems (5 to 7 processes). As we focus on synchronous algorithms, we have a different set of benchmarks compared to the work in [DHV+14, MSB17].

The only exception is FloodMin, for $k = 1$, which is considered in [DHV+14]. In contrast to our model checking approach, to verify FloodMin, for $k = 1$, the approach in [DHV+14] checked 5 user-provided verification conditions, such as invariants or ranking functions, in less than a second. Our approach has a higher degree of automation, as we do not require the verification engineer to provide invariants.

The cut-off results of [MSB17] target at completely automated verification. To achieve this, the authors had to restrict the fragment to which the cut-off theorem applies. First, the cut-off only applies to consensus algorithms, that is, to the three specific properties (recall Section 1.6.1). As noted in [MSB17], generalizing this to other properties, e.g., those for k-set agreement, non-blocking atomic commit (recall Section 1.6.2, 1.6.3), or even a more complete logic fragment would require more theoretical work. Our benchmarks, discussed in Section 3.6, in addition to consensus algorithms, include k-set agreement, non-blocking atomic commit algorithms. Second, [MSB17] introduced a guarded assignment language that can express only threshold guards containing predicates on the number of messages received by a process in the current round. However, there are several round-based distributed algorithms, in particular synchronous ones, that contain other guards; for instance, *termination guards* that check whether a given round number is reached, or guards that check whether messages from the same set of processes are received in two consecutive rounds. Our guarded assignments capture these guards.

The predicate abstraction step currently requires some domain knowledge to capture the interplay of the number of faults and round number. One possible direction for future work is automatic generation of predicates and verification conditions for the environment. This would automate the only step in our technique where manual intervention is needed, as all other abstraction steps introduced in this chapter can be automated. Another topic that can be studied in the future is extending our formalization and abstraction methods to other syntactic constructs, such as, e.g., threshold guards. Finally, more complex resilience conditions that appear in the literature, such as $n > 2t$ for send omission faults, or $n > 3t$ for Byzantine faults, would require a finer abstraction than the one presented in this chapter.

CHAPTER 4

Synchronous Threshold Automata

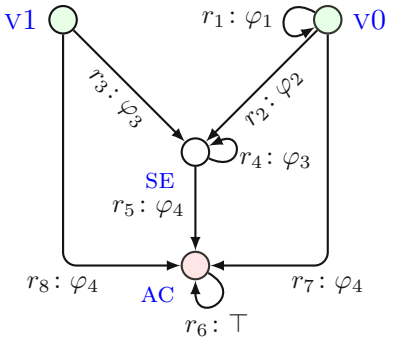
In Chapter 2, we introduced process and environment specifications that required minimal manual effort for encoding the process behavior described by pseudocode. However, these process and environment specifications were tailored to algorithms tolerating crash faults, and moreover, in order to apply parameterized verification techniques to the synchronous system specification, we needed to apply an abstraction, defined in Chapter 3. In this chapter, we introduce synchronous threshold automata as a formalism for modeling the process behavior and the assumptions imposed by the environment. Synchronous threshold automata are the *synchronous* variant of threshold automata [KVW17], which were introduced to model, verify, and synthesize *asynchronous* fault-tolerant distributed algorithms [KLVW17, LKWB17, BKLW19]. They will model the process behavior in a more abstract way, and will allow us to encode algorithms tolerating other kinds of faults, such as Byzantine faults, in addition to crash faults.

Before we formally define synchronous threshold automata, we demonstrate how we use them to model process behaviors on an example. Consider Figure 4.1, which shows the pseudocode of the authenticated broadcast algorithm SAB and its corresponding synchronous threshold automaton.

The pseudocode of SAB describes the behavior of a process running the algorithm. This algorithm is designed to tolerate Byzantine-faulty processes, and its resilience condition is $n > 3t \wedge t \geq f$. In a system of n processes running SAB, such that f processes are faulty, but not more than t are faulty, where $\mathsf{n}, \mathsf{t}, \mathsf{f} \in \mathbb{N}$ are values assigned to the parameters n, t, f, with $\mathsf{n} > 3\mathsf{t} \wedge \mathsf{t} \geq \mathsf{f}$, the processes perform the following actions in lock-step:

1. send a message ECHO if $\mathsf{v} = 1$ (line 5),
2. receive messages from all other processes (line 6),

```
1  v := input({0,1})
2  accept := ⊥
3  while true do {
4    if v = 1 then
5      broadcast ECHO
6    receive messages from other processes
7    if received ECHO from ≥ t + 1 processes then
8      v := 1
9    if received ECHO from ≥ n − t processes then
10     accept := ⊤
11 }
```



$$\varphi_1 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} < t+1 \qquad \varphi_2 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \geq t+1-f$$
$$\varphi_3 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} < n-t \qquad \varphi_4 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \geq n-t-f$$

Figure 4.1: Pseudocode of the authenticated broadcast algorithm SAB, its synchronous threshold automaton, and its guards.

3. update the variables v and `accept` based on the received messages (lines 7 to 10)

The synchronous threshold automaton that encodes the process behavior described by the pseudocode is depicted as a directed graph in Figure 4.1. Its nodes are called *locations*, and its edges are called *rules*. It models the loop body of the loop at line 3. One iteration of the loop is expressed as a guarded rule in the automaton, that connects the locations before and after an iteration.

The locations encode the current values of the control variables of a process (as defined in Definition 2.1 on page 38) as well as the value of the program counter. The automaton in Figure 4.1 has four locations: V0, V1, SE, AC. The location V0 encodes that the value of v in line 1 is set to 0, while the location V1 encodes that v is set to 1 in line 1. The location SE encodes that $\texttt{v} = 1$ and that $\texttt{accept} = \bot$, and the location AC encodes that $\texttt{v} = 1$ and that $\texttt{accept} = \top$.

We observe that a process sends a message ECHO, if the value of its variable v is 1 (lines 4 to 5). That is, in the synchronous threshold automaton, processes send a message ECHO if they are in one of the locations V1, SE, or AC. To test how many processes have sent a message ECHO in a system of n processes running SAB, it suffices to *count* how many processes are in one of the locations V1, SE, or AC.

However, the pseudocode of SAB contains conditions over the number of *received* messages (lines 7 and 9), rather than the number of *sent* messages. In synchronous systems, all messages sent by correct processes in a given round are received by all correct processes in the same round. On the contrary, not all messages sent by faulty processes are received by all correct processes. In the case of SAB, as the Byzantine-faulty processes may send spurious messages to some correct processes, the number of received messages may deviate from the number of correct processes that sent a message. For example, consider

a system where the processes run SAB, and the parameters n, t, and f have values n, t, and f, that satisfy the resilience condition. If the guard in line 7 evaluates to true for some process i, the $\mathsf{t}+1$ received messages may contain up to f messages from faulty processes. That is, if the number of correct processes that send ECHO is between 1 and t, the messages sent by faulty processes may "help" some correct processes to pass over the $\mathsf{t}+1$ threshold. In the synchronous threshold automaton, this is modeled by both the rules r_1 and r_2 being enabled. Thus, the assignment $\mathsf{v} := 1$ in line 8 is modeled by the rule r_2, which is guarded by $\varphi_2 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \geq t+1-f$. The implicit "else" branch between lines 7 and 9 is modeled by the rule r_1, guarded by $\varphi_1 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} < t+1$.

The locations and the rules of the synchronous threshold automaton are formally defined in Section 4.1. Together, they represent the process specification. The environment specification is given by an *environment assumption* of the synchronous threshold automaton, which is a linear integer arithmetic formula that imposes constraints on the number of processes allowed to populate certain locations. For example, as SAB tolerates Byzantine faults, and as the effect of the f Byzantine-faulty processes on the correct processes is captured by the guards, the synchronous threshold automaton in Figure 4.1 will be used to model the behavior only the correct processes. That is, the environment assumption ensures that the number of processes in locations $\text{V0}, \text{V1}, \text{SE}, \text{AC}$ is $\mathsf{n}-\mathsf{f}$. Additionally, the automata used to model algorithms that tolerate different kinds of faults have a different shape. We discuss the way in which we model faults in Section 4.1.2.

In Section 4.2, we introduce the system specification for a synchronous threshold automaton STA, given as a process and environment specification. That is, we introduce a *parameterized counter system*, $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$, where $\boldsymbol{\pi}$ is a vector of parameters, used to represent the infinite family $\{\mathsf{CS}(\mathsf{STA}, \mathbf{p}) \mid \mathbf{p} \text{ is a valuation of } \boldsymbol{\pi}\}$ of finite-state *counter systems* $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$. For a valuation $\mathbf{p}$ of the parameter vector $\boldsymbol{\pi}$, the configurations of the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ store a *counter* $\boldsymbol{\kappa}[\ell]$ for each location ℓ, that counts the number of processes that are in the location ℓ. Every transition in $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ moves all processes simultaneously, potentially by using a different rule for each process, provided that the rule guards evaluate to true.

Finally, in Section 4.3, we will define the logic that we will use to formalize the properties of an algorithm whose pseudocode we modeled with using a synchronous threshold automaton.

4.1 Process and Environment Specification: Synchronous Threshold Automaton

We introduce the notion of synchronous threshold automaton, as an object that is used to specify both the process behavior and the assumptions imposed by the environment.

Definition 4.1 (Synchronous threshold automaton)**.** A *synchronous threshold automaton* is the tuple $\mathsf{STA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Pi, RC, \mathsf{Env})$, where:

- $\mathcal{L}$ is a finite set of *locations*,
- $\mathcal{I} \subseteq \mathcal{L}$ is a non-empty set of *initial locations*,
- $\mathcal{R}$ is a finite set of *rules*,
- Π is a finite set of *parameters*,
- RC is a *resilience condition*,
- Env is an *environment assumption*. □

The locations $\mathcal{L}$, initial locations $\mathcal{I}$, and rules $\mathcal{R}$ constitute the process specification. The environment specification is defined by the environment assumption Env. We discuss each of the components of the synchronous threshold automaton in detail below.

We assume that the set Π of *parameters* contains at least the parameter n, denoting the total number of processes. The *resilience condition* RC is an expression over the parameters from the set Π.

Definition 4.2 (Parameter vector)**.** We call the vector $\boldsymbol{\pi} = \langle \pi_1, \ldots, \pi_{|\Pi|} \rangle$ the *parameter vector*, where each $\pi_i \in \Pi$ is a parameter, for $1 \leq i \leq |\Pi|$.

The vector $\mathbf{p} = \langle \mathsf{p}_1, \ldots, \mathsf{p}_{|\Pi|} \rangle \in \mathbb{N}^{|\Pi|}$ is called a *valuation of* $\boldsymbol{\pi}$. We denote by $\mathbf{p}[\pi_i] = \mathsf{p}_i$, for $\mathsf{p}_i \in \mathbb{N}$ and $1 \leq i \leq |\Pi|$, the value assigned to the parameter π_i in the valuation $\mathbf{p}$. □

We will use the parameter vector $\boldsymbol{\pi}$ to define the syntax of the guards that the processes use to move from one location to another in the synchronous threshold automaton. The resilience condition RC imposes conditions on the values of the parameters from the set Π, and thus defines which valuations $\mathbf{p} \in \mathbb{N}^{|\Pi|}$ of $\boldsymbol{\pi}$ are admissible.

Definition 4.3 (Admissible valuations)**.** The set of *admissible valuations of* $\boldsymbol{\pi}$ is denoted by $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}^{|\Pi|} \mid \mathbf{p} \text{ is a valuation of } \boldsymbol{\pi} \text{ and } \mathbf{p} \text{ satisfies } RC\}$. □

For some fault models, we will use synchronous threshold automata to model the behavior of the correct processes only. That is, in some cases we will not capture the behavior of the faulty processes explicitly, but rather capture their influence on the correct processes as a part of the environment, as we will see in Section 4.1.2. To define the number of processes whose behavior is modeled using the synchronous threshold automaton, we define the following mapping.

Definition 4.4 (Participating processes)**.** The mapping $N : \mathbf{P}_{RC} \to \mathbb{N}$ maps an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$ to the number $N(\mathbf{p}) \in \mathbb{N}$ of *participating processes*, i.e., the number of processes whose behavior is modeled using the STA. We denote by $N(\boldsymbol{\pi})$ the linear combination of parameters that defines the number of participating processes. □

Example 4.1. For the synchronous threshold automaton in Figure 4.1, we have $\Pi = \{n, t, f\}$ and $RC \equiv n > 3t \wedge t \geq f$, hence a vector $\mathbf{p} \in \mathbb{N}^{|\Pi|}$ is an admissible valuation of the parameter vector $\boldsymbol{\pi} = \langle n, t, f \rangle$, i.e., $\mathbf{p} \in \mathbf{P}_{RC}$, if $\mathbf{p}[n] > 3\mathbf{p}[t] \wedge \mathbf{p}[t] \geq \mathbf{p}[f]$. Furthermore, we have $N(\boldsymbol{\pi}) = n - f$, hence for an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$, we have $N(\mathbf{p}) = \mathbf{p}[n] - \mathbf{p}[f]$. As there are $n - f$ correct processes in the system, this means that the synchronous threshold automaton in Figure 4.1 models the behavior of the correct processes that run the algorithm SAB.

4.1.1 Process Specification: Locations and Rules

The *locations* $\ell \in \mathcal{L}$ encode the current value of the process control variables x that store values local to the process, such as, e.g., an initial value or a decision value; together with information about the program counter. Similarly to Definition 2.2 on page 39, we assume that each process control variable x ranges over a finite set of values. This, together with the fact that the program counter takes values from a fixed and finite set, implies that the set $\mathcal{L}$ of locations is a finite set. Contrary to the process local states defined in Chapter 2, the locations of a synchronous threshold automaton do not contain neighborhood variables, and are not parameterized by the number n of processes.

The *initial locations* in $\mathcal{I} \subseteq \mathcal{L}$ encode the initial values of the process control variables.

Let $\mathcal{M}$ denote the set of *message types*. To encode sending messages in the synchronous threshold automaton, we define a mapping $\mathsf{sent} : \mathcal{M} \to 2^{\mathcal{L}}$, that maps message types to sets of locations. The mapping sent captures the semantics of the synchronous computation model, where the processes send messages based on their location.

Definition 4.5 (Sending messages)**.** The mapping $\mathsf{sent} : \mathcal{M} \to 2^{\mathcal{L}}$ maps a message type $m \in \mathcal{M}$ to a set $\mathsf{sent}(m) \subseteq \mathcal{L}$, such that:

$$\mathsf{sent}(m) = \{\ell \in \mathcal{L} \mid \text{a process in } \ell \text{ sends message of type } m\}$$

We assume that the sets $\mathsf{sent}(m) \subseteq \mathcal{L}$, for $m \in \mathcal{M}$, are pairwise mutually disjoint, that is:

$$\text{for every } m_1, m_2 \in \mathcal{M} \;\; m_1 \neq m_2 \quad \text{implies} \quad \mathsf{sent}(m_1) \cap \mathsf{sent}(m_2) = \emptyset \qquad \square$$

The requirements that the sets $\mathsf{sent}(m) \subseteq \mathcal{L}$, for $m \in \mathcal{M}$, are pairwise mutually disjoint captures the fact that a process in a given location cannot send messages of two (or more) different types.

Let $L \subseteq \mathcal{L}$ denote a set of locations, and let $\#L$, for $0 \leq \#L \leq N(\mathbf{p})$, denote the number of processes in locations from the set L. For the purpose of defining guards and expressing properties, we define *c-propositions*.

Definition 4.6 (Counter propositions)**.** We define c-propositions, which are expressions of the form:

$$\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \text{ for } L \subseteq \mathcal{L}, \boldsymbol{a} \in \mathbb{Z}^{|\Pi|}, \text{ and } b \in \mathbb{Z}$$

We will use two abbreviations: $\#L = \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ for the formula $(\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b) \wedge \neg(\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b + 1)$, and $\#L > \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ for the formula $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b + 1$.

We denote by CP the set of c-propositions. □

The intended meaning of the c-propositions is to check wether the number of processes currently in locations $\ell \in L$ is greater than or equal to a linear combination of the parameters, also called a *threshold*. When the set L of locations in the c-proposition is equal to the set $\mathsf{sent}(m)$, for some $m \in \mathcal{M}$, the c-proposition is used to check whether the number of sent messages of type $m \in \mathcal{M}$ passes the given threshold.

We now define the *rules* $r \in \mathcal{R}$, which describe how the processes move from one location to another.

Definition 4.7 (Rule)**.** A *rule* $r \in \mathcal{R}$ is a tuple $(\mathit{from}, \mathit{to}, \varphi)$, where:

- $\mathit{from} \in \mathcal{L}$ is an *origin location*,
- $\mathit{to} \in \mathcal{L}$ is a *goal location*,
- φ is a *guard*, i.e., a Boolean combination of c-propositions. □

The guards $r.\varphi$, for $r \in \mathcal{R}$, are evaluated in tuples $(\boldsymbol{\kappa}, \mathbf{p})$, where $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of *counters*, such that for a location $\ell \in \mathcal{L}$, the counter $\boldsymbol{\kappa}[\ell]$ denotes the number of processes that are currently in the location ℓ, and $\mathbf{p} \in \mathbf{P}_{RC}$. We define the formal semantics of the c-propositions, the semantics of the Boolean connectives is standard.

Definition 4.8 (Semantics of c-propositions)**.** Given a tuple $(\boldsymbol{\kappa}, \mathbf{p})$, where $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of counters, and $\mathbf{p} \in \mathbf{P}_{RC}$ is an admissible valuation of $\boldsymbol{\pi}$, the formal semantics of c-propositions is defined as follows:

$$(\boldsymbol{\kappa}, \mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{iff} \quad \sum_{\ell \in L} \boldsymbol{\kappa}[\ell] \geq \boldsymbol{a} \cdot \mathbf{p} + b \qquad \square$$

Example 4.2. The set $\mathcal{L} = \{\text{V0}, \text{V1}, \text{SE}, \text{AC}\}$ is the set of locations of the synchronous threshold automaton in Figure 4.1, and $\mathcal{I} = \{\text{V0}, \text{V1}\}$ is the set of initial locations. There is only one message type, $\mathsf{ECHO} \in \mathcal{M}$, and we define $\mathsf{sent}(\mathsf{ECHO}) = \{\text{V1}, \text{SE}, \text{AC}\}$, since a process in either V1, SE, or AC has a value 1, and sends a message ECHO, as described in the pseudocode.

The set $\mathcal{R}$ of rules contains the rules $r_1, \ldots, r_8$, which are guarded by the guards $\varphi_1, \ldots, \varphi_4$, also depicted in Figure 4.1. These guards check if enough ECHO messages have been sent by the processes, as the set $L = \{\text{V1}, \text{SE}, \text{AC}\}$ of locations occurring in the c-propositions is equal to the set $\mathsf{sent}(\mathsf{ECHO}) = \{\text{V1}, \text{SE}, \text{AC}\}$. □

4.1.2 Environment Specification: Environment Assumption and Modeling Faults

The *environment assumption* Env is a conjunction of c-propositions and their negations, where each c-proposition $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ occurring in Env restricts the number of processes allowed to populate the locations in $L \subseteq \mathcal{L}$. In addition to constraints imposed by the fault model, which we will discuss below, the environment assumption Env contains the following conjuncts:

($\mathsf{C1}$) $\#\{\ell\} \geq 0$, for each $\ell \in \mathcal{L}$, which states that the number of processes in a location ℓ is non-negative, and

($\mathsf{C2}$) $\#\mathcal{L} = N(\boldsymbol{\pi})$, which states that the number of processes that are allowed to populate the locations from the $\mathcal{L}$ is equal to the number of participating processes.

Thus, we define the environment assumption Env as the formula:

$$\mathsf{Env} \equiv \mathsf{C1} \wedge \mathsf{C2} \wedge \mathsf{Env}_{\mathrm{CP},*}$$

where the formulas $\mathsf{Env}_{\mathrm{CP},*}$ for $* \in \{\mathsf{cr}, \mathsf{so}, \mathsf{byz}\}$, depend on the fault model, i.e., on whether we model crash, send omission, or Byzantine faults.

Further, when constructing a synchronous threshold automaton that models the behavior of a process running a given algorithm, we typically need to introduce additional locations or rules that depend on the fault model, and that are used to capture its semantics. Thus, in the following, for each of the three fault models we introduced in Section 1.2, that is, for crash, send omission, and Byzantine faults, we propose a modeling step that precisely captures the faulty semantics. The synchronous threshold automata for algorithms tolerating hybrid faults, such as, e.g., send omission and Byzantine faults, can be obtained by combining the environment specifications for send omission and Byzantine faults.

Modeling Crash Faults. To model the behavior of the crash-faulty processes, the set $\mathcal{L}$ of locations of the synchronous threshold automaton is the set:

$$\mathcal{L} = \mathcal{L}_{\mathsf{corr}} \cup \mathcal{L}_{\mathsf{cr}} \cup \{\ell_{\text{✖}}\}$$

where $\mathcal{L}_{\mathsf{corr}}$ is a set of *correct* locations, $\mathcal{L}_{\mathsf{cr}} = \{\ell_{\mathsf{cr}} \mid \ell_{\mathsf{cr}} \text{ is a fresh copy of } \ell \in \mathcal{L}_{\mathsf{corr}}\}$ is a set of *crash* locations, and $\ell_{\text{✖}}$ is a *failed* location. The crash locations $\ell_{\mathsf{cr}} \in \mathcal{L}_{\mathsf{cr}}$ model the same values of the control variables and program counter as their correct counterpart $\ell \in \mathcal{L}_{\mathsf{corr}}$. The only difference is that the processes in the crash locations $\ell_{\mathsf{cr}} \in \mathcal{L}_{\mathsf{cr}}$ are flagged by the environment to crash in the current round. After crashing, the processes move to the failed location $\ell_{\text{✖}}$, where they remain forever. This models that the crashed processes cannot restart.

Recall that a crash-faulty process may send a message to a subset of the other processes in the round in which it crashes. To model this, we first introduce the mapping $\mathsf{sent}_{\mathsf{cr}}$:

$\mathcal{M} \to 2^{\mathcal{L}_{\mathsf{cr}}}$, which defines, for each message type $m \in \mathcal{M}$, the set of crash locations $\mathsf{sent}_{\mathsf{cr}}(m) \subseteq \mathcal{L}_{\mathsf{cr}}$ where processes send a message of type m. Then, the c-proposition used to check the total number of sent messages of type $m \in \mathcal{M}$ is the expression $\#(\mathsf{sent}(m) \cup \mathsf{sent}_{\mathsf{cr}}(m)) \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$. Similarly, the c-proposition that checks the total number of messages of type $m \in \mathcal{M}$ sent by *correct* processes is the expression $\#\mathsf{sent}(m) \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$.

Introducing new locations implies that new rules are added in the automaton as well:

(cr1) for every rule $r \in \mathcal{R}$, if $r.from \in \mathcal{L}_{\mathsf{corr}}$ and $r.to \in \mathcal{L}_{\mathsf{corr}}$, then we add the rule $(r.from, \ell_{\mathsf{cr}}, r.\varphi)$, where $\ell_{\mathsf{cr}} \in \mathcal{L}_{\mathsf{cr}}$ is the crash location corresponding to the location $r.to$,

(cr2) for every crash location $\ell_{\mathsf{cr}} \in \mathcal{L}_{\mathsf{cr}}$, we add the rule $(\ell_{\mathsf{cr}}, \ell_{\maltese}, \top)$,

(cr3) for the failed location $\ell_{\maltese}$, we add the rule $(\ell_{\maltese}, \ell_{\maltese}, \top)$.

The rules (cr1) are used by the processes to move from the correct to the crash locations, in the rounds where the environment flags them as crashed. The rules (cr2) move the processes from the crashed locations to the failed location, where they can only apply the self-loop rule (cr3), which keeps them in the failed location.

By introducing new locations and rules, we can capture the behavior of the crash-faulty processes in the synchronous threshold automaton. That is, we model the behavior of crash-faulty processes explicitly, and define the number $N(\boldsymbol{\pi})$ of participating processes equal to the total number n of processes, i.e., $N(\boldsymbol{\pi}) = n$.

Finally, the environment constraint $\mathsf{Env}_{\mathrm{CP},\mathsf{cr}}$ specific to crash faults is:

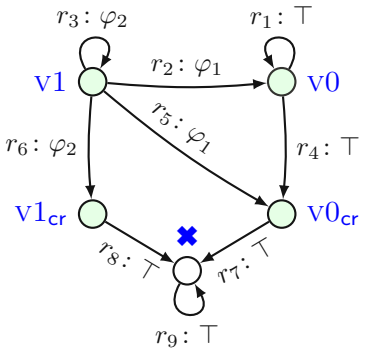
$$\mathsf{Env}_{\mathrm{CP},\mathsf{cr}} \equiv \#(\mathcal{L}_{\mathsf{cr}} \cup \{\ell_{\maltese}\}) \leq f \tag{4.1}$$

and ensures that there are no more than f faults.

Example 4.3. Recall the FloodMin algorithm, whose pseudocode is given in Figure 2.1 on page 40. The synchronous threshold automaton that models the loop body of the algorithm is depicted in Figure 4.2. The algorithm is designed to tolerate crash faults, and thus in the synchronous threshold automaton used to model it we have correct locations $\mathcal{L}_{\mathsf{corr}} = \{\mathrm{v0}, \mathrm{v1}\}$, crash locations $\mathcal{L}_{\mathsf{cr}} = \{\mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}\}$, and a failed location ✖. The locations v0 and v1 encode that the value of the variable `best` of the process is 0 and 1, and their corresponding crash locations are $\mathrm{v0}_{\mathsf{cr}}$ and $\mathrm{v1}_{\mathsf{cr}}$, respectively. As the variable `dec` is assigned a value outside of the loop body (line 8 of the pseudocode), and the synchronous threshold automaton models only the loop body, the locations of the automaton do not encode the value of the variable `dec`.

The set $\mathcal{M}$ of message types contains the message types m_0 and m_1, used to represent the messages 0 and 1, respectively, and we define:

$$\mathsf{sent}(m_0) = \{\mathrm{v0}\} \qquad \mathsf{sent}_{\mathsf{cr}}(m_0) = \{\mathrm{v0}_{\mathsf{cr}}\}$$
$$\mathsf{sent}(m_1) = \{\mathrm{v1}\} \qquad \mathsf{sent}_{\mathsf{cr}}(m_1) = \{\mathrm{v1}_{\mathsf{cr}}\}$$



$$
\begin{array}{rcl}
\mathcal{L} & = & \{\mathrm{v0}, \mathrm{v1}, \mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}, ✖\} \\
\mathcal{I} & = & \{\mathrm{v0}, \mathrm{v1}, \mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}\} \\
\mathcal{R} & = & \{r_1, \ldots, r_9\} \\
\varphi_1 & \equiv & \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \geq 1 \\
\varphi_2 & \equiv & \#\{\mathrm{v0}\} < 1 \\
\Pi & = & \{n, t, f\} \\
RC & \equiv & n > t \wedge t \geq f \\
\mathsf{Env} & \equiv & \bigwedge_{\ell \in \mathcal{L}} \#\{\ell\} \geq 0 \\
 & & \wedge\ \#\mathcal{L} = n \\
 & & \wedge\ \#\{\mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}, ✖\} \leq f
\end{array}
$$

Figure 4.2: The synchronous threshold automaton encoding the loop body of the FloodMin algorithm for $k = 1$, whose pseudocode is given in Figure 2.1, and which tolerates crash faults.

Observe that in the pseudocode of FloodMin, the processes update their value `best` with the smallest received value. The correct processes that are in location v0 can only stay in v0, which is captured by the rule r_1. The correct processes that are in location v1 can either:

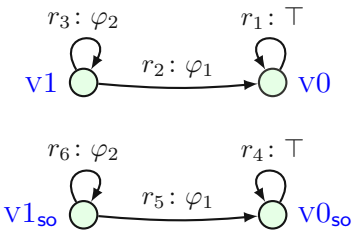
1. move to the location v0, if there is at least one message of type m_0 sent either by a correct or by a faulty process, which is captured by the rule r_2, whose guard φ_1 checks if there is at least one process in the locations from the set $\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} = \mathsf{sent}(m_0) \cup \mathsf{sent}_{\mathsf{cr}}(m_0)$,
2. stay in the location v1, if there is no message of type m_0 by sent by a correct process, which is captured by the rule r_3, whose guard φ_2 checks if there is no process in the locations from the set $\{\mathrm{v0}\} = \mathsf{sent}(m_0)$.

Both the guards φ_1 and φ_2 can be satisfied in location v1. This is used to model the non-determinism imposed by the crash-faulty processes sending messages to a subset of the other processes.

The rules r_4, r_5, and r_6, corresponding to r_1, r_2, and r_3, respectively, move processes from the correct to the crash locations (described in (cr1)). The rules r_7 and r_8 move the processes from the crashed locations to the failed location (described in (cr2)), and the rule r_9 keeps the processes in the failed location (described in (cr3)).

The environment assumption Env states that each location can be populated by a non-negative number of processes, that the total number of processes in all locations $\mathcal{L}$ is $N(\boldsymbol{\pi}) = n$, and that the number of processes in the crashed and failed locations $\{\mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}, ✖\}$ is at most f. □

Modeling Send Omission Faults. To model algorithms tolerating send omission faults, we proceed similarly to the way in which we modeled crash faults. The set $\mathcal{L}$



$$
\begin{array}{rcl}
\mathcal{L} & = & \{\mathrm{v0}, \mathrm{v1}, \mathrm{v0}_{\mathsf{so}}, \mathrm{v1}_{\mathsf{so}}\} \\
\mathcal{I} & = & \{\mathrm{v0}, \mathrm{v1}, \mathrm{v0}_{\mathsf{so}}, \mathrm{v1}_{\mathsf{so}}\} \\
\mathcal{R} & = & \{r_1, \ldots, r_6\} \\
\varphi_1 & \equiv & \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{so}}\} \geq 1 \\
\varphi_2 & \equiv & \#\{\mathrm{v0}\} < 1 \\
\Pi & = & \{n, t, f\} \\
RC & \equiv & n > t \wedge t \geq f \\
\mathsf{Env} & \equiv & \bigwedge_{\ell \in \mathcal{L}} \#\{\ell\} \geq 0 \\
 & & \wedge\ \#\mathcal{L} = n \\
 & & \wedge\ \#\{\mathrm{v0}, \mathrm{v1}\} = n - f \\
 & & \wedge\ \#\{\mathrm{v0}_{\mathsf{so}}, \mathrm{v1}_{\mathsf{so}}\} = f
\end{array}
$$

Figure 4.3: The synchronous threshold automaton encoding the loop body of the FloodMinOmit algorithm for $k = 1$, whose pseudocode is given in Figure 2.1, and which tolerates send omission faults.

of locations of a synchronous threshold automaton that models an algorithm tolerating send omission faults is the set:

$$\mathcal{L} = \mathcal{L}_{\mathsf{corr}} \cup \mathcal{L}_{\mathsf{so}}$$

where $\mathcal{L}_{\mathsf{corr}}$ is a set of *correct* locations and $\mathcal{L}_{\mathsf{so}} = \{\ell_{\mathsf{so}} \mid \ell_{\mathsf{so}}$ is a fresh copy of $\ell \in \mathcal{L}_{\mathsf{corr}}\}$ is a set of *send-omission* locations. That is, for each correct location $\ell \in \mathcal{L}_{\mathsf{corr}}$ there exists a single send-omission location $\ell_{\mathsf{so}} \in \mathcal{L}_{\mathsf{so}}$, which is a copy of ℓ. Further, for every rule $r \in \mathcal{R}$ connecting two correct locations $\ell, \ell' \in \mathcal{L}_{\mathsf{corr}}$, there exists a rule $(\ell_{\mathsf{so}}, \ell'_{\mathsf{so}}, r.\varphi) \in \mathcal{R}$, connecting their two corresponding send-omission locations $\ell_{\mathsf{so}}, \ell'_{\mathsf{so}} \in \mathcal{L}_{\mathsf{so}}$. No additional rules are introduced, that is, no rules connect the correct locations $\ell \in \mathcal{L}_{\mathsf{corr}}$ to the send-omission locations $\ell_{\mathsf{so}} \in \mathcal{L}_{\mathsf{so}}$.

The automaton thus consists of two parts: one used by the correct processes, and one used by the send-omission faulty processes. The behavior of the send-omission faulty processes is encoded explicitly, using locations and rules in the automaton, hence, as in the crash fault model, we define $N(\boldsymbol{\pi}) = n$. The environment constraint $\mathsf{Env}_{\mathsf{so}}$, specific to send omission faults, is:

$$\mathsf{Env}_{\mathsf{so}} \equiv (\#\mathcal{L}_{\mathsf{corr}} = n - f) \wedge (\#\mathcal{L}_{\mathsf{so}} = f) \tag{4.2}$$

and ensures that the number of processes populating the correct locations is $n - f$, and the number of processes populating the send-omission locations is f.

The two mappings $\mathsf{sent} : \mathcal{M} \to 2^{\mathcal{L}_{\mathsf{corr}}}$ and $\mathsf{sent}_{\mathsf{so}} : \mathcal{M} \to 2^{\mathcal{L}_{\mathsf{so}}}$ define for a given message type $m \in \mathcal{M}$, the set of correct and send-omission locations, respectively, where processes send a message of type m.

Example 4.4. We give an example on how we model send omission faults by showing how we model the algorithm FloodMinOmit, which is a variant of FloodMin (and thus has the same pseudocode) which tolerates send omission faults. The synchronous

threshold automaton that models the loop body of the algorithm is given in Figure 4.3. Its set of locations is partitioned into the set $\mathcal{L}_{\mathsf{corr}} = \{\mathrm{v0}, \mathrm{v1}\}$ of correct and the set $\mathcal{L}_{\mathsf{so}} = \{\mathrm{v0}_{\mathsf{so}}, \mathrm{v1}_{\mathsf{so}}\}$ of send-omission locations. The whole automaton in fact is partitioned into two identical parts: one populated by the correct, and the other by the send-omission-faulty processes.

The set $\mathcal{M}$ of message types contains the message types m_0 and m_1, and we define

$$\mathsf{sent}(m_0) = \{\mathrm{v0}\} \qquad \mathsf{sent}_{\mathsf{so}}(m_0) = \{\mathrm{v0}_{\mathsf{so}}\}$$
$$\mathsf{sent}(m_1) = \{\mathrm{v1}\} \qquad \mathsf{sent}_{\mathsf{so}}(m_1) = \{\mathrm{v1}_{\mathsf{so}}\}$$

A process in location v1 (resp. $\mathrm{v1}_{\mathsf{so}}$) can either:

- move to the location v0 (resp. $\mathrm{v0}_{\mathsf{so}}$) if at least one message of type m_0 was sent either by a correct or a send-omission-faulty process, captured by the rule r_2 (resp. r_5), whose guard φ_1 checks if there is at least one process in a location where it sends a message of type m_0, i.e., if there is at least one process in the locations $\{\mathrm{v0}, \mathrm{v0}_{\mathsf{so}}\} = \mathsf{sent}(m_0) \cup \mathsf{sent}_{\mathsf{so}}(m_0)$.
- stay in the location v1 (resp. $\mathrm{v1}_{\mathsf{so}}$) if no message of type m_0 was sent by a correct process, captured by the rule r_3 (resp. r_6), whose guard φ_2 checks if there is no correct process in a location where it sends a message of type m_0, i.e., if there is no process in the set $\{\mathrm{v0}\} = \mathsf{sent}(m_0)$ of correct locations.

Again, as in the crash fault model, the non-determinism imposed by the send-omission-faulty processes sending messages to a subset of the other processes is captured by both r_2 and r_3 (resp. r_5 and r_6) being enabled.

Finally, the environment assumption Env defines that the number $\#\mathcal{L}$ of processes in all locations is $N(\boldsymbol{\pi}) = n$, the number $\#\mathcal{L}_{\mathsf{corr}}$ of processes in the correct locations is $n - f$, and that the number $\#\mathcal{L}_{\mathsf{so}}$ of processes in the send-omission locations is f. □

Modeling Byzantine Faults. To model the behavior of the Byzantine-faulty processes, we do not introduce new locations and rules in the synchronous threshold automaton. Instead, the synchronous threshold automaton is used to model the behavior of the correct processes, while the effect that the Byzantine-faulty processes have on the correct ones is captured in the guards as follows.

Let $m \in \mathcal{M}$ be a message type, and $\mathsf{sent}(m) \subseteq \mathcal{L}$ the set of locations where correct processes send a message of type m. The c-proposition that checks if the total number of messages of type $m \in \mathcal{M}$ passes a given threshold is the expression $\#\mathsf{sent}(m) + f \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$. That is, the number f of faults is an upper bound on the number of messages of type m sent by the Byzantine-faulty processes. Similarly, a c-proposition that checks if the total number of messages of two message types $m_1, m_2 \in \mathcal{M}$ pass a given threshold is the expression $\#(\mathsf{sent}(m_1) \cup \mathsf{sent}(m_2)) + f \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$. In this case, the number f represents

the upper bound on the number of messages of types m_1 and m_2 sent by Byzantine-faulty processes.

The non-determinism introduced by the Byzantine-faulty processes is captured by having the following two kinds of rules outgoing from a location $\ell \in \mathcal{L}$ being enabled:

- a rule $r \in \mathcal{R}$, whose guard $r.\varphi$ checks if the total number of messages sent by both correct and Byzantine-faulty processes passes a certain threshold, that is, an expression of the form $\#\mathsf{sent}(m) + f \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$, for $m \in \mathcal{M}$,
- a rule $r' \in \mathcal{R}$, whose guard $r'.\varphi$ checks if the number of messages sent only by correct processes is below the same threshold, that is, an expression of the form $\#\mathsf{sent}(m) < \boldsymbol{a} \cdot \boldsymbol{\pi} + b$, for $m \in \mathcal{M}$.

As the synchronous threshold automaton is used to model the behavior of the correct processes, we have that $N(\boldsymbol{\pi}) = n - f$. Moreover, since we do not introduce new locations for faulty processes, we have $\mathsf{Env}_{\mathrm{CP},\mathsf{byz}} \equiv \top$, that is, the only constraints in the environment assumption Env are the constraints (C1) and (C2), which encode that the number of processes in each location is non-negative, and that the number of processes in all locations is equal to the number of participating processes, respectively.

Example 4.5. Recall the synchronous threshold automaton of the algorithm SAB which tolerates Byzantine faults, depicted in Figure 4.1. Observe that for the single message type m_{ECHO}, we have $\mathsf{sent}(m_{\mathsf{ECHO}}) = \{\mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$. Consider the rules r_4, r_5 outgoing of the location SE. Both rules r_4, r_5 can be enabled at the same time. The guard $r_5.\varphi$ of the rule r_5 checks if the total number $\#\{\mathrm{V1}, \mathrm{SE}, \mathrm{AC}\} + f$ of messages of type m_{ECHO}, sent by both correct and Byzantine-faulty processes, is at least $n - t$. The guard $r_4.\varphi$ of the rule r_4 checks if the number $\#\{\mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$ of messages sent by correct processes is less than $n - t$. By applying r_5, a process allows the messages sent by Byzantine-faulty processes to "help" it pass the threshold $n - t$. On the contrary, by applying the rule r_4, a process assumes that it has not received any message by the faulty processes in the case when there are not enough messages sent by the correct processes to pass the threshold $n - t$. Once there are at least $n - t$ messages sent by correct processes, this rule becomes disabled, and the process is forced to leave the location SE.

We do not add environment constraints specific to Byzantine faults in the environment assumption – that is, the environment assumption Env for the algorithm SAB is the formula:

$$\mathsf{Env} \equiv \bigwedge_{\ell \in \mathcal{L}} \#\{\ell\} \geq 0 \wedge \#\mathcal{L} = n - f \qquad \square$$

4.2 Synchronous System Specification: Counter System

Let $\mathsf{STA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Pi, RC, \mathsf{Env})$ be a synchronous threshold automaton, and $\mathbf{p} \in \mathbf{P}_{RC}$ an admissible valuation of the parameter vector $\boldsymbol{\pi}$. The counter system, defined below, represents a system of $N(\mathbf{p})$ processes, whose behavior is modeled using the STA.

Definition 4.9 (Counter system)**.** A *counter system* w.r.t. an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$ and an $\mathsf{STA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Pi, RC, \mathsf{Env})$ is the tuple $\mathsf{CS}(\mathsf{STA}, \mathbf{p}) = (\Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p}))$, where:

- $\Sigma(\mathbf{p})$ is the set of *configurations*,
- $I(\mathbf{p})$ is the set of *initial configurations*,
- $R(\mathbf{p})$ is the *transition relation*. □

The set $\Sigma(\mathbf{p})$ of configurations, the set $I(\mathbf{p})$ of initial configurations, and the transition relation $R(\mathbf{p})$ of the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ will be formally defined below, namely in Definitions 4.10, 4.11, and 4.15.

We denote by $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$ the *parameterized counter system*, which is used to represent the infinite family $\{\mathsf{CS}(\mathsf{STA}, \mathbf{p}) \mid \mathbf{p} \in \mathbf{P}_{RC}\}$ of finite-state counter systems.

Configurations. We now define the configurations and the initial configurations of a counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, for a given STA and an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$.

Definition 4.10 (Configurations $\Sigma(\mathbf{p})$)**.** A *configuration* $\sigma \in \Sigma(\mathbf{p})$ is a tuple $(\boldsymbol{\kappa}, \mathbf{p})$, where:

- $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of *counters*,
- $\mathbf{p} \in \mathbf{P}_{RC}$ is an admissible valuation of $\boldsymbol{\pi}$,

such that $\sigma \models \mathsf{Env}$. □

As a consequence of Definition 4.10, for every $\sigma \in \Sigma(\mathbf{p})$, we have $\sum_{\ell \in \mathcal{L}} \sigma.\boldsymbol{\kappa}[\ell] = N(\mathbf{p})$. This follows from $\sigma \models \mathsf{Env}$, in particular from $\sigma \models \#\mathcal{L} = N(\boldsymbol{\pi})$, the definition of $N(\mathbf{p})$, and the semantics of the c-propositions.

Definition 4.11 (Initial configurations $I(\mathbf{p})$)**.** A configuration $\sigma \in \Sigma(\mathbf{p})$ is *initial*, i.e., $\sigma \in I(\mathbf{p}) \subseteq \Sigma(\mathbf{p})$, iff $\sigma.\boldsymbol{\kappa}[\ell] = 0$, for every $\ell \in \mathcal{L} \setminus \mathcal{I}$. □

That is, the value $\sigma.\boldsymbol{\kappa}[\ell]$ of the counter for each non-initial location $\ell \in \mathcal{L} \setminus \mathcal{I}$ is set to 0 in every initial configuration $\sigma \in \mathcal{I}$. As a consequence of Definition 4.10 and 4.11, we have $\sum_{\ell \in \mathcal{I}} \sigma.\boldsymbol{\kappa}[\ell] = N(\mathbf{p})$.

Transition Relation. To define the transition relation $R(\mathbf{p})$, we first define the notion of a transition.

Definition 4.12 (Transition)**.** A *transition* is a function $tr : \mathcal{R} \to \mathbb{N}$ that maps each rule $r \in \mathcal{R}$ to a *factor* $tr(r) \in \mathbb{N}$.

Given a valuation $\mathbf{p}$ of $\boldsymbol{\pi}$, the set $Tr(\mathbf{p}) = \{tr \mid \sum_{r \in \mathcal{R}} tr(r) = N(\mathbf{p})\}$ contains transitions whose factors sum up to $N(\mathbf{p})$. □

For a transition tr and a rule $r \in \mathcal{R}$, the factor $tr(r)$ denotes the number of processes that act upon this rule. By restricting the set $Tr(\mathbf{p})$ to contain transitions whose factors sum up to $N(\mathbf{p})$, we ensure that in a transition, every process takes a step. This captures the semantics of synchronous computation.

To define how transitions are applied in configurations, we define the conditions that make a transition $tr \in Tr(\mathbf{p})$ *enabled* in a tuple $(\boldsymbol{\kappa}, \mathbf{p})$.

Definition 4.13 (Enabled transition)**.** Given a tuple $(\boldsymbol{\kappa}, \mathbf{p})$, where $\boldsymbol{\kappa}$ is an $|\mathcal{L}|$-dimensional vector of counters and $\mathbf{p} \in \mathbf{P}_{RC}$ an admissible valuation of $\boldsymbol{\pi}$, a transition $tr : \mathcal{R} \to \mathbb{N}$, with $tr \in Tr(\mathbf{p})$ is *enabled* in σ, iff:

1. for every $r \in \mathcal{R}$, such that $tr(r) > 0$, it holds that $(\boldsymbol{\kappa}, \mathbf{p}) \models r.\varphi$, and
2. for every $\ell \in \mathcal{L}$, we have $\boldsymbol{\kappa}[\ell] = \sum_{r \in \mathcal{R} \wedge r.from = \ell} tr(r)$. □

The first condition ensures that processes only use rules whose guards are satisfied, and the second that every process moves in an enabled transition.

Definition 4.14 (Origin and goal of a transition)**.** Given a transition $tr \in Tr(\mathbf{p})$, we define:

- the *origin* $o(tr)$ of tr, as the tuple $o(tr) = (\boldsymbol{\kappa}, \mathbf{p})$, where for every location $\ell \in \mathcal{L}$, we have $\boldsymbol{\kappa}[\ell] = \sum_{r \in \mathcal{R} \wedge r.from = \ell} tr(r)$,
- the *goal* $g(tr)$ of tr, as the tuple $g(tr) = (\boldsymbol{\kappa}', \mathbf{p})$, where for every location $\ell \in \mathcal{L}$, we have $\boldsymbol{\kappa}'[\ell] = \sum_{r \in \mathcal{R} \wedge r.to = \ell} tr(r)$. □

The origin $o(tr)$ of a transition $tr \in Tr(\mathbf{p})$ is the unique tuple $(\boldsymbol{\kappa}, \mathbf{p})$ where the transition tr is enabled, while its goal $g(tr)$ is the unique tuple $(\boldsymbol{\kappa}', \mathbf{p})$ that is obtained by applying the transition tr to its origin $o(tr)$.

We are now ready to define the transition relation of the counter system.

Definition 4.15 (Transition relation $R(\mathbf{p})$)**.** The *transition relation* $R(\mathbf{p})$ is the relation $R(\mathbf{p}) \subseteq \Sigma(\mathbf{p}) \times Tr(\mathbf{p}) \times \Sigma(\mathbf{p})$, such that $\langle \sigma, tr, \sigma' \rangle \in R(\mathbf{p})$, iff the configuration $\sigma = o(tr)$ is the origin and the configuration $\sigma' = g(tr)$ the goal of the transition tr. We write $\sigma \xrightarrow{tr} \sigma'$, to denote that $\langle \sigma, tr, \sigma' \rangle \in R(\mathbf{p})$. □

To ensure that every process moves when the counter system takes a step using the transition relation $R(\mathbf{p})$, we restrict ourselves to *deadlock-free* counter systems, i.e., counter systems where the transition relation is total (every configuration has a successor). A sufficient condition for deadlock-freedom, which requires that for each location $\ell \in \mathcal{L}$, the guard of at least one rule $r \in \mathcal{R}$, outgoing of ℓ, is satisfied. Formally, we encode this condition as follows:

$$\bigwedge_{\ell \in \mathcal{L}} \left(\mathsf{Env} \rightarrow \bigvee_{r \in \mathcal{R} \wedge r.from = \ell} r.\varphi \right)$$

We now define paths and schedules of a counter system, as sequences of configurations and transitions, respectively.

Definition 4.16 (Path)**.** A *path* in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p}) = (\Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p}))$ is a finite sequence $\{\sigma_i\}_{i=0}^{k}$ of configurations, such that for every two consecutive configurations σ_{i-1} and σ_i, for $0 < i \leq k$, there exists a transition $tr_i \in Tr(\mathbf{p})$ such that $\sigma_{i-1} \xrightarrow{tr_i} \sigma_i$.

The *length* of a path $\{\sigma_i\}_{i=0}^{k}$ is the number of transitions occurring in it, equal to k. □

Definition 4.17 (Execution)**.** A path $\{\sigma_i\}_{i=0}^{k}$ in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p}) = (\Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p}))$ is called an *execution* iff $\sigma_0 \in I(\mathbf{p})$. □

Definition 4.18 (Schedule)**.** A *schedule* is a finite sequence $\tau = \{tr_i\}_{i=1}^{k}$ of transitions $tr_i \in Tr(\mathbf{p})$, for $0 < i \leq k$. We denote by $|\tau| = k$ the *length* of the schedule τ. □

Note that in general, a schedule τ is any finite sequence of transitions from the set $Tr(\mathbf{p})$. We now state the conditions under which a schedule τ defines a path in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$.

Definition 4.19 (Feasible schedule)**.** A schedule $\tau = \{tr_i\}_{i=1}^{k}$ is *feasible* if there exists a path $\{\sigma_i\}_{i=0}^{k}$ in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p}) = (\Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p}))$, such that $\sigma_{i-1} \xrightarrow{tr_i} \sigma_i$, for $0 < i \leq k$.

We call the configuration $\sigma_0 = o(\tau)$ the *origin* of the schedule τ, and the configuration $\sigma_k = g(\tau)$ the *goal* of the feasible schedule τ, and write $\sigma_0 \xrightarrow{\tau} \sigma_k$. □

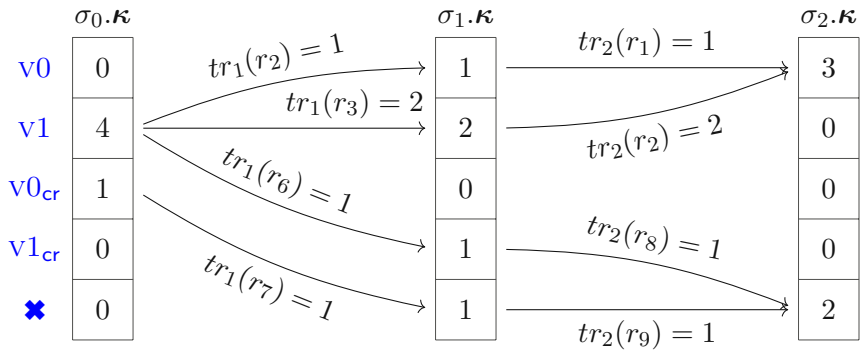
As a feasible schedule τ defines a path in a counter system, all processes take a step in every transition of the feasible schedule τ. This property of feasible schedules is stated by the following proposition, which is a consequence of Definition 4.19.

Proposition 4.1. *The schedule $\tau = \{tr_i\}_{i=1}^{k}$ is feasible iff for every $\ell \in \mathcal{L}$, we have:*

$$\sum_{r.to = \ell} tr_{i-1}(r) = \sum_{r'.from = \ell} tr_i(r') \quad \text{for } r, r' \in \mathcal{R}, \text{ and } 1 < i \leq k$$ □

Intuitively, Proposition 4.1 states that for every feasible schedule $\tau = \{tr_i\}_{i=1}^{k}$, with $1 < i \leq k$, the goal of the transition tr_{i-1} is the origin of the transition tr_i,

Example 4.6. Recall the STA of the algorithm FloodMin, presented in Figure 4.2. Let $\mathbf{p} \in \mathbf{P}_{RC}$ be the admissible valuation of $\boldsymbol{\pi} = \langle n, t, f \rangle$, where $\mathbf{p}[n] = 5, \mathbf{p}[t] = 2$, and $\mathbf{p}[f] = 2$. As FloodMin tolerates crash faults, we have $N(\mathbf{p}) = \mathbf{p}[n]$. A path $\{\sigma_i\}_{i=0}^{2}$ of length 2 in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, is given below, where we depict the $|\mathcal{L}|$-dimensional vectors $\boldsymbol{\kappa}$ of counters of the configurations, and the non-zero factors $tr(r)$, where $r \in \mathcal{R}$.



The configurations $\sigma_0, \sigma_1, \sigma_2$ satisfy the environment assumption Env for crash faults, as:

1. the counters have non-negative values,
2. the sum of counters in each configurations is $N(\mathbf{p})$,
3. the number of processes in the crashed and failed locations $\{\mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}, ✖\}$ is at most $\mathbf{p}[f] = 2$.

The configuration σ_0 is initial, as there are no processes in the locations $\mathcal{L} \setminus \mathcal{I} = \{✖\}$.

For the rules that are not explicitly depicted in the above execution, we assume that their factors in the two transitions tr_1, tr_2 are equal to 0. Therefore, the transitions tr_1, tr_2 are in the set $Tr(\mathbf{p})$, as the sum of factors for all rules is equal to $N(\mathbf{p})$.

In σ_0 the guards on all rules in $\mathcal{R}$ are satisfied, as $\#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} = 1 \geq 1$ and $\#\{\mathrm{v0}\} = 0 < 1$. Thus, a transition which is enabled in σ_0 assigns non-zero factors to any rule outgoing of the locations $\mathrm{v1}, \mathrm{v0}_{\mathsf{cr}}$ (which are the ones populated in σ_0). In particular, the transition tr_1 moves:

- one process from v1 to v0 using the rule r_2, guarded by $r_2.\varphi \equiv \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \geq 1$,
- two processes from v1 to v1 using the rule r_3, guarded by $r_3.\varphi \equiv \#\{\mathrm{v0}\} < 1$,

- one process from V1 to $\mathrm{V1}_{\mathsf{cr}}$ using the rule r_6, guarded by $\top$, and
- one process form $\mathrm{V0}_{\mathsf{cr}}$ to ✖ using the rule r_7, guarded by $\top$.

In σ_1, the guard $r_3.\varphi \equiv \#\{\mathrm{V0}\} < 1$ is not satisfied, as there exists a process in V0. Furthermore, in σ_1, we have that $\#\{\mathrm{V0}_{\mathsf{cr}}, \mathrm{V1}_{\mathsf{cr}}, ✖\} = 2 = \mathbf{p}[f]$, hence the environment stops moving processes to the crash locations $\mathrm{V0}_{\mathsf{cr}}, \mathrm{V1}_{\mathsf{cr}}$. This means that the transition tr_2 moves all processes in V0, V1 from σ_1 to V0 in σ_2, using the rules r_1, r_2, respectively. The transition tr_2 also moves the crashed process from $\mathrm{V1}_{\mathsf{cr}}$ to the failed location ✖ using the rule r_8, and uses r_9 to keep the failed process in ✖.

The schedule $\tau = \{tr_i\}_{i=1}^{2}$ is feasible, as the configuration σ_1 is the goal of tr_1 and the origin of tr_2. □

4.3 Temporal Logic for Specifying Properties

Given a synchronous fault-tolerant distributed algorithm, whose process behavior is modeled using a synchronous threshold automaton STA, we will use linear temporal logic (LTL) formulas over *c-propositions* to encode its properties. We defined the syntax and semantics of *c*-propositions in Section 4.1. The semantics of the Boolean connectives and temporal operators is standard.

In this thesis, we will focus on verifying *safety properties* for the benchmarks we encoded using synchronous threshold automata. The verification methods that we develop will be presented in the next chapter. For each class of benchmarks, we verify the following properties:

- *Validity* and *Agreement* for consensus algorithms (recall Section 1.6.1),
- *Validity* and *k-Agreement* for *k*-set agreement algorithms (recall Section 1.6.2),
- *Unforgeability* for authenticated broadcast algorithms (recall Section 1.6.4).

In the following examples, we show how we can formalize these safety properties for the algorithms whose synchronous threshold automata we introduced in this chapter, namely FloodMin, for $k = 1$; FloodMinOmit, for $k = 1$; and SAB.

Example 4.7. We formalize the properties *Validity* and *k-Agreement* for the algorithm FloodMin, for $k = 1$, whose synchronous threshold automaton is presented in Figure 4.2. In this case, *k-Agreement* coincides with *Agreement*. As described in the pseudocode of the algorithm FloodMin, given in Figure 2.1 on page 40, the algorithm runs for $\lfloor t/k \rfloor + 1$ rounds, after which the processes decide, by assigning the value stored in its variable `best` to the variable `dec` (line 8 of the pseudocode). In the synchronous threshold automaton, we do not have dedicated locations to which the processes move after they have decided, that is, the locations do not encode the value of the variable `dec`.

This is because the automaton only models the loop body of the pseudocode, and the decision in case of FloodMin is done outside of the loop body (cf. Figure 2.1). Therefore, we introduce a flag *decided*, which is true when the loop on line 3 of the pseudocode of FloodMin in Figure 2.1 has terminated, and check whether the processes agree on their value of `best`, which is encoded using the locations v0, v1.

- *Validity.* A value that is not an initial value of any process is not a value that is decided on. We express this using two formulas – one that checks whether all processes have an initial value different than 0, and an analogous formula that checks the same condition for the value 1.

$$\#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} = 0 \rightarrow \mathbf{G}\,(\mathit{decided} \rightarrow \#\{\mathrm{v0}\} = 0)$$
$$\#\{\mathrm{v1}, \mathrm{v1}_{\mathsf{cr}}\} = 0 \rightarrow \mathbf{G}\,(\mathit{decided} \rightarrow \#\{\mathrm{v1}\} = 0)$$

- *Agreement.* No two correct processes decide on different values. That is, we check whether there are no processes in both v0 and v1 once *decided* evaluates to true.

$$\mathbf{G}\,(\mathit{decided} \rightarrow (\#\{\mathrm{v0}\} = 0 \vee \#\{\mathrm{v1}\} = 0))$$ □

Example 4.8. We now formalize the properties for FloodMinOmit, with $k = 1$, which is a variant of FloodMin that tolerates send omission faults. Its synchronous threshold automaton is given in Figure 4.3. As was mentioned in Section 1.2, the send-omission-faulty processes act correctly on the receiving side. That is, for *Validity* and *Agreement*, we check that both *correct* and *send-omission-faulty* processes agree on a value.

- *Validity*, for send omission faults

$$\#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{so}}\} = 0 \rightarrow \mathbf{G}\,(\mathit{decided} \rightarrow \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{so}}\} = 0)$$
$$\#\{\mathrm{v1}, \mathrm{v1}_{\mathsf{so}}\} = 0 \rightarrow \mathbf{G}\,(\mathit{decided} \rightarrow \#\{\mathrm{v1}, \mathrm{v1}_{\mathsf{so}}\} = 0)$$

- *Agreement*, for send omission faults

$$\mathbf{G}\,(\mathit{decided} \rightarrow (\#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{so}}\} = 0 \vee \#\{\mathrm{v1}, \mathrm{v1}_{\mathsf{so}}\} = 0))$$ □

Example 4.9. For the algorithm SAB, whose synchronous threshold automaton is depicted in Figure 4.1, we formalize the property *Unforgeability*:

- *Unforgeability.* If no correct process broadcasts a message initially, then no correct process ever accepts a message.

$$\#\{\mathrm{v1}\} = 0 \rightarrow \mathbf{G}\,(\#\{\mathrm{AC}\} = 0)$$ □

4.4 Discussion

Threshold automata were proposed in [KVW17] for the purpose of modeling *asynchronous* fault-tolerant distributed algorithms. Extensions of threshold automata were studied in [KKW18] and [BKLW19]. The extensions presented in [KKW18] explored the limits on the expressive power of threshold automata for the interleaving semantics. [BKLW19] introduced threshold automata for modeling *round-based randomized* asynchronous fault-tolerant distributed algorithms. Recently, [BEL20] studied the complexity of various verification and synthesis problems for threshold automata. Additionally, [BEL20] defined threshold automata as a tuple of locations $\mathcal{L}$, initial locations $\mathcal{I}$, and rules $\mathcal{R}$, relative to an environment, consisting of the parameters Π and resilience condition RC. This definition, proposed independently of the modeling approach introduced in this thesis, follows a similar idea of splitting the process and environment specification when building formal models of fault-tolerant distributed algorithms. None of these works considered the synchronous case.

In this chapter, we introduced the *synchronous* variant of threshold automata. The synchronous threshold automata allowed us to model multiple synchronous fault-tolerant distributed algorithms, which have both:

- existential guards, that check whether there exists at least one process in a given set of locations,
- threshold guards, that compare the number of processes in a given set of locations against a threshold, i.e., a linear integer arithmetic expression over the parameter values.

Thus, synchronous threshold automata extend the guarded protocols introduced by [EN96], as well as the guarded assignments we introduced in Chapter 2, which support only existential guards and their negations. Generalizing the results from [EN96] to synchronous threshold automata is not straightforward.

Further, with synchronous threshold automata, we are able to capture algorithms that operate in different fault models. We proposed a uniform modeling approach for each of the three fault models that we consider in this thesis: crash, send omission, and Byzantine faults. By combining the different features of the synchronous threshold automata, implied by the fault model, we are able to model algorithms that tolerate hybrid faults as well. In our benchmarks, we encoded the combination of send omission and Byzantine faults for several algorithms from [BSW11], namely HybridKing, HybridQueen, and HybridSAB. Different variants of these algorithms (e.g., crash and Byzantine faults) can certainly be modeled using our approach, but we have not considered them in this thesis.

CHAPTER 5

Parameterized Verification of Safety using Bounded Model Checking

In this chapter, given a synchronous threshold automaton STA, that is used to model the behavior of a process running a synchronous fault-tolerant distributed algorithm, and a parameterized counter system $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$, induced by the STA, we will reduce the parameterized model checking problem for safety properties to the bounded model checking problem.

The main ideas of this reduction are the following. We show that for our benchmarks, we can solve the parameterized model checking of safety properties by solving its *dual* parameterized reachability problem, which we introduce in this chapter. We show that the parameterized reachability problem for synchronous threshold automata is undecidable in general. However, for STA where all bad configurations in all counter systems, represented by a parameterized counter system $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$, can be reached using executions of bounded length, we can use bounded model checking as a complete verification procedure. This allows us to either verify the safety properties of our benchmarks, or obtain non-spurious counterexamples. Therefore, we are interested in showing that the parameterized counter system $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$, for a given STA, has a bound on the diameter, which can be used as a completeness threshold for bounded model checking. Due to the undecidability, this bound does not always exist. The remainder of this chapter is organized along the issues highlighted below.

Safety as Reachability. To verify the safety properties of our benchmarks, which we formalized in Section 4.3, for all admissible values of the parameters, we can consider the following dual problem. Namely, given an STA, we can check if there exists some

$\mathbf{p} \in \mathbf{P}_{RC}$, and an execution of finite length in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, that reaches a *bad* configuration, i.e., a configuration in which the safety property is violated. In Section 5.1, we formalize the *parameterized reachability problem* for STA, and show the duality of the parameterized model checking problem for safety properties and the parameterized reachability problem.

Undecidability of Parameterized Reachability. In general, the question whether a certain configuration can be reached in a counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, for some $\mathbf{p} \in \mathbf{P}_{RC}$, involves reasoning about infinitely many finite-state counter systems represented by the parameterized counter system $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$. In Section 5.2, we give a proof of *undecidability* of parameterized reachability for STA, by reduction from the halting problem of two-counter machines [Min67].

Bounded Diameter. We are interested in computing a *bound* $\mathsf{d} \in \mathbb{N}$, which, if it exists, does not depend on the values of the parameters from the set Π. Moreover, for the bound d, we want to show that every execution can be shortened to an execution of length at most d (that starts and ends in the same configurations), in any counter system induced by the STA and an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$ of the parameters. If this bound d exists, we say that the STA has *bounded diameter*. In Section 5.3, we formally introduce the notion of diameter for STA adapted from the definition of diameter from [BCCZ99]. A bound on the diameter allows us to verify safety properties by searching for their violations in executions of finite, bounded length.

Semi-Decision Procedure. The undecidability of parameterized reachability implies that, in general, a bound d on the diameter of a given STA may not exist, i.e., that there are STA with unbounded diameter. Nevertheless, in Section 5.3.1, we propose a *semi-decision procedure* for computing the bound $\mathsf{d} \in \mathbb{N}$ on the diameter using SMT. This procedure enumerates candidates for the diameter bound, checks if the candidate is indeed the diameter, and it terminates if it finds such a bound. For example, for the STA in of the algorithm SAB, depicted in Figure 4.1, this procedure computes the diameter 2.

Theoretical Bound. For some benchmarks, whose STA satisfy certain conditions, we establish the existence of a bounded diameter *theoretically*. In Section 5.4, we introduce a class of STA, that captures several algorithms (such as the authenticated broadcast algorithm SAB whose STA is presented in Figure 4.1), and prove that a bound on the diameter can be computed and is independent of the parameters.

Bounded Model Checking. The existence of a bound on the diameter motivates the use of bounded model checking [BCCZ99, KS03, CKOS04] for verifying safety properties. Crucially, this approach is complete because if an execution reaches a bad configuration that violates the property, this bad configuration can also be reached by an execution of finite length, bounded by the diameter. In Section 5.5, we give an SMT encoding for checking the violation of a safety property by executions with length up to the diameter.

Experimental Evaluation. We implemented our SMT-based semi-decision procedure for computing the diameter, with Z3 [dMB08] and CVC4 [CTTV04] as back-end solvers, and applied it to the benchmarks listed in Table 1.1. In Section 5.6, we present the bounds on the diameter computed by our semi-decision procedure. We observe that for the STA defined in this thesis (with linear guards and linear constraints on the parameters), the SMT encoding of the executions of bounded length, that represent violations of the safety properties, results in a formula in Presburger arithmetic. Hence, checking safety properties (that can be expressed in Presburger arithmetic) is decidable for STA with bounded diameter. In Section 5.6, we also experimentally demonstrate that current SMT solvers can handle these formulas well.

To our knowledge, we are the first to automatically verify the benchmarks that tolerate Byzantine, send omission, and hybrid faults. For the benchmarks that tolerate crash faults, and which overlap with the benchmarks verified in Chapter 3, the bounded model checking technique we present in this chapter performs significantly better than the abstraction-based technique presented in Chapter 3.

5.1 Parameterized Model Checking of Safety to Parameterized Reachability

We observe that the safety properties we expressed for our benchmarks in Section 4.3 are LTL formulas of the shape:

$$\phi \equiv \mathsf{init} \rightarrow \mathbf{G}\,(\mathsf{global}) \tag{5.1}$$

where both init and global are Boolean combinations of c-propositions, i.e., they do not contain other temporal operators.

Recall that a feasible schedule in a counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, for a given synchronous threshold automaton STA and an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$, is a sequence $\tau = \{tr_i\}_{i=1}^{k}$ of transitions that induces a path in the counter system (Definition 4.19 on page 117). Further, we denote by $o(\tau)$ and $g(\tau)$ the origin and goal of the feasible schedule τ, that is, the first and final configuration in the path induced by τ. We now define the parameterized reachability problem for synchronous threshold automata.

Parameterized Reachability PRP

INPUT:
- a synchronous threshold automaton STA
- a Boolean combination ϕ_{init} of c-propositions
- a Boolean combination ϕ_{reach} of c-propositions

QUESTION: Do there exist:

- an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$,
- an initial configuration σ_0 in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$,
- a feasible schedule τ in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$,

with $\sigma_0 \models \phi_{\mathsf{init}}$ and $\sigma_0 = o(\tau)$, such that $g(\tau) \models \phi_{\mathsf{reach}}$?

Fix a synchronous threshold automaton STA, and an instance $\mathrm{PMCP}(\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi}), \phi)$ of the parameterized model checking problem (stated in Section 1.3.2 on page 11), for the parameterized counter system $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$ and a safety property ϕ, whose shape is as in equation (5.1). The question of the parameterized model checking problem is answered positively if we can verify that for every valuation $\mathbf{p} \in \mathbf{P}_{RC}$, the safety property $\phi \equiv \mathsf{init} \rightarrow \mathbf{G}\,(\mathsf{global})$ holds in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$. More precisely:

$$\begin{aligned} \forall \mathbf{p} \in \mathbf{P}_{RC} \quad & \mathsf{CS}(\mathsf{STA}, \mathbf{p}) \models \phi && \text{iff} \\ \forall \mathbf{p} \in \mathbf{P}_{RC}\ \forall \sigma_0 \in I(\mathbf{p}) \quad & \sigma_0 \models \phi && \text{iff} \\ \forall \mathbf{p} \in \mathbf{P}_{RC}\ \forall \sigma_0 \in I(\mathbf{p}) \quad & \sigma_0 \models \mathsf{init} \text{ implies } \sigma_0 \models \mathbf{G}\,(\mathsf{global}) && \text{iff} \\ \neg(\exists \mathbf{p} \in \mathbf{P}_{RC}\ \exists \sigma_0 \in I(\mathbf{p}) \quad & \sigma_0 \models \mathsf{init} \text{ and } \sigma_0 \models \mathbf{F}\,(\neg\mathsf{global})) && (5.2) \end{aligned}$$

We create an instance $\mathrm{PRP}(\mathsf{STA}, \phi_{\mathsf{init}}, \phi_{\mathsf{reach}})$ of the parameterized reachability problem, where $\phi_{\mathsf{init}} \equiv \mathsf{init}$ and $\phi_{\mathsf{reach}} \equiv \neg\mathsf{global}$. The following theorem is a consequence of (5.2), and shows the duality of $\mathrm{PMCP}(\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi}), \phi)$, where $\phi \equiv \mathsf{init} \rightarrow \mathbf{G}\,(\mathsf{global})$, and $\mathrm{PRP}(\mathsf{STA}, \mathsf{init}, \neg\mathsf{global})$, for a given STA.

Theorem 5.1 (PMCP to PRP)**.** *Given a synchronous threshold automaton STA and a safety property $\phi \equiv \mathsf{init} \rightarrow \boldsymbol{G}\,(\mathsf{global})$, the answer to $PMCP(\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi}), \phi)$ is positive iff the answer to $PRP(\mathsf{STA}, \mathsf{init}, \neg\mathsf{global})$ is negative.* □

Thus, we can verify safety properties $\phi \equiv \mathsf{init} \rightarrow \mathbf{G}\,(\mathsf{global})$ of an algorithm modeled using a synchronous threshold automaton STA, for all admissible parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, by checking whether there exists some finite execution of some counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p}')$, for $\mathbf{p}' \in \mathbf{P}_{RC}$, whose initial configuration satisfies init, and whose last configuration satisfies $\neg\mathsf{global}$.

Example 5.1. Recall the safety properties *Validity*, *Agreement*, and *Unforgeability*, for the benchmarks $\mathsf{FloodMin}$, $\mathsf{FloodMinOmit}$, and SAB, which we formalized in Examples 4.7 on page 119, 4.8 on page 120, and 4.9 on page 120, respectively.

The parameterized reachability checks are as follows:

- for *Validity*, we check for every value $v_i \in V$, whether a configuration where $\#\{\mathrm{v}i\} > 0$ holds is reachable from an initial configuration that satisfies $\#\{\mathrm{v}i\} = 0$;
- for *Agreement* where $k = 1$, we have that $\mathsf{init} \equiv \top$, thus we check:
  - in the case of crash faults, that a configuration where $\#\{\mathrm{v0}\} > 0 \wedge \#\{\mathrm{v1}\} > 0$ holds is reachable from any initial state,
  - in the case of send omission faults, that a configuration where $\#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{so}}\} > 0 \wedge \#\{\mathrm{v1}, \mathrm{v1}_{\mathsf{so}}\} > 0$ holds is reachable from any initial state;
- for *Unforgeability*, we check whether a configuration where $\#\{\mathrm{AC}\} > 0$ holds is reachable from an initial configuration that satisfies $\#\{\mathrm{v1}\} = 0$. □

5.2 Undecidability of Parameterized Reachability

We show that the parameterized reachability problem for STA is undecidable in general, by reduction from the halting problem of a two-counter machine [Min67], which is known to be undecidable. Such reductions are common in parameterized verification, e.g., see [BJK+15].

5.2.1 Two-Counter Machine

A *two-counter machine* $\mathcal{M}$ consists of: two *registers* A and B, and a set $\mathcal{I}$ of *instructions*. The set $\mathcal{I}$ contains the following instructions:

- increment inc_i, for $i \in \{A, B\}$
- decrement dec_i, for $i \in \{A, B\}$,
- jump-if-zero $jz_i(k)$, for $i \in \{A, B\}$ and $k \in \mathbb{N}$,
- halting instruction *halt*.

The increment, decrement, and jump-if-zero instructions are defined for each of the registers.

A finite sequence $P = \langle inst_1, \ldots, inst_{|P|} \rangle$ of instructions, where $inst_j \in \mathcal{I}$, for $1 \leq j \leq |P|$ and $|P| \in \mathbb{N}$, is called the *program* of the machine $\mathcal{M}$. A configuration of the machine $\mathcal{M}$ is given by the tuple $\langle \mathsf{pc}, \mathsf{a}, \mathsf{b} \rangle$, where pc is the *program counter*, that ranges from 1 to $|P|$, and $\mathsf{a}, \mathsf{b} \in \mathbb{N}$ are the current values of the registers A, B, respectively.

The machine $\mathcal{M}$ starts the execution of the program P by executing the initial instruction $inst_1$. We also assume that initially the registers A and B contain the value 0. That is, the initial configuration of the machine is $\langle 1, 0, 0 \rangle$. Depending on the value of the program counter, the machine executes the program as follows. If the instruction $inst_{\mathsf{pc}}$, for $1 \leq \mathsf{pc} \leq |P|$, is:

- an increment instruction inc_i, for $i \in \{A, B\}$, then the machine $\mathcal{M}$ *increments* register i, and moves the control of the program to location $\mathsf{pc} + 1$. That is, the machine moves from the configuration $\langle \mathsf{pc}, \mathsf{a}, \mathsf{b} \rangle$ to the configuration $\langle \mathsf{pc}+1, \mathsf{a}+1, \mathsf{b} \rangle$ if $i = A$, and to the configuration $\langle \mathsf{pc} + 1, \mathsf{a}, \mathsf{b} + 1 \rangle$ if $i = B$;
- a decrement instruction dec_i, for $i \in \{A, B\}$, then the machine $\mathcal{M}$ *decrements* register i, and moves the control of the program to location $\mathsf{pc} + 1$, That is, the machine moves from the configuration $\langle \mathsf{pc}, \mathsf{a}, \mathsf{b} \rangle$ to the configuration $\langle \mathsf{pc}+1, \max\{\mathsf{a}-1, 0\}, \mathsf{b} \rangle$ if $i = A$, and to the configuration $\langle \mathsf{pc} + 1, \mathsf{a}, \max\{\mathsf{b} - 1, 0\} \rangle$ if $i = B$;
- a jump-if-zero instruction $jz_i(k)$, for $i \in \{A, B\}$ and $1 \leq k \leq |P|$, then the machine $\mathcal{M}$ moves the control of the program to the location k, in case the

register i contains the value 0, and to the location $\mathsf{pc}+1$ otherwise. That is, the machine moves from the configuration $\langle \mathsf{pc}, \mathsf{a}, \mathsf{b}\rangle$ to the configuration $\langle k, \mathsf{a}, \mathsf{b}\rangle$ if $i = A$ and $\mathsf{a} = 0$, and to the configuration $\langle \mathsf{pc}+1, \mathsf{a}, \mathsf{b}\rangle$ if $i = A$ and $\mathsf{a} \neq 0$. The steps that the machine makes in case $i = B$ are analogous;

- a halting instruction *halt*, then $\mathsf{pc} = |P|$, i.e., $inst_{|P|} = halt$.

That is, the increment, decrement, and jump-if-zero instructions cause the machine to perform an action and move the control of the program to another program location, while the halting instruction is the last instruction of the program, which stops the machine.

We recall the halting problem for two-counter machines [Min67].

Halting Problem for Two-Counter Machines HALTING$_{2CM}$

INPUT:	• two-counter machine $\mathcal{M}$ • program $P = \langle inst_1, \ldots, inst_{|P|}\rangle$
QUESTION:	Does there exist an execution of the program P by the machine $\mathcal{M}$, that starts in the configuration $\langle 1, 0, 0\rangle$ and ends in a configuration $\langle |P|, \mathsf{a}, \mathsf{b}\rangle$, for some $\mathsf{a}, \mathsf{b} \in \mathbb{N}$?

5.2.2 Undecidability of Parameterized Reachability

To prove undecidability of the parameterized reachability problem, we construct an automaton $\mathsf{STA}_{\mathcal{M},P}$, such that every counter system $\mathsf{CS}(\mathsf{STA}_{\mathcal{M},P}, \mathbf{p})$ induced by it simulates the steps that a two-counter machine $\mathcal{M}$ takes when executing an arbitrary program P.

The constructed $\mathsf{STA}_{\mathcal{M},P} = (\mathcal{L}_{\mathcal{M},P}, \mathcal{I}_{\mathcal{M},P}, \mathcal{R}_{\mathcal{M},P}, \Pi_{\mathcal{M},P}, RC_{\mathcal{M},P}, \mathsf{Env}_{\mathcal{M},P})$ has a single parameter – the number n of processes, that is, $\Pi_{\mathcal{M},P} = \{n\}$ and $\boldsymbol{\pi}_{\mathcal{M},P} = \langle n\rangle$. Furthermore, as there is only one parameter, the resilience condition is $RC_{\mathcal{M},P} \equiv \top$. This implies that the set $\mathbf{P}_{RC}$ of admissible valuations contains valuations of $\boldsymbol{\pi}_{\mathcal{M},P}$, where n is assigned a natural number, i.e., where $\boldsymbol{\pi}_{\mathcal{M},P}[n] \in \mathbb{N}$.

The main idea in the construction of $\mathsf{STA}_{\mathcal{M},P}$ is that each of the n processes plays one of two roles:

1. *controller* role, that is, a controller process is used to encode the control flow of the program P,
2. *storage* role, that is, a storage process is used to encode the values of the registers in unary, as in [EN03].

Thus, $\mathsf{STA}_{\mathcal{M},P}$ consists of two parts – one per role, as depicted in Figure 5.1. We now proceed by defining the structure of the automaton $\mathsf{STA}_{\mathcal{M},P}$.

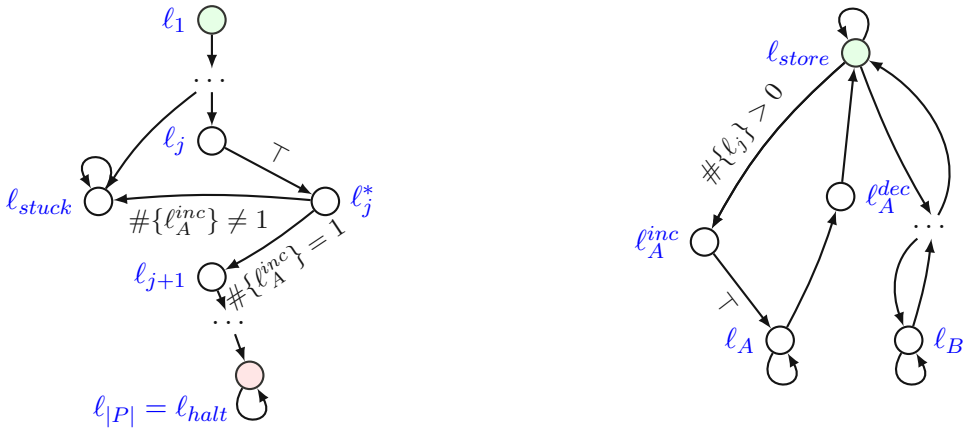


Figure 5.1: The synchronous threshold automaton $\mathsf{STA}_{\mathcal{M},P}$, with the controller part on the left, and the storage part on the right. The rules whose guards are depicted in the figure are used to encode increment of register A.

Locations $\mathcal{L}_{\mathcal{M},P}$. The set $\mathcal{L}_{\mathcal{M},P} = \mathcal{L}_C \cup \mathcal{L}_S$ of locations is partitioned into locations $\mathcal{L}_C$ and $\mathcal{L}_S$ of the controller and of the storage role, respectively.

The set $\mathcal{L}_C$ of *controller locations* contains a location ℓ_j for each instruction $inst_j$, where $1 \leq j \leq |P|$, an additional location ℓ_j^* if $inst_j$ is an increment or decrement instruction, and a special location ℓ_{stuck} that denotes a stuck configuration of the two-counter machine.

The set $\mathcal{L}_S$ of *storage locations* contains the location ℓ_{store}, one location for each of the registers ℓ_A, ℓ_B, and one location per increment/decrement instruction $\ell_i^{inc}, \ell_i^{dec}$, for $i \in \{A, B\}$. Intuitively, the state ℓ_{store} is used to store processes that will eventually make transitions to one of ℓ_i, for $i \in \{A, B\}$, via ℓ_i^{inc}, and those that make transitions from ℓ_i via ℓ_i^{dec}.

The set $\mathcal{I}_{\mathcal{M},P} \subseteq \mathcal{L}_{\mathcal{M},P}$ of *initial locations* contains the locations $\ell_1 \in \mathcal{L}_C$ and $\ell_{store} \in \mathcal{L}_S$.

Rules $\mathcal{R}_{\mathcal{M},P}$. The set $\mathcal{R}_{\mathcal{M},P} = \mathcal{R}_C \cup \mathcal{R}_S$ of rules consists of rules $\mathcal{R}_C$ and $\mathcal{R}_S$ for the controller and storage processes, respectively. An increment (resp. decrement) of register A is modeled by two steps that move a storage process to (resp. from) the location ℓ_A from (resp. to) the location ℓ_{store}. This ensures that exactly one storage process moves, which will be explained in more detail below. The controller processes move to the location corresponding to the next instruction, that is, from ℓ_j to ℓ_{j+1} via the additional location ℓ_j^*, if $inst_j$ is an increment (resp. decrement) instruction, for $1 \leq j < |P|$. The jump-if-zero instruction of register A is modeled by moving the controller processes to the location ℓ_k if there are no processes in the location ℓ_A, and to the location corresponding to the next instruction otherwise. The increment, decrement, and jump-if-zero for register B are modeled in an analogous way.

We now formally define the rules in $\mathcal{R}_{\mathcal{M},P} = \mathcal{R}_C \cup \mathcal{R}_S$. Let $inst_j$, for $1 \leq j \leq |P|$, be

an instruction of the program P, and $i \in \{A, B\}$. For convenience, we use the notation "$\ell \to \ell'$ if φ" for the rule $(\ell, \ell', \varphi) \in \mathcal{R}_{\mathcal{M},P}$.

The set $\mathcal{R}_C$ of *controller rules* contains:

$$\begin{aligned} \ell_{stuck} &\to \ell_{stuck} && \text{if } \top \\ \ell_{|P|} &\to \ell_{|P|} && \text{if } \top \end{aligned}$$

Depending on $inst_j$, we consider the following cases.

Case 1. If $inst_j$ is an inc_i instruction, with $1 \leq j < |P|$, then $\mathcal{R}_C$ contains the rules (depicted in Figure 5.1):

$$\begin{aligned} \ell_j &\to \ell_j^* && \text{if } \top \\ \ell_j^* &\to \ell_{j+1} && \text{if } \#\{\ell_i^{inc}\} = 1 \\ \ell_j^* &\to \ell_{stuck} && \text{if } \#\{\ell_i^{inc}\} \neq 1 \end{aligned} \tag{5.3}$$

Case 2. If $inst_j$ is a dec_i instruction, with $1 \leq j < |P|$, then $\mathcal{R}_C$ contains the rules:

$$\begin{aligned} \ell_j &\to \ell_j^* && \text{if } \top \\ \ell_j^* &\to \ell_{j+1} && \text{if } \#\{\ell_i^{dec}\} = 1 \\ \ell_j^* &\to \ell_{stuck} && \text{if } \#\{\ell_i^{dec}\} \neq 1 \end{aligned} \tag{5.4}$$

Case 3. If $inst_j$ is a $jz_i(k)$ instruction, with $1 \leq j < |P|$ and $1 \leq k \leq |P|$, then $\mathcal{R}_C$ contains the rules:

$$\begin{aligned} \ell_j &\to \ell_k && \text{if } \#\{\ell_i\} = 0 \\ \ell_j &\to \ell_{j+1} && \text{if } \#\{\ell_i\} \neq 0 \end{aligned} \tag{5.5}$$

The set $\mathcal{R}_S$ of *storage rules* contains, for $i \in \{A, B\}$:

$$\begin{aligned} \ell_{store} &\to \ell_{store} && \text{if } \top \\ \ell_i &\to \ell_i && \text{if } \top \end{aligned}$$

Again, based on $inst_j$, we consider the cases:

Case 1. If $inst_j$ is an inc_i instruction, then $\mathcal{R}_S$ contains the rules:

$$\begin{aligned} \ell_{store} &\to \ell_i^{inc} && \text{if } \#\{\ell_j\} > 0 \\ \ell_i^{inc} &\to \ell_i && \text{if } \top \end{aligned}$$

Case 2. If $inst_j$ is a dec_i instruction, then $\mathcal{R}_S$ contains the rules:

$$\begin{aligned} \ell_i &\to \ell_i^{dec} && \text{if } \#\{\ell_j\} > 0 \\ \ell_i^{dec} &\to \ell_{store} && \text{if } \top \end{aligned}$$

Note that, in case $inst_j$ is a $jz_i(k)$ instruction, we do not need to introduce new rules in $\mathcal{R}_S$, as the two-counter machine does not modify the value of the registers when performing a $jz_i(k)$ instruction.

Environment Assumption $\mathsf{Env}_{\mathcal{M},P}$. As there are no faults, the environment assumptions $\mathsf{Env}_{\mathcal{M},P}$ contains the conjuncts $\#\{\ell\} \geq 0$, for $\ell \in \mathcal{L}_{\mathcal{M},P}$ and $\#\mathcal{L}_{\mathcal{M},P} = n$. Moreover, the environment requires that there is at least one process in the controller locations, that is, $\#\mathcal{L}_C \geq 1$.

Reduction. The above construction allows multiple processes to act as controllers, and since we assume that the two-counter machine is deterministic, all the controllers behave the same.

To truly model an increment (resp. decrement) of register i, for $i \in \{A, B\}$, the controller processes have to ensure that exactly one process was moved to (resp. from) the location ℓ_i via the location ℓ_i^{inc} (resp. ℓ_i^{dec}). In principle, several storage processes can move to the location ℓ_i^{inc} (resp. ℓ_i^{dec}) at once. To ensure that exactly one process was moved, for every inc_i (resp. dec_i) instruction in the program P, for $i \in \{A, B\}$, the controller processes check whether the guard of the rule (5.3) (resp. (5.4)) is satisfied. If this is the case, that is, if there is a single storage process in the location ℓ_i^{inc} (resp. ℓ_i^{dec})), the controllers move to the location corresponding to the next instruction, and the number of storage processes in the location ℓ_i is increased (resp. decreased) by exactly one, modeled by moving the single storage process from the location ℓ_i^{inc} (resp. ℓ_i^{dec})) to the location ℓ_i (resp. ℓ_{store}). Otherwise, that is, if there are multiple storage process in the location ℓ_i^{inc} (resp. ℓ_i^{dec}), all controller processes are moved to the stuck location, and the number of processes in ℓ_i no longer corresponds to the value of register i.

Consider Figure 5.1, which depicts the locations and rules that encode the increment of register A. The controllers that are in location ℓ_j^* move to the location ℓ_{j+1} if the guard $\#\{\ell_A^{inc}\} = 1$ is satisfied. If $\#\{\ell_A^{inc}\} \neq 1$, the controllers move from ℓ_j^* to the stuck location ℓ_{stuck}.

Similarly, for every $jz_i(k)$ instruction, all the controllers move to the location ℓ_k if the guard of rule (5.5) is satisfied, that is, if the number of storage processes in the location ℓ_i is 0, which corresponds to the value of register i being equal to 0. Otherwise, the controllers move to the location corresponding to the next instruction in the program P.

The main invariant which ensures correctness of the construction is that every transition in a counter system induced by $\mathsf{STA}_{\mathcal{M},P}$ either faithfully simulates a step of the two-counter machine, or moves all of the controller processes to the stuck location. Furthermore, if there are no controller processes in the stuck location, the number of processes in locations ℓ_A, ℓ_B denote the current values of the registers A, B, respectively.

To formally state the reduction, given the constructed $\mathsf{STA}_{\mathcal{M},P}$, we create an instance $\mathrm{PRP}(\mathsf{STA}_{\mathcal{M},P}, \phi_{\mathsf{init}}, \phi_{\mathsf{reach}})$ of the parameterized reachability problem, where $\phi_{\mathsf{init}} \equiv \top$, and $\phi_{\mathsf{reach}} \equiv \#\{\ell_{|P|}\} > 0$. The latter formula states that the controller processes reach the location $\ell_{|P|} \in \mathcal{L}_C$, which encodes the halting instruction $inst_{|P|} = halt$ of the program P.

Theorem 5.2 (Reduction)**.** *The answer to* $PRP(\mathsf{STA}_{\mathcal{M},P}, \phi_{\mathsf{init}}, \phi_{\mathsf{reach}})$ *is positive iff the answer to* $\textsc{Halting}_{2CM}(\mathcal{M}, P)$ *is positive.* □

In other words, there exists an execution in some counter system $\mathsf{CS}(\mathsf{STA}_{\mathcal{M},P}, \mathsf{n})$, where $\mathsf{n} \in \mathbb{N}$, in which the controller processes reach the halting instruction $\ell_{|P|}$ iff the two-counter machine $\mathcal{M}$ halts while executing the program P.

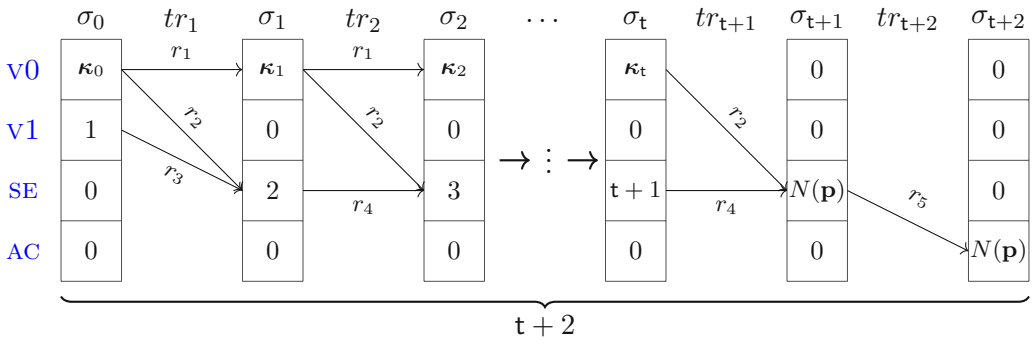
We now sketch the two arguments that give us undecidability of parameterized reachability. Suppose that the machine $\mathcal{M}$ halts while executing the program P. By the above construction we get that for some $\mathsf{n} \in \mathbb{N}$, there exists an initial configuration σ, and a configuration σ', reachable from the configuration σ in the counter system $\mathsf{CS}(\mathsf{STA}_{\mathcal{M},P}, \mathsf{n})$, such that ϕ_{reach} holds in σ', i.e., $\sigma' \models \#\{\ell_{|P|}\} > 0$. In the other direction, suppose that the machine $\mathcal{M}$ does not halt while executing the program P. Then, for every $\mathsf{n} \in \mathbb{N}$, and every initial configuration σ in the counter system $\mathsf{CS}(\mathsf{STA}_{\mathcal{M},P}, \mathsf{n})$, it holds that for every configuration σ', reachable from the initial configuration σ, we have $\sigma' \models \#\{\ell_{|P|}\} = 0$, that is, $\sigma' \not\models \phi_{\mathsf{reach}}$.

5.3 Diameter

Given a synchronous threshold automaton STA, we call the diameter of an STA the maximal number of transitions needed to reach all possible configurations in every counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ induced by the STA and an admissible instance $\mathbf{p} \in \mathbf{P}_{RC}$. We adapt the definition of diameter from [BCCZ99].

Definition 5.1 (Diameter)**.** Given an STA, the *diameter* of the STA is the smallest number $\mathsf{d} \in \mathbb{N}$ such that for every $\mathbf{p} \in \mathbf{P}_{RC}$ and every path $\{\sigma_i\}_{i=0}^{\mathsf{d}+1}$ of length $\mathsf{d}+1$ in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, there exists a path $\{\sigma'_j\}_{j=0}^{\mathsf{e}}$ of length $\mathsf{e} \leq \mathsf{d}$ in $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, where $\sigma_0 = \sigma'_0$ and $\sigma_{\mathsf{d}+1} = \sigma'_{\mathsf{e}}$. □

Example 5.2. Consider an execution of a counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, where STA is the synchronous threshold automaton for the algorithm SAB (given in Figure 4.1 on page 104), and where $\mathbf{p} \in \mathbf{P}_{RC}$ is some admissible valuation of the parameter vector $\boldsymbol{\pi}$. The counter system has $N(\mathbf{p}) = \mathbf{p}[n] - \mathbf{p}[f]$, and we assume that $\mathbf{p}[t] = \mathsf{t}$. The following is an execution of length $\mathsf{t}+2$, where we have $\boldsymbol{\kappa}_i = N(\mathbf{p}) - (i+1)$, for $0 \leq i \leq \mathsf{t}$:



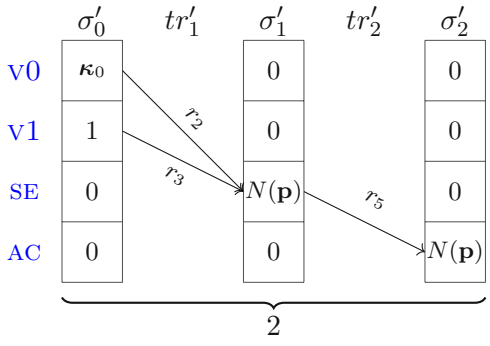
$\boldsymbol{\kappa}_0 = N(\mathbf{p}) - 1 \quad \boldsymbol{\kappa}_1 = N(\mathbf{p}) - 2 \quad \boldsymbol{\kappa}_2 = N(\mathbf{p}) - 3 \quad \boldsymbol{\kappa}_{\mathsf{t}} = N(\mathbf{p}) - (\mathsf{t}+1)$

The execution starts in the initial configuration σ_0, where one process is in location V1, and the remaining $N(\mathbf{p}) - 1$ processes are in location V0. Observe that the guards of the rules r_1 and r_2 are both satisfied in the configuration σ_0. Thus, the processes in location V0 can either stay in V0 by applying the rule r_1, or move to the location SE by applying the rule r_2. The transition tr_1, moves one process from V0 to SE, while keeping the other $N(\mathbf{p}) - 2$ processes in the location V0. Additionally, the single process in V1 moves to SE.

In the configuration σ_1, obtained by applying the transition tr_1 to the initial configuration σ_0, it is again the case that the guards of the rules r_1 and r_2 are both satisfied. Thus, transition tr_2, and all subsequent transitions until the configuration $\sigma_{\mathtt{t}}$ is reached, move a single process from V0 to SE by applying the rule r_2, and keep the other processes in V0 and V1, by applying the self-loop rules r_1 and r_4, respectively.

In the configuration $\sigma_{\mathtt{t}}$, there are $\mathtt{t}+1$ processes in the location SE, and $N(\mathbf{p}) - (\mathtt{t}+1)$ processes in location V0. Here, the guard $r_1.\varphi \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \leq t+1$ does not hold anymore, hence in the transition $tr_{\mathtt{t}+1}$, all processes move to the location SE, and finally, using the transition $tr_{\mathtt{t}+2}$, they move to the location AC.

The length of this execution depends on the value $\mathtt{t} = \mathbf{p}[t]$ of the parameter t, which implies that, for increasing values of the parameter t, the length of this execution can become unbounded. A question that arises is: Can every configuration be reached in every counter system by an execution of finite length, that does not depend on the values for the parameters? Below, we give an execution of length 2 that starts in the same initial configuration, and reaches the same final configuration as the long execution above:



where $\sigma'_0 = \sigma_0$ and $\sigma'_2 = \sigma_{\mathtt{t}+2}$.

In this short execution, all processes that are initially in location V0 are moved by the transition tr'_1 to the location SE. In the configuration σ'_1, all the $N(\mathbf{p})$ processes are in location SE, and they use the rule r_5 to move to the location AC in the transition tr'_2. □

5.3.1 Semi-Decision Procedure for Computing the Diameter

By Definition 5.1, given an STA, the diameter of the parameterized counter system $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$ is the smallest number d that satisfies the formula:

$$\forall \mathbf{p} \in \mathbf{P}_{RC}. \forall \sigma_0, \ldots, \sigma_{\mathsf{d}+1}. \forall tr_1, \ldots, tr_{\mathsf{d}+1}. \exists \sigma'_0, \ldots, \sigma'_{\mathsf{d}}. \exists tr'_1, \ldots, tr'_{\mathsf{d}}.$$
$$Path(\sigma_0, \sigma_{\mathsf{d}+1}, \mathsf{d}+1) \;\rightarrow\; (\sigma_0 = \sigma'_0) \wedge Path(\sigma'_0, \sigma'_{\mathsf{d}}, \mathsf{d}) \wedge \bigvee\nolimits_{i=0}^{\mathsf{d}} \sigma'_i = \sigma_{\mathsf{d}+1} \quad (5.6)$$

where

$$Path(\sigma_0, \sigma_{\mathsf{d}}, \mathsf{d}) \equiv \bigwedge_{i=0}^{\mathsf{d}-1} R(\sigma_i, tr_{i+1}, \sigma_{i+1}), \quad (5.7)$$

and $R(\sigma, tr, \sigma')$ is a predicate which evaluates to true if $\sigma \xrightarrow{tr} \sigma'$. That is, for a fixed $\mathbf{p} \in \mathbf{P}_{RC}$, the predicate $R(\sigma, tr, \sigma')$ models one transition, while $Path(\sigma_0, \sigma_{\mathsf{d}}, \mathsf{d})$ models a path of length d in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$.

Since we assume deadlock-free counter systems (see Section 4.2 on page 114), a path of length d, modeled by the formula $Path(\sigma'_0, \sigma'_{\mathsf{d}}, \mathsf{d})$ in (5.6), exists in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ even if the disjunction $\bigvee_{i=0}^{\mathsf{d}} \sigma'_i = \sigma_{\mathsf{d}+1}$ is satisfiable because $\sigma'_i = \sigma_{\mathsf{d}+1}$ holds for some $i < \mathsf{d}$.

Given an STA, we can determine its diameter d using the following procedure:

1. initialize the candidate diameter d to 1;
2. check whether the negation of the formula (5.6) is unsatisfiable;
3. if yes, then output d and terminate;
4. if not, then increment d and jump to step 2.

Due to the undecidability of parameterized reachability (Section 5.2.2), the procedure may not terminate. If it does, it outputs the diameter. We implemented this procedure, and used a back-end SMT solver to automate the test in step 2. We report on the results obtained by applying this procedure to our benchmarks in Section 5.6. Before we do that, in the remainder of this section, we discuss the SMT encoding we used in our implementation.

5.3.2 SMT Encoding

Consider the negation of the formula (5.6):

$$\exists \mathbf{p} \in \mathbf{P}_{RC}. \exists \sigma_0, \ldots, \sigma_{\mathsf{d}+1}. \exists tr_1, \ldots, tr_{\mathsf{d}+1}. Path(\sigma_0, \sigma_{\mathsf{d}+1}, \mathsf{d}+1)$$
$$\wedge \;\; \forall \sigma'_0, \ldots, \sigma'_{\mathsf{d}}. \forall tr'_1, \ldots, tr'_{\mathsf{d}}. ((\sigma_0 = \sigma'_0) \wedge Path(\sigma'_0, \sigma'_{\mathsf{d}}, \mathsf{d}) \rightarrow \bigwedge\nolimits_{i=0}^{\mathsf{d}} \sigma'_i \neq \sigma_{\mathsf{d}+1}) \quad (5.8)$$

To be able to check the unsatisfiability of the formula (5.8), which is done in step 2 of the above procedure, our implementation generates an SMT-LIB [BFT16] file that encodes the formula (5.8). The produced SMT encoding of the formula (5.8) is in Skolem normal form, that is, we create constants for the existentially quantified variables. We give details on the SMT encoding below.

Admissible Parameter Values. We declare integer constants π, for each parameter $\pi \in \Pi$, and add an assertion that encodes the resilience condition, which ensures that the values assigned to the parameters are admissible.

Example 5.3. For the algorithm SAB, whose STA is given in Figure 4.1 on page 104, we have the following declarations and assertion that encode the admissible parameter values, which satisfy the resilience condition $n > 3t \wedge t \geq f$ of the algorithm SAB:

```
(declare-const n Int)
(declare-const t Int)
(declare-const f Int)

; resilience condition
(assert
 (and (> n 0) (>= t 0) (>= f 0) (>= t f) (> n (* 3 t)))
)
```

□

Configurations and Transitions of $Path(\sigma_0, \sigma_{\mathsf{d}+1}, \mathsf{d}+1)$. To encode the existentially quantified configurations and transitions in the formula $Path(\sigma_0, \sigma_{\mathsf{d}+1}, \mathsf{d}+1)$, we proceed as follows. We declare integer constants `c_i_j`, that correspond to the value of the counter $\sigma_i.\boldsymbol{\kappa}[\ell_j]$ in the configuration σ_i, for $0 \leq i \leq \mathsf{d}+1$, and $0 \leq j < |\mathcal{L}|$, and integer constants `t_i_k` corresponding to the factor $tr_i(r_k)$ of rule r_k in the transition tr_i, for $0 < i \leq \mathsf{d}+1$, and $0 \leq k < |\mathcal{R}|$. This captures the existential quantification over configurations and transitions.

To ensure that the declared integer constants correspond to valid configurations, we add assertions which encode the constraints imposed by the environment assumption Env. For the transitions, we encode the following constraints: the factors $tr_i(r_k)$ are non-negative, and the sum of factors in a transition is equal to the number of participating processes.

Then, we add assertions that model the predicate $R(\sigma_i, tr_{i+1}, \sigma_{i+1})$, for $0 \leq i < \mathsf{d}+1$:

- for every rule $r_k \in \mathcal{R}$, we assert that its factor $tr_{i+1}(r_k)$ is 0, if its guard is not satisfied in the configuration σ_i,
- we assert that σ_i is the origin and σ_{i+1} is the goal of tr_{i+1}.

Example 5.4. For the algorithm SAB, we have $|\mathcal{L}| = 4$ locations, and $|\mathcal{R}| = 8$ rules. Below are the constant declarations and the encoding of the environment assumption Env, as well as the constraints on the transitions, in the case when $\mathsf{d} = 2$:

```
; configurations
(declare-const c_0_0 Int)
...
(declare-const c_3_3 Int)

; transitions
(declare-const t_1_0 Int)
...
(declare-const t_3_7 Int)

; environment assumption
(assert
 (and
  ; non-negative counters
  (>= c_0_0 0)
  ...
  (>= c_3_3 0)

  ; sum of counters equals number of participating processes
  (= (+ c_0_0 c_0_1 c_0_2 c_0_3) (- n f))
  ...
  (= (+ c_3_0 c_3_1 c_3_2 c_3_3) (- n f))
 )
)

; transition constraints
(assert
 (and
  ; non-negative factors
  (>= t_1_0 0)
  ...
  (>= t_3_7 0)

  ; sum of factors equals number of participating processes
  (= (+ t_1_0 t_1_1 t_1_2 t_1_3 t_1_4 t_1_5 t_1_6 t_1_7) (- n f))
  ...
  (= (+ t_3_0 t_3_1 t_3_2 t_3_3 t_3_4 t_3_5 t_3_6 t_3_7) (- n f))
 )
)
```

The following assertions model the formula $Path(\sigma_0, \sigma_{\mathsf{d}+1}, \mathsf{d}+1)$ (5.7), for $\mathsf{d} = 2$:

```
(assert
 (and
  ; if r_1.φ is not satisfied in σ_0, then the factor tr_1(r_1) is 0
  (=> (not (< (+ c_0_1 c_0_2 c_0_3) (+ t 1))) (= t_1_0 0))
  ...
  ; if r_8.φ is not satisfied in σ_3, then the factor tr_3(r_8) is 0
  (=> (not (>= (+ (+ c_3_1 c_3_2 c_3_3) f) (- n t))) (= t_3_7 0))
```

```
    ; σ0 is the origin of tr1
    (= (+ t_1_0 t_1_1 t_1_6) c_0_0)
    (= (+ t_1_2 t_1_7) c_0_1)
    (= (+ t_1_3 t_1_4) c_0_2)
    (= t_1_5 c_0_3)
    ...
    ; σ3 is the goal of tr3
    (= t_3_0 c_3_0)
    (= 0 c_3_1)
    (= (+ t_3_1 t_3_2 t_3_3) c_3_2)
    (= (+ t_3_4 t_3_5 t_3_6 t_3_7) c_3_3)
  )
)
```

□

Diameter Query. What remains to encode is the *diameter query*, that is, the universally quantified subformula of (5.8):

$$\forall \sigma'_0, \ldots, \sigma'_{\mathsf{d}}. \forall tr'_1, \ldots, tr'_{\mathsf{d}}. ((\sigma_0 = \sigma'_0) \wedge Path(\sigma'_0, \sigma'_{\mathsf{d}}, \mathsf{d}) \rightarrow \bigwedge\nolimits_{i=0}^{\mathsf{d}} \sigma'_i \neq \sigma_{\mathsf{d}+1}) \quad (5.9)$$

To do so, we introduce:

- universally quantified integer variables `x_i_j`, used to model the counters $\sigma'_i.\boldsymbol{\kappa}[\ell_j]$ in the configuration σ'_i, for $0 \leq i \leq \mathsf{d}$ and $0 \leq j < |\mathcal{L}|$,
- universally quantified integer variables `y_i_j`, used to model the factors $tr'_i(r_k)$, for $0 < i \leq d$ and $0 \leq k < |\mathcal{R}|$,

as well as the necessary constraints to model the implication in (5.9). For the configurations σ'_i and $\sigma_{\mathsf{d}+1}$, where $1 \leq i \leq \mathsf{d}$, we encode that $\sigma'_i \neq \sigma_{\mathsf{d}+1}$ using the disjunction $\bigvee_{j=0}^{|\mathcal{L}|-1}$ `x_i_j` $\neq$ `c_(d + 1)_j`, where `c_(d + 1)_j` encodes the value of the counter $\boldsymbol{\kappa}[\ell_j]$ in $\sigma_{\mathsf{d}+1}$.

Example 5.5. We present the SMT encoding of the diameter query (5.9) for the algorithm SAB and $\mathsf{d} = 2$:

```
(assert
 (forall ((x_0_0 Int) ... (x_2_3 Int) (y_1_0 Int) ... (y_2_7 Int))
  (=>
   ; σ0 = σ'0
   (and
    (= c_0_0 x_0_0)
    (= c_0_1 x_0_1)
    (= c_0_2 x_0_2)
    (= c_0_3 x_0_3)

    ; constraints that encode Path(σ'0, σ'd, d)
```

```
      ...
    )

    ; ⋀_{i=0}^{d} σ'_i ≠ σ_{d+1}
    (and
     ; σ'_0 ≠ σ_3
     (or
      (not (= x_0_0 c_3_0))
      (not (= x_0_1 c_3_1))
      (not (= x_0_2 c_3_2))
      (not (= x_0_3 c_3_3))
     )
     ...
    )
   )
  )
 )
```

By putting together these constraints with the constraints presented in Examples 5.3 and 5.4, we obtain an SMT-LIB file that encodes the formula (5.8) for $\mathsf{d} = 2$ that we can feed into an SMT solver. If the solver outputs `unsat`, then we can conclude that indeed the diameter of SAB is 2. This is confirmed in our experimental evaluation, detailed in Section 5.6. □

5.4 Bounded Diameter for a Fragment of STA

The experiments we conducted in Section 5.6 show that the semi-decision procedure from Section 5.3 works well on our benchmarks, despite the undecidability. A natural question that arises is why is this the case? Intuitively, we believe that the executions of synchronous fault-tolerant distributed algorithms are structured in a way that defines a decidable fragment for the parameterized reachability problem. The theoretical challenge is to characterize this fragment.

In this section, we characterize one class of STA for which we guarantee that a bound on the diameter exists. The STA that fall in this class are *monotonic* and *1-cyclic*, two notions that we will define below. Under these two conditions, we guarantee that for every schedule, there exists a schedule of bounded length that has the same origin and goal. We show that the bound on the length of the latter schedule depends on the c-propositions occurring in the guards of the STA, and the length of the longest path in the STA.

Let $\mathrm{CP}_{\mathcal{R}} \subseteq \mathrm{CP}$ denote the set of c-propositions that occur in the guards of the rules $\mathcal{R}$ of the STA, and let chain denote the length of the longest path in the STA. The main result of this section is stated by the following theorem.

Theorem 5.3. *For every feasible schedule τ in a counter system $\mathsf{CS}(\mathit{STA}, \mathbf{p})$, where STA is monotonic and 1-cyclic, and $\mathbf{p} \in \mathbf{P}_{RC}$, there exists a feasible schedule τ' of length $O(|\mathrm{CP}_{\mathcal{R}}| \cdot \mathsf{chain})$, such that τ and τ' have the same origin and goal.* □

In the remainder of this section, we will introduce the necessary definitions and lemmas that will allow us to prove this theorem.

5.4.1 Monotonic and 1-Cyclic STA

We start by defining monotonic STA.

Definition 5.2 (Monotonic STA)**.** A synchronous threshold automaton STA is *monotonic* iff for every $\mathbf{p} \in \mathbf{P}_{RC}$, every path $\{\sigma_i\}_{i=0}^{k}$ in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, induced by STA and $\mathbf{p}$, and every c-proposition $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \in \mathrm{CP}$, we have $\sigma_i \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ implies $\sigma_j \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$, for every i, j, such that $0 \leq i < j < k$. □

Intuitively, an STA is monotonic iff every c-proposition changes its truth value at most once in every path of a counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, induced by the STA and an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$.

The monotonicity of an STA implies that every feasible schedule in a counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, for $\mathbf{p} \in \mathbf{P}_{RC}$ can be partitioned into finitely many sub-schedules that satisfy a property we call *steadiness*. To show that we can partition a schedule into finitely many sub-schedules, we need the notion of a context.

Definition 5.3 (Context)**.** Given an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$, a *context* of a transition $tr \in \mathit{Tr}(\mathbf{p})$ is the set $\mathcal{C}(tr) = \{\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \in \mathrm{CP} \mid o(tr) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b\}$ of c-propositions $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$, satisfied in the origin $o(tr)$ of the transition tr. □

Definition 5.4 (Context switch)**.** Let $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ be a counter system induced by a given STA and $\mathbf{p} \in \mathbf{P}_{RC}$. For every feasible schedule τ in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, the point i is a *context switch*, if $\mathcal{C}(tr_{i-1}) \neq \mathcal{C}(tr_i)$, where $1 < i \leq |\tau|$. □

The lemma below states that in a monotonic STA, whose c-propositions come from the set $\mathrm{CP}_{\mathcal{R}}$, there are at most $|\mathrm{CP}_{\mathcal{R}}|$ context switches in every feasible schedule of the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, for $\mathbf{p} \in \mathbf{P}_{RC}$.

Lemma 5.1. *Let $\mathsf{CS}(\mathit{STA}, \mathbf{p})$ be a counter system induced by a given monotonic STA and $\mathbf{p} \in \mathbf{P}_{RC}$. Every feasible schedule τ in the counter system $\mathsf{CS}(\mathit{STA}, \mathbf{p})$ has at most $|\mathrm{CP}_{\mathcal{R}}|$ context switches.*

Proof. Let $\tau = \{tr_i\}_{i=1}^{k}$ be a feasible schedule and $\mathrm{CP}_{\mathcal{R}}$ the set of c-propositions appearing on the rules $\mathcal{R}$ of the monotonic STA. For every $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \in \mathrm{CP}_{\mathcal{R}}$, there is at most one context switch i, for $0 < i \leq k$, such that $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \notin \mathcal{C}(tr_{i-1})$ and $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \in \mathcal{C}(tr_i)$, hence the total number of context switches is at most $|\mathrm{CP}_{\mathcal{R}}|$. □

We now define the notion of steadiness. We call a schedule *steady* if the set of rules whose guards are satisfied does not change in all of its transitions.

Definition 5.5 (Steady schedule)**.** Let $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ be a counter system induced by a given STA and $\mathbf{p} \in \mathbf{P}_{RC}$. A schedule $\tau = \{tr_i\}_{i=1}^{k}$ in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ is *steady*, if $\mathcal{C}(tr_i) = \mathcal{C}(tr_j)$, for $0 < i < j \leq k$.

The following proposition states a property of steady schedules in counter systems induced by monotonic STA, and is a consequence of the monotonicity and Definition 5.5. It states that if the context of the first and the last transition of a schedule in a counter system induced by a monotonic STA are the same, then the schedule is steady.

Proposition 5.1. *Let STA be monotonic, $\mathbf{p} \in \mathbf{P}_{RC}$, and $\tau = \{tr_i\}_{i=1}^{k}$ a schedule in $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$. If $\mathcal{C}(tr_1) = \mathcal{C}(tr_k)$, then τ is a steady schedule.*

Example 5.6. Consider the path of length $\mathsf{t}+2$ in a counter system induced by the STA of the algorithm SAB from Example 5.2. The schedule τ, associated with this path is the schedule $\tau = \{tr_i\}_{i=1}^{\mathsf{t}+2}$. Let L denote the set $\{\text{V1}, \text{SE}, \text{AC}\}$ of locations in STA. The transitions in the schedule τ have the following contexts:

- for the transitions tr_i, for $1 \leq i \leq \mathsf{t}$, we have $\mathcal{C}(tr_i) = \{\#L \geq t+1-f\}$,
- for the transition $tr_{\mathsf{t}+1}$, we have $\mathcal{C}(tr_{\mathsf{t}+1}) = \{\#L \geq t+1-f, \#L \geq t+1\}$,
- for the transition $tr_{\mathsf{t}+2}$, we have $\mathcal{C}(tr_{\mathsf{t}+2}) = \{\#L \geq t+1-f, \#L \geq t+1, \#L \geq n-t-f\}$.

Thus, the schedule τ has two context switches: at positions $\mathsf{t}+1$ and $\mathsf{t}+2$. The sub-schedule $\tau' = \{tr_i\}_{i=1}^{\mathsf{t}}$ of τ is steady: all its transitions have the same context, that contains the c-proposition $\#L \geq t+1-f$. □

Observe that for an STA, we can build a directed graph $G_{\mathsf{STA}} = (V_{\mathsf{STA}}, E_{\mathsf{STA}})$ where $V_{\mathsf{STA}} = \mathcal{L}$, and where $(\ell, \ell') \in E_{\mathsf{STA}}$ iff there exists a rule $r \in \mathcal{R}$, such that $\ell = r.from$, $\ell' = r.to$, and $\ell \neq \ell'$. That is, the directed graph does not contain edges corresponding to the self-loops in the STA. The following two definitions define notions using the directed graph G_{STA} of a given STA.

Definition 5.6 (Longest chain)**.** Given an STA, let $G_{\mathsf{STA}} = (V_{\mathsf{STA}}, E_{\mathsf{STA}})$ be its corresponding directed graph. We call the length of the longest path between two nodes in the graph G_{STA} the *longest chain* of STA, and denote it by chain. □

Definition 5.7 (1-cyclic STA)**.** Given an STA, let $G_{\mathsf{STA}} = (V_{\mathsf{STA}}, E_{\mathsf{STA}})$ be its corresponding directed graph. We call the STA *1-cyclic* if its corresponding directed graph G_{STA} is acyclic. □

As a consequence of Definition 5.7, in a 1-cyclic STA, the only cycles that can be formed by its rules are self-loops, that is, the STA contains only cycles of length one.

Example 5.7. For the STA of the algorithm SAB, whose graph G_{STA} is depicted in Figure 4.1, we have that the longest chain is $\mathsf{chain} = 2$. Furhtermore, this STA is 1-cyclic, as its only cycles are self-loops. □

5.4.2 Sufficient Condition for Monotonicity

We now present a sufficient condition for monotonicity of STA. We introduce the notion of *trapped* c-propositions, and show that given an STA, if its c-propositions from the set $\mathrm{CP}_{\mathcal{R}}$ are trapped, then the STA is monotonic.

Definition 5.8 (Trap)**.** A set $L \subseteq \mathcal{L}$ of locations is called a *trap*, iff for every $\ell \in L$ and every $r \in \mathcal{R}$ such that $\ell = r.from$, it holds that $r.to \in L$.

A c-proposition $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \in \mathrm{CP}_{\mathcal{R}}$ is *trapped* iff the set L is a trap. □

Example 5.8. Consider the STA of the algorithm SAB, depicted in Figure 4.1 on page 104. Its set $\mathrm{CP}_{\mathcal{R}}$ of c-propositions contains four c-propositions:

1. $\#\{\text{V1}, \text{SE}, \text{AC}\} \geq t + 1$, as $\varphi_1 \equiv \neg(\#\{\text{V1}, \text{SE}, \text{AC}\} \geq t + 1)$,
2. $\#\{\text{V1}, \text{SE}, \text{AC}\} \geq t + 1 - f$, as $\varphi_2 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \geq t + 1 - f$,
3. $\#\{\text{V1}, \text{SE}, \text{AC}\} \geq n - t$, as $\varphi_3 \equiv \neg(\#\{\text{V1}, \text{SE}, \text{AC}\} \geq n - t)$,
4. $\#\{\text{V1}, \text{SE}, \text{AC}\} \geq n - t - f$, as $\varphi_4 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \geq n - t - f$.

All four c-propositions check the number of processes in locations $L = \{\text{V1}, \text{SE}, \text{AC}\}$. The set L is a trap, since there are no rules, outgoing of either V1, SE, of AC, that can be used by a process to move outside of L. □

The following lemma states that once a trapped c-proposition holds in a configuration, it also holds in its immediate successor.

Lemma 5.2. *Let* $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ *be a counter system induced by a given* STA *and* $\mathbf{p} \in \mathbf{P}_{RC}$. *Let* $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ *be a trapped c-proposition occurring in a guard of a rule in the* STA, *let* σ *be a configuration in the counter system* $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ *such that* $\sigma \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$, *and let* tr *be a transition enabled in* σ. *If* $\sigma \xrightarrow{tr} \sigma'$, *then* $\sigma' \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$.

Proof. Suppose that $\sigma \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$, where L is a trap. Then, by the semantics of the c-propositions, we have that $\sum_{\ell \in L} \sigma.\boldsymbol{\kappa}[\ell] \geq \boldsymbol{a} \cdot \mathbf{p} + b$. Because L is a trap, by Definitions 4.9 and 5.8, we have

$$\sum_{\ell \in L} \sigma'.\boldsymbol{\kappa}[\ell] = \sum_{r \in \mathcal{R} \wedge r.to \in L} tr(r) \geq \sum_{r \in \mathcal{R} \wedge r.from \in L} tr(r) = \sum_{\ell \in L} \sigma.\boldsymbol{\kappa}[\ell]$$

Thus, $\sum_{\ell \in L} \sigma'.\boldsymbol{\kappa}[\ell] \geq \boldsymbol{a} \cdot \mathbf{p} + b$, and $\sigma' \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$. □

As a consequence of Lemma 5.2, we have that once a c-proposition $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ holds in some configuration, it holds in all of its successors. The following corollary states that an STA, whose c-propositions are trapped, is monotonic (Definition 5.2).

Corollary 5.1. *Let STA be an automaton such that all c-propositions that occur in the guards on its rules are trapped. Then STA is monotonic.* □

In Example 5.8 we saw that all the c-propositions that occur on the guards of the rules of the STA of the algorithm SAB are trapped. Hence, by Corollary 5.1, the STA of the algorithm SAB is monotonic. However, this is not always the case for all our benchmarks, as we show in the following example.

Example 5.9. Consider the STA of the algorithm FloodMin, for $k = 1$, in Figure 4.2 on page 111. Its set $\mathrm{CP}_{\mathcal{R}}$ of c-propositions contains two c-propositions: $\#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \geq 1$ and $\#\{\mathrm{v0}\} \geq 1$ (recall that $\varphi_1 \equiv \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \geq 1$ and $\varphi_2 \equiv \neg(\#\{\mathrm{v0}\} \geq 1)$). We observe that both c-propositions are not trapped, since:

1. for $\#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \geq 1$, there is a rule r_7, outgoing of $\mathrm{v0}_{\mathsf{cr}}$, such that $r_7.to \equiv \maltese \notin \{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\}$,
2. for $\#\{\mathrm{v0}\} \geq 1$, there is a rule r_4, outgoing of $\mathrm{v0}$, such that $r_4.to \equiv \mathrm{v0}_{\mathsf{cr}} \notin \{\mathrm{v0}\}$. □

Although the c-propositions occurring on the guards of the STA for the algorithm FloodMin, for $k = 1$, are not trapped, and hence we cannot use Corollary 5.1 to conclude whether the STA is monotonic, applying our semi-decision procedure from Section 5.3 in the experimental evaluation in Section 5.6 shows that a bound on the diameter for this STA exists. Finding a characterization that guarantees a theoretical bound on the diameter for a class of algorithms whose STA have the same features as the STA for the algorithm FloodMin is subject to future work.

5.4.3 Shortening Feasible Schedules

We now focus on shortening feasible schedules in counter systems induced by monotonic and 1-cyclic STA. More precisely, given a feasible schedule, we construct a feasible schedule of bounded length, with the same origin and goal.

The main idea of the shortening is the following. Let τ be a feasible schedule in a counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, induced by a monotonic and 1-cyclic STA, and $\mathbf{p} \in \mathbf{P}_{RC}$. As STA is monotonic, by Lemma 5.1, the schedule τ has at most $|\mathrm{CP}_{\mathcal{R}}|$ context switches, where $\mathrm{CP}_{\mathcal{R}}$ is the set of c-propositions occurring on the guards of the rules of the monotonic STA. The context switches break the schedule τ into chunks, which are in fact steady feasible schedules. If there are $\mathsf{cs} \leq |\mathrm{CP}_{\mathcal{R}}|$ context switches in the schedule τ, then the schedule τ is a concatenation of $\mathsf{cs} + 1$ steady feasible schedules, i.e., $\tau = \tau_0\tau_1 \ldots \tau_{\mathsf{cs}}$ (recall Example 5.6).

Consider a steady feasible schedule τ_i, for $0 \leq i \leq \mathsf{cs}$, which is a part of the original schedule τ. We construct a shorter schedule, corresponding to the schedule τ_i as follows. If the length $|\tau_i|$ of the steady feasible schedule τ_i is greater than the length of the longest path in the automaton STA, then in some transition of the schedule τ_i, some processes followed a rule which is a self-loop. As processes may follow self-loops at different transitions, we cannot shorten the schedule τ_i by eliminating transitions as a whole. Instead, we deconstruct the schedule τ_i into sequences of process steps, which we call *local runs*, shorten the local runs, and reconstruct a new shorter schedule from the shortened local runs. The main challenge is to show that the newly obtained schedule is feasible and steady.

In the following, we fix a monotonic and 1-cyclic STA, as well as an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$, and assume that the schedules we shorten come from the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$.

Schedules as Multisets of Local Runs. We proceed by defining local runs and showing that each schedule can be represented by a multiset of local runs.

Definition 5.9 (Local run)**.** A *local run* is a sequence $\varrho = \{r_i\}_{i=1}^{k}$ of rules, for $r_i \in \mathcal{R}$, such that $r_i.to = r_{i+1}.from$, for $0 < i < k$.

We denote by $\varrho[i] = r_i$ the i-th rule in the local run ϱ, and by $|\varrho|$ the *length* of the local run ϱ. □

The following lemma shows that a feasible schedule can be deconstructed into a multiset of local runs.

Lemma 5.3. *For every feasible schedule $\tau = \{tr_i\}_{i=1}^{k}$, there exists a multiset $(\mathcal{P}, \mathsf{mult})$, where:*

1. *$\mathcal{P}$ is a set of local runs ϱ of length k, and*
2. *$\mathsf{mult} : \mathcal{P} \to \mathbb{N}$ is a multiplicity function, such that for every location $\ell \in \mathcal{L}$, it holds that $\sum_{r.from=\ell} tr_i(r) = \sum_{\varrho[i].from=\ell} \mathsf{mult}(\varrho)$, for $0 < i \leq k$.*

Proof. We proceed by induction on the length of the schedule.

Induction base. Let τ be a schedule of length one, that is, τ consists of a single transition tr_1. By the definition of a transition, for every $r \in \mathcal{R}$, the factor $tr_1(r)$ denotes the number of processes that follow rule r in the transition tr_1. We define the multiset $(\mathcal{P}, \mathsf{mult})$ by setting $\mathcal{P} = \{r \in \mathcal{R} \mid tr_1(r) \neq 0\}$ to be the set of rules that have non-zero factors in tr_1, and for every $r \in \mathcal{P}$, we set $\mathsf{mult}(r) = tr_1(r)$.

Induction step. Consider a schedule $\tau = \{tr_i\}_{i=1}^{k+1}$ of length $k+1$. For the prefix $\tau' = \{tr_i\}_{i=1}^{k}$ of τ, which is a feasible schedule of length k, we have, by the induction hypothesis, that there exists a multiset $(\mathcal{P}', \mathsf{mult}')$ such that $\mathcal{P}'$ is a set of local runs of length k, and for every location $\ell \in \mathcal{L}$ it holds that $\sum_{r.from=\ell} tr_i(r) = \sum_{\varrho'[i].from=\ell} \mathsf{mult}'(\varrho')$, for $0 < i \leq k$ and $\varrho' \in \mathcal{P}'$. Let $\sigma_k = g(tr_k)$ be the goal configuration of the transition tr_k. For every $\ell \in \mathcal{L}$, it holds that $\sigma_k.\boldsymbol{\kappa}[\ell] = \sum_{r.to=\ell} tr_k(r)$, by Definition 4.14. Observe that $\sigma_k.\boldsymbol{\kappa}[\ell]$ also represents the number of local runs that end in a rule that points to the location $\ell \in \mathcal{L}$, that is,

$$\sigma_k.\boldsymbol{\kappa}[\ell] = \sum_{r.to=\ell} tr_k(r) = \sum_{\varrho'[k].to=\ell} \mathsf{mult}'(\varrho') \tag{5.10}$$

Given the transition tr_{k+1}, let $\mathcal{R}_{k+1} = \{r \in \mathcal{R} \mid tr_{k+1}(r) \neq 0\}$ be the set of rules that have non-zero factors in tr_{k+1}. We define the set

$$\mathcal{P} = \{\varrho' r \mid \varrho' \in \mathcal{P}', r \in \mathcal{R}_{k+1}, \text{ and } \varrho'[k].to = r.from\}$$

of local runs of length $k+1$, where $\varrho' r$ is the local run obtained by appending r to ϱ'. We define the function mult that maps local runs $\varrho = \varrho' r$ from the set $\mathcal{P}$ such that for every $\ell \in \mathcal{L}$, it holds that $\sum_{\varrho'[k].to=\ell} \mathsf{mult}'(\varrho') = \sum_{r.from=\ell} \mathsf{mult}(\varrho' r)$.

We now check that the multiset $(\mathcal{P}, \mathsf{mult})$ satisfies the two properties. Clearly, the set $\mathcal{P}$ contains local runs of length $k+1$. To show the second property, we use the assumption that τ is a feasible schedule. Hence, for every $\ell \in \mathcal{L}$, it holds that

$$\sum_{r.from=\ell} tr_{k+1}(r) = \sum_{r.to=\ell} tr_k(r) = \sum_{\varrho'[k].to=\ell} \mathsf{mult}'(\varrho') = \sum_{r.from=\ell} \mathsf{mult}(\varrho' r)$$

by Proposition 4.1, (5.10), and the construction, respectively. □

We can also easily translate back from a multiset $(\mathcal{P}, \mathsf{mult})$ of local runs of length k to a schedule $\tau = \{tr_i\}_{i=1}^{k}$ of length k. That is, for every rule $r \in \mathcal{R}$ and $0 < i \leq k$, we can define $tr_i(r) = \sum_{\varrho[i]=r} \mathsf{mult}(\varrho)$, and obtain the schedule τ of length k.

Example 5.10. Recall the path of length $\mathsf{t}+2$ in a counter system induced by the STA of the algorithm SAB from Example 5.4, and its associated schedule τ from Example 5.6. We give the decomposition of the schedule τ into a multiset $(\mathcal{P}, \mathsf{mult})$ of local runs below:

$\mathcal{P}$, for τ	tr_1	tr_2	tr_3	...	tr_{t}	$tr_{\mathsf{t}+1}$	$tr_{\mathsf{t}+2}$	mult, for τ
ϱ_1	r_3	r_4	r_4	...	r_4	r_4	r_5	1
ϱ_2	r_2	r_4	r_4	...	r_4	r_4	r_5	1
ϱ_3	r_1	r_2	r_4	...	r_4	r_4	r_5	1
...				...				...
ϱ_{t}	r_1	r_1	r_1	...	r_4	r_4	r_5	1
$\varrho_{\mathsf{t}+1}$	r_1	r_1	r_1	...	r_2	r_4	r_5	1
$\varrho_{\mathsf{t}+2}$	r_1	r_1	r_1	...	r_1	r_2	r_5	$N(\mathbf{p}) - (\mathsf{t}+1)$

There are $\mathsf{t}+2$ local runs of length $\mathsf{t}+2$ in the set $\mathcal{P}$, where:

- the local run ϱ_1 is the sequence of rules applied by the single process, initially in location V1, in order to:
  1. move to location SE in the transition tr_1,
  2. stay in location SE for t transitions,
  3. move to the location AC in the transition $tr_{\mathsf{t}+2}$,
- the local run ϱ_i, for $2 \leq i \leq \mathsf{t}+1$ is the sequence of rules applied by the single process, initially in location V0, in order to:
  1. stay in location V0 in the first $i-2$ transitions,
  2. move to location SE in the transition tr_{i-1},
  3. stay in location SE for $\mathsf{t}-(i-2)$ transitions,
  4. move to the location AC in the transition $tr_{\mathsf{t}+2}$,
- the local run $\varrho_{\mathsf{t}+2}$ is the sequence of rules applied by the $N(\mathbf{p})-(\mathsf{t}+1)$ processes, initially in location V0, in order to:
  1. stay in the location V0 in the first t transitions,
  2. move to the location SE in the transition $tr_{\mathsf{t}+1}$,
  3. move to the location AC in the transition $tr_{\mathsf{t}+2}$.

To check the second condition in Lemma 5.3, we look at the transition tr_1, where one process applies the rule r_2, one process applies the rule r_3, and $N(\mathbf{p})-2$ processes apply the rule r_1. From $r_1.\mathit{from} = r_2.\mathit{from} = \text{V0}$, we have that $\sum_{r.\mathit{from}=\text{V0}} tr_1(r) = N(\mathbf{p})-1$. In the set of local runs $\mathcal{P}$, we have $\varrho_i[1] = \text{V0}$, for $2 \leq i \leq \mathsf{t}+2$. Thus, $\sum_{\varrho[1]=\text{V0}} \mathsf{mult}(\varrho) = N(\mathbf{p})-1$. Similarly, from $r_3.\mathit{from} = \text{V1}$, we have $\sum_{r.\mathit{from}=\text{V1}} tr_1(r) = 1$, and as ϱ_1 is the only run where $\varrho_1[1] = \text{V1}$, we get $\sum_{\varrho[1]=\text{V1}} \mathsf{mult}(\varrho) = 1$. □

Self-Loops in Feasible Schedules. We now show that for every feasible schedule τ, whose length is longer than the longest chain (Definition 5.6), every local run contains a self-loop.

Lemma 5.4. *Let $\tau = \{tr_i\}_{i=1}^{k}$ be a feasible schedule, and $(\mathcal{P}, \mathsf{mult})$ its corresponding multiset of local runs of length $|\tau|$. If $|\tau| > \mathsf{chain}$, where chain is the longest chain in the STA, then every local run $\varrho \in \mathcal{P}$ contains a rule $r \in \mathcal{R}$ such that $r.\mathit{from} = r.\mathit{to}$.*

Proof. We prove this lemma by contradiction.

Let $\tau = \{tr_i\}_{i=1}^{k}$ be a feasible schedule, and let $\mathcal{P}$ denote its corresponding set of local runs, such that the length of each run $\varrho \in \mathcal{P}$ is equal to the length of the schedule, that

is, $|\tau| = |\varrho| = k$, for every $\varrho \in \mathcal{P}$. Suppose that $k > \mathsf{chain}$ and that there exists a local run $\varrho \in \mathcal{P}$ that does not contain a self-loop rule $r \in \mathcal{R}$, where $r.from = r.to$.

For this local run ϱ and for every $0 < i \leq k$, it holds that $\varrho[i].from \neq \varrho[i].to$. Since chain is the longest chain in the 1-cyclic STA, and since $k > \mathsf{chain}$, it must be the case that there exist indices i, j, with $0 < i < j \leq k$, such that $\varrho[i].from = \varrho[j].to$. This implies that the STA contains a cycle which is not a self-loop, which is a contradiction to it being 1-cyclic. □

Example 5.11. Recall the decomposition of the schedule τ into a multiset $(\mathcal{P}, \mathsf{mult})$ of local runs from Example 5.10. As shown in Example 5.7, the longest chain for the STA of the algorithm SAB (Figure 4.1) is $\mathsf{chain} = 2$. Suppose $\mathsf{t} > 0$, then the local runs in the set $\mathcal{P}$ are of length $\mathsf{t} + 2 > \mathsf{chain}$. To check Lemma 5.4, observe that:

- the local run ϱ_i, for $1 \leq i \leq \mathsf{t}+1$ contains the rule r_4, such that $r_4.from = r_4.to = \text{SE}$,
- the local run $\varrho_{\mathsf{t}+2}$ contains the rule r_1, such that $r_1.from = r_1.to = \text{V0}$.

Observe that while for the local runs ϱ_i, for $1 \leq i \leq \mathsf{t}+1$, we have $\varrho_i[\mathsf{t}+1] = r_4$, we cannot simply remove the whole transition $tr_{\mathsf{t}+1}$, since in the run $\varrho_{\mathsf{t}+2}$, we have $\varrho_{\mathsf{t}+2}[\mathsf{t}+1] = r_2$, which is not a self-loop rule. This justifies the necessity to decompose a schedule into local runs, shorten the local runs, and then compose a new shorter schedule from the shorter runs. □

Constructing Shorter Schedules. We now show that given a steady feasible schedule of arbitrary length, we can construct a shorter feasible schedule, such that the length of the shorter schedule does not exceed chain, the longest chain of the underlying STA. Before formally show how we can construct a shorter schedule, we show the construction on an example.

Example 5.12. Recall the steady sub-schedule τ' of the schedule τ, from Example 5.6. That is, $\tau' = \{tr_i\}_{i=1}^{\mathsf{t}}$. Suppose that $\mathsf{t} > 3$, that is, suppose that $|\tau'| > \mathsf{chain} + 1$ (as longest chain for the STA of the algorithm SAB is $\mathsf{chain} = 2$). A decomposition into a multiset $(\mathcal{P}', \mathsf{mult}')$ of local runs is given below:

$\mathcal{P}'$	tr_1	tr_2	tr_3	...	$tr_{\mathsf{t}-2}$	$tr_{\mathsf{t}-1}$	tr_{t}	mult'
ϱ_1	r_3	r_4	r_4	...	r_4	r_4	r_4	1
ϱ_2	r_2	r_4	r_4	...	r_4	r_4	r_4	1
ϱ_3	r_1	r_2	r_4	...	r_4	r_4	r_4	1
...				...				...
$\varrho_{\mathsf{t}-2}$	r_1	r_1	r_1	...	r_1	r_2	r_4	1
$\varrho_{\mathsf{t}-1}$	r_1	r_1	r_1	...	r_1	r_1	r_2	1
ϱ_{t}	r_1	r_1	r_1	...	r_1	r_1	r_1	$N(\mathbf{p}) - (\mathsf{t}+1)$

The goal is to use the schedule τ' to construct a shorter schedule τ'', such that both schedules have the same origin and goal. To do so, we construct τ'' as follows. Consider the prefix $\theta = \{tr_i\}_{i=1}^{\mathsf{t}-1}$ of τ' of length $\mathsf{t}-1$, which is also a steady and feasible schedule. As $|\theta| = \mathsf{t}-1 > \mathsf{chain}$, by Lemma 5.4, every local run in θ has a self-loop rule. To construct a shorter schedule from θ, we proceed by removing one occurrence of a self-loop rule in every local run, e.g.: the first occurrence of the self-loop rule r_4 from the local runs ϱ_1 and ϱ_2, and the first occurrence of the self-loop rule r_1 from the local runs $\varrho_3, \ldots, \varrho_{\mathsf{t}+2}$. By composing the local runs again, we obtain a schedule θ', of length $|\theta'| = \mathsf{t}-2$, to which we append the transition tr_{t}, and obtain the shorter schedule τ'', whose decomposition into a multiset $(\mathcal{P}'', \mathsf{mult}'')$ of local runs is given below:

$\mathcal{P}''$	tr'_1	tr'_2	tr'_3	$\ldots$	$tr'_{\mathsf{t}-2}$	$tr'_{\mathsf{t}-1}$	mult''
ϱ'_1	r_3	r_4	r_4	$\ldots$	r_4	r_4	1
ϱ'_2	r_2	r_4	r_4	$\ldots$	r_4	r_4	2
ϱ'_3	r_1	r_2	r_4	$\ldots$	r_4	r_4	1
$\ldots$			$\ldots$				$\ldots$
$\varrho'_{\mathsf{t}-1}$	r_1	r_1	r_1	$\ldots$	r_2	r_4	1
ϱ'_{t}	r_1	r_1	r_1	$\ldots$	r_1	r_2	1
$\varrho'_{\mathsf{t}+1}$	r_1	r_1	r_1	$\ldots$	r_1	r_1	$N(\mathbf{p}) - (\mathsf{t}+1)$

Observe that by removing the rule r_4 from the local run ϱ_2, and the rule r_1 from the local run ϱ_3, we obtain the same local run, ϱ'_2, and thus we have $\mathsf{mult}''(\varrho'_2) = \mathsf{mult}'(\varrho_1) + \mathsf{mult}'(\varrho_2) = 2$. The multiplicities for the other runs remain the same. □

The following lemma shows that for a given steady feasible schedule τ, such that $|\tau| > \mathsf{chain}+1$, it is always possible to construct a shorter steady and feasible schedule τ' with the same origin and goal, of length $|\tau| - 1$.

Lemma 5.5. *Let τ be a steady feasible schedule. If $|\tau| > \mathsf{chain}+1$, where chain is the longest chain in the STA, then there exists a steady feasible schedule τ' such that $|\tau'| = |\tau| - 1$, and the schedules τ and τ' have the same origin and goal.*

Proof. Suppose $\tau = \{tr_i\}_{i=1}^{k+1}$, with $|\tau| = k+1 > \mathsf{chain}+1$, is a steady schedule. Then, by Definition 5.5, $\mathcal{C}(tr_1) = \mathcal{C}(tr_k)$. Let $\theta = \{tr_i\}_{i=1}^{k}$ be the prefix of τ of length $|\theta| = k > \mathsf{chain}$, which is also a steady and feasible schedule.

By Lemma 5.3, for θ, there exists a multiset $(\mathcal{P}, \mathsf{mult})$ of local runs of length k that describes it. Since $k > \mathsf{chain}$, by Lemma 5.4, every local run $\varrho \in \mathcal{P}$ contains a rule r which is a self-loop, that is, $r.from = r.to$.

Construct a set $\mathcal{P}'$ of local runs of length $k-1$, such that every $\varrho' \in \mathcal{P}'$ is obtained from some $\varrho \in \mathcal{P}$ by removing one occurrence of a self-loop rule. Given a local run $\varrho' \in \mathcal{P}'$ of length $k-1$, denote by $\mathcal{P}(\varrho')$ the set of local runs $\varrho \in \mathcal{P}$ of length k such that ϱ' was obtained by removing exactly one occurrence of a self-loop rule in ϱ. For every i,

where $0 < i < k$, and $\varrho \in \mathcal{P}(\varrho')$, it holds that either $\varrho[i] = \varrho'[i]$, if the self-loop rule in run ϱ is at position $j > i$, or $\varrho[i+1] = \varrho'[i]$ otherwise. Construct a multiplicity function $\mathsf{mult}' : \mathcal{P}' \to \mathbb{N}$, such that for every $\varrho' \in \mathcal{P}'$, we have $\mathsf{mult}'(\varrho') = \sum_{\varrho \in \mathcal{P}(\varrho')} \mathsf{mult}(\varrho)$.

The multiset $(\mathcal{P}', \mathsf{mult}')$ defines a schedule $\theta' = \{tr'_i\}_{i=1}^{k-1}$ of length $k-1$. We now show that θ' is feasible (Definition 4.19 on page 117), and that it has the same origin and goal as θ.

To show that θ' is feasible, by Proposition 4.1 on page 117, it suffices to show that for every i, for $1 < i < k$, and every $\ell \in \mathcal{L}$, we have $\sum_{r.to=\ell} tr'_{i-1}(r) = \sum_{r.from=\ell} tr'_i(r)$. By the definition of $\mathcal{P}(\varrho')$, we can associate to every local run $\varrho \in \mathcal{P}(\varrho')$ an index j_ϱ, which denotes the position of the self-loop rule that was removed from ϱ in order to obtain ϱ'. Thus:

$$\sum_{\varrho'[i].from=\ell} \mathsf{mult}'(\varrho') = \sum_{\substack{\varrho[i].from=\ell \\ j_\varrho > i}} \mathsf{mult}(\varrho) + \sum_{\substack{\varrho[i+1].from=\ell \\ j_\varrho \leq i}} \mathsf{mult}(\varrho)$$
$$= \sum_{\substack{\varrho[i-1].to=\ell \\ j_\varrho > i}} \mathsf{mult}(\varrho) + \sum_{\substack{\varrho[i].to=\ell \\ j_\varrho \leq i}} \mathsf{mult}(\varrho) = \sum_{\varrho'[i-1].to=\ell} \mathsf{mult}'(\varrho')$$

which implies $\sum_{r.from=\ell} tr'_i(r) = \sum_{r.to=\ell} tr'_{i-1}(r)$. This and Proposition 4.1 give us the feasibility of θ'.

To show that θ and θ' have the same origin, we will show that for every $\ell \in \mathcal{L}$, we have $o(tr_1).\boldsymbol{\kappa}[\ell] = o(tr'_1).\boldsymbol{\kappa}[\ell]$. Let $\ell \in \mathcal{L}$ be a location. W.l.o.g., suppose that there is one local run $\varrho^* \in \mathcal{P}$ whose first rule originates in ℓ and is a self-loop, that is, $\varrho^*[1].from = \ell$ and $\varrho^*[1].from = \varrho^*[1].to$, such that the self-loop rule $\varrho^*[1]$ was removed in order to obtain ϱ'. Thus, we have that $\varrho'[1].from = \varrho^*[2].from$. As all the other local runs remain unchanged in the first rule, we have that:

$$\sum_{\varrho'[1].from=\ell} \mathsf{mult}'(\varrho') = \mathsf{mult}(\varrho^*) + \sum_{\substack{\varrho[1].from=\ell \\ \varrho \neq \varrho^*}} \mathsf{mult}(\varrho) = \sum_{\varrho[1].from=\ell} \mathsf{mult}(\varrho)$$

which implies $\sum_{r.from=\ell} tr'_1(r) = \sum_{r.from=\ell} tr_1(r)$, and hence $o(tr'_1).\boldsymbol{\kappa}[\ell] = o(tr_1).\boldsymbol{\kappa}[\ell]$.

To show that θ and θ' have the same goal, we follow similar reasoning.

Observe that the goal of θ is the origin of the transition tr_{k+1} in the schedule τ. Since θ and θ' have the same goal configurations, we can append $tr'_k = tr_{k+1}$ to the schedule θ' and obtain a new schedule $\tau' = \{tr'_i\}_{i=0}^{k}$. The following holds for the schedule τ':

- it is feasible, since θ' is feasible and $tr'_k = tr_{k+1}$ is applicable in the goal of θ';
- it is steady, because $o(tr_1) = o(tr'_1)$ and $o(tr_{k+1}) = o(tr'_k)$, hence the contexts of tr'_1 and tr'_k are equal, since τ is steady and the contexts of tr_1 and tr_{k+1} are equal. The steadiness of τ' follows from this and Proposition 5.1;

- $|\tau'| = |\theta'| + 1 = k = |\tau| - 1$
- τ and τ' have the same origin, as $o(tr_1) = o(tr'_1)$;
- τ and τ' have the same goal, as $g(tr_{k+1}) = g(tr'_k)$.

This completes the proof. □

As a consequence of Lemma 5.1 and 5.5, we obtain Theorem 5.3, restated below:

Theorem 5.3. *For every feasible schedule τ in a counter system $\mathsf{CS}(\mathit{STA}, \mathbf{p})$, where STA is monotonic and 1-cyclic, and $\mathbf{p} \in \mathbf{P}_{RC}$, there exists a feasible schedule τ' of length $O(|\mathrm{CP}_{\mathcal{R}}| \cdot \mathsf{chain})$, such that τ and τ' have the same origin and goal.* □

Theorem 5.3 tells us that for any feasible schedule, there exists a feasible schedule of length $O(|\mathrm{CP}_{\mathcal{R}}| \cdot \mathsf{chain})$. This bound does not depend on the parameters, but on the number of context switches and the longest chain chain, which are properties of the STA.

5.5 Bounded Model Checking of Safety Properties

Recall that in Section 5.1 we reduced the parameterized model checking problem PMCP of safety properties for our benchmarks to the parameterized reachability problem PRP. In Section 5.3, we introduced a semi-decision procedure for computing the diameter, while in Section 5.4 we showed that a bound on the diameter exists for a class of STA. Assuming we know the bound on the diameter d, we can further reduce the parameterized reachability problem to the following *parameterized bounded reachability problem* PBRP:

Parameterized Bounded Reachability PBRP

INPUT:
- a synchronous threshold automaton STA
- a Boolean combination ϕ_{init} of c-propositions
- a Boolean combination ϕ_{reach} of c-propositions
- a bound $k \in \mathbb{N}$

QUESTION: Do there exist:

- an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$,
- an initial configuration σ_0 in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$,
- a feasible schedule τ in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$,

with $\sigma_0 \models \phi_{\mathsf{init}}$, $\sigma_0 = o(\tau)$, and $|\tau| = k$, such that $g(\tau) \models \phi_{\mathsf{reach}}$?

The following theorem states the reduction from PRP to PBRP, and is the analog of Theorem 9 from [BCCZ99]:

Theorem 5.4 (PRP to PBRP)**.** *Let STA be a synchronous threshold automaton, and let $\phi_{\mathsf{init}}, \phi_{\mathsf{reach}}$ be Boolean combinations of c-propositions. $\mathit{PRP}(\mathit{STA}, \phi_{\mathsf{init}}, \phi_{\mathsf{reach}})$ has a*

positive answer iff there exists a bound $k \in \mathbb{N}$ *for which* $PBRP(\mathsf{STA}, \phi_{\mathsf{init}}, \phi_{\mathsf{reach}}, k)$ *has a positive answer.* □

By Theorem 5.4, a safety property ϕ of the shape $\phi \equiv \mathsf{init} \rightarrow \mathbf{G}(\mathsf{global})$ holds in a parameterized counter system $\mathsf{CS}(\mathsf{STA}, \boldsymbol{\pi})$ iff there does not exist a schedule of bounded length, depending on the diameter bound d, whose origin satisfies init, but whose goal does not satisfy global. Thus, once we obtain the diameter bound d (either using the procedure from Section 5.3 or by Theorem 5.3), we use it to search for violations of safety properties in executions of bounded length. The length of the bounded executions depends on d and on the way in which the algorithms are designed. In the literature, we found two kinds of benchmarks:

1. algorithms that are designed to run for a parameterized number of rounds, and that assume the existence of a clean round, such as the algorithm FloodMin, whose pseudocode is given in Figure 2.1 on page 40,
2. algorithms that are designed to run for infinitely many rounds, and that do not assume the clean round condition, such as the algorithm SAB, whose pseudocode is given in Figure 4.1 on page 104.

We will discuss the specific SMT queries, which we produce for the bounded executions for each of these two classes of algorithms in Sections 5.5.1 and 5.5.2, respectively.

5.5.1 Algorithms with a Clean Round Assumption

Many of our benchmarks run for $t+1$ rounds and are designed for $t \geq f$ faults. This design feature ensures the correctness of the algorithm as follows: by having at least one *clean* round (among the $t+1$ rounds) in which there were no faulty processes, the correct processes cannot be influenced by the faulty processes into making a wrong step. Recall that in Chapter 3, we used pattern-based predicate abstraction and verification conditions to abstract away from the parameter t in the algorithm loop bounds and to ensure that a clean round occurs. In the bounded model checking approach, we make use of the existence of clean rounds by employing the following two-step methodology for the verification of safety properties:

1. find all reachable clean-round configurations,
2. check if a bad configuration is reachable from those configurations.

We also distinguish the following specializations of the clean round condition, which we encounter in our benchmarks:

- the k-set agreement algorithms run for $\lfloor t/k \rfloor + 1$ rounds (e.g., the pseudocode for FloodMin, presented in Figure 2.1 on page 40), in which at most $k-1$ processes fail,

- the algorithms whose executions are organized in phases, consisting of multiple rounds assume a *clean phase* condition, stating that there are no faults in either of the rounds constituting a phase,
- the algorithms that have a dedicated process acting as a coordinator assume that in a clean round, the coordinator is non-faulty.

In particular, the Byzantine consensus algorithms PhaseKing and PhaseQueen, as well as their variants presented in Section 1.6, have their executions organized in phases, and have a dedicated process acting as a coordinator. Thus, these algorithms assume that the coordinator is non-faulty in a clean phase.

In the following, we abuse the notation, and assume that a clean round condition refers either to the general condition, or any of the specializations described above.

Checking Safety by Bounded Reachability. Let $\phi \equiv \mathsf{init} \rightarrow \mathbf{G}(\mathsf{global})$ be the safety property we are interested in verifying. For algorithms that assume a clean round, we check for violations of the safety property ϕ in executions of length $\mathsf{e} \leq 2\mathsf{d}$, where $\mathsf{e} = \mathsf{e}_1 + \mathsf{e}_2$ such that:

1. we find all reachable clean-round configurations in an execution of length e_1, for $\mathsf{e}_1 \leq \mathsf{d}$, such that the last configuration σ_{e_1} satisfies the clean round condition,
2. we check if a bad configuration is reachable from σ_{e_1} by a path of length $\mathsf{e}_2 \leq \mathsf{d}$.

That is, we check satisfiability of the formula:

$$\exists \mathbf{p} \in \mathbf{P}_{RC}.\, \exists \sigma_0, \ldots, \sigma_{\mathsf{e}}.\, \exists tr_1, \ldots, tr_{\mathsf{e}}.\\ \mathit{Init}(\sigma_0) \wedge \mathit{Path}(\sigma_0, \sigma_{\mathsf{e}_1}, \mathsf{e}_1) \wedge \mathit{Clean}(\sigma_{\mathsf{e}_1}) \wedge \mathit{Path}(\sigma_{\mathsf{e}_1}, \sigma_{\mathsf{e}}, \mathsf{e}_2) \wedge \mathit{Bad}(\sigma_{\mathsf{e}}) \quad (5.11)$$

where

- $\mathit{Init}(\sigma_0)$ encodes that σ_0 is an initial configuration that satisfies init,
- $\mathit{Path}(\sigma_0, \sigma_{\mathsf{e}_1}, \mathsf{e}_1)$ and $\mathit{Path}(\sigma_0, \sigma_{\mathsf{e}_2}, \mathsf{e}_2)$ encode paths of length e_1 and e_2, respectively, as defined in (5.7),
- $\mathit{Clean}(\sigma_{\mathsf{e}_1})$ encodes that the configuration σ_{e_1} satisfies the clean round condition,
- $\mathit{Bad}(\sigma_{\mathsf{e}})$ encodes that the configuration σ_{e} satisfies $\neg$global.

Example 5.13. Recall Example 5.1, where we encoded the safety properties of the algorithm FloodMin. In our bounded reachability approach, to verify the safety property *Agreement* we check for violations of *Agreement* in executions of length $\mathsf{e} \leq 2\mathsf{d}$. For $k = 1$, FloodMin runs for $t + 1$ rounds, among which at least one is *clean*, with $k - 1 = 0$

faulty processes. We enforce the clean round condition by asserting that the number of processes in locations $\mathrm{V0}_{\mathsf{cr}}, \mathrm{V1}_{\mathsf{cr}}$ are $k-1=0$ in the configuration σ_{e_1}. *Agreement* is violated if in the last configuration both the locations V0 and V1 are populated, that is, if their counters in the configuration σ_{e} are non-zero. That is, to check *Agreement* for FloodMin, for $k=1$, in formula (5.11) we set:

$$\begin{aligned} \mathit{Init}(\sigma_0) &\equiv \sigma_0.\boldsymbol{\kappa}[\mathrm{V0}] + \sigma_0.\boldsymbol{\kappa}[\mathrm{V1}] + \sigma_0.\boldsymbol{\kappa}[\mathrm{V0}_{\mathsf{cr}}] + \sigma_0.\boldsymbol{\kappa}[\mathrm{V1}_{\mathsf{cr}}] = N(\mathbf{p}) \\ \mathit{Clean}(\sigma_{\mathsf{e}_1}) &\equiv \sigma_{\mathsf{e}_1}.\boldsymbol{\kappa}[\mathrm{V0}_{\mathsf{cr}}] + \sigma_{\mathsf{e}_1}.\boldsymbol{\kappa}[\mathrm{V1}_{\mathsf{cr}}] = 0 \\ \mathit{Bad}(\sigma_{\mathsf{e}}) &\equiv \sigma_{\mathsf{e}}[\mathrm{V0}] > 0 \wedge \sigma_{\mathsf{e}}[\mathrm{V1}] > 0 \end{aligned}$$ □

5.5.2 Algorithms that do not Assume a Clean Round

The synchronous authenticated broadcast algorithm SAB, as well as its variants OmitSAB and HybridSAB, tolerating send omission and hybrid faults, respectively, fall into the class of algorithms that do not have a clean round assumption. Moreover, their STA are monotonic and 1-cyclic, which implies that Theorem 5.3 is applicable. Hence, for these three algorithms, we have a theoretical guarantee that a bound on the diameter exists.

Checking Safety by Bounded Reachability. For the algorithms that do not assume a clean round, we search for violations of the safety property $\phi \equiv \mathsf{init} \rightarrow \mathbf{G}\,(\mathsf{global})$ in executions of length $\mathsf{e} \leq \mathsf{d}$, by checking satisfiability of the formula:

$$\exists \mathbf{p} \in \mathbf{P}_{RC}.\, \exists \sigma_0, \ldots, \sigma_{\mathsf{e}}.\, \exists tr_1, \ldots, tr_{\mathsf{e}}.\; \mathit{Init}(\sigma_0) \wedge \mathit{Path}(\sigma_0, \sigma_{\mathsf{e}}, \mathsf{e}) \wedge \mathit{Bad}(\sigma_{\mathsf{e}}) \tag{5.12}$$

where $\mathit{Init}(\sigma)$, $\mathit{Path}(\sigma_0, \sigma_{\mathsf{e}}, \mathsf{e})$, and $\mathit{Bad}(\sigma)$ are defined as above.

Example 5.14. Again, recall Example 5.1, where we encoded the safety property *Unforgeability* of the algorithm SAB. In our bounded reachability approach, we check executions of length $\mathsf{e} \leq \mathsf{d}$, whose initial configuration has no processes in the location V1, and whose final configuration has more than zero processes in the location AC. Thus, to find violations of *Unforgeability*, in formula (5.12), we set:

$$\begin{aligned} \mathit{Init}(\sigma_0) &\equiv \sigma_0.\boldsymbol{\kappa}[\mathrm{V0}] + \sigma_0.\boldsymbol{\kappa}[\mathrm{V1}] = N(\mathbf{p}) \wedge \sigma_0.\boldsymbol{\kappa}[\mathrm{V1}] = 0 \\ \mathit{Bad}(\sigma_{\mathsf{e}}) &\equiv \sigma_{\mathsf{e}}.\boldsymbol{\kappa}[\mathrm{AC}] > 0 \end{aligned}$$ □

5.6 Experimental Evaluation

Using synchronous threshold automata, presented in Chapter 4, we were able to model algorithms which are designed to tolerate different kinds of faults: crashes, send omissions, Byzantine, or hybrid faults. More precisely, we were able to encode all consensus benchmarks (except EDAC and ESC, which require to store information about messages in the current and previous round that is currently not expressible in synchronous threshold automata), k-set agreement benchmarks, and authenticated broadcast benchmarks, presented in Section 1.6. The encodings of the benchmarks as synchronous threshold

automata, together with the implementations of the procedures for finding the diameter and applying bounded model checking are available at [Stoa]. The experiments were run on a machine with 2,8 GHz Quad-Core Intel(R) Core(TM) i7 CPU and 16GB of RAM.

Computing the Diameter. We implemented the procedure from Section 5.3.1 and used Z3 [dMB08] v4.8.7 and CVC4 [CTTV04] v1.7 as back-end SMT solvers. Our implementation was able to compute diameter bounds even for the benchmarks for which we do not have a theoretical guarantee on the existence of a bound. Our experiments reveal extremely low values for the diameter, that range between 1 and 8. The values for the diameter and the time needed to compute them are presented in Table 5.1. The timeout, denoted by t.o. in the table, was set to 24 hours.

Computing the diameter using this configuration timed out for three benchmarks, kSetOmit, for $k = 2$, HybridKing, and HybridQueen. We ran the diameter procedure for these benchmarks on a more powerful machine. On this machine, we were able to obtain the diameter $\mathsf{d} = 8$ for HybridKing with Z3 in 4 days, and were able to run the bounded model checking on the standard configuration. However, for HybridQueen and kSetOmit, for $k = 2$, we were not able to obtain an answer from either solver for the negation of (5.6) within one week on the machine with more computing power. Hence, at this point, we cannot conclude whether a bound on the diameter for the algorithms HybridQueen and kSetOmit exists.

Checking the Algorithms. We have implemented the procedure which encodes violations of the safety properties as reachability questions on executions of bounded length, as described in Section 5.5, and uses a back-end SMT solver to check their satisfiability. Table 5.1 contains the results that we obtained by checking reachability for our benchmarks, using the diameter bound computed using the procedure from Section 5.3.1. Table 5.2 contains the results we obtained by checking reachability using the diameter from Theorem 5.3, for benchmarks whose STA are monotonic and 1-cyclic.

To our knowledge, we are the first to verify the listed algorithms that tolerate send omission, Byzantine, and hybrid faults. For the algorithms with crash faults, which overlap with those that we checked using the abstraction-based approach from Chapter 3 (for which we presented parameterized model checking results in Table 3.2 on page 100), we observe that the bounded reachability approach is a significant improvement.

Counterexamples. Our implementation of bounded reachability found a bug in the version of the algorithm PhaseKing that was given in [BGP], which was corrected in the version of the algorithm in [BGP89]. The version from [BGP] had the wrong threshold '$> n-t$' in one guard, while the one in [BGP89] had the correct threshold '$\geq n-t$' for the same guard. Although this fix was not due to the result produced by our tool, we believe that it is noteworthy, as it shows that our tool can quickly produce counterexamples. This motivated us to test our tool further, and apply it to erroneous encodings for some of our benchmarks, which we produced. For SAB, HybridSAB, OmitSAB, PhaseKing,

Table 5.1: Results for our benchmarks, available at [Stoa]: $|\mathcal{L}|, |\mathcal{R}|, |\mathrm{CP}_{\mathcal{R}}|$, are the number of locations, rules, and c-propositions in each STA, and d is the diameter computed using SMT. We report on the time it took the solvers Z3 and CVC4 to (i) compute the diameter using SMT, and (ii) check reachability using the diameter computed using the SMT procedure from Section 5.3.1.

algorithm	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert\mathrm{CP}_{\mathcal{R}}\rvert$	d	(i) d time		(ii) BMC time	
					Z3	CVC4	Z3	CVC4
SAB	4	8	4	2	0.07s	0.27s	0.02s	0.03s
HybridSAB	8	16	4	2	0.09s	0.67s	0.03s	0.05s
OmitSAB	8	16	4	2	0.09s	0.67s	0.02s	0.04s
FairCons	11	20	2	2	0.14s	2.68s	0.06s	0.14s
FloodMin, $k = 1$	5	9	2	2	0.06s	0.25s	0.06s	0.09s
FloodMin, $k = 2$	7	16	4	2	0.15s	2.22s	0.06s	0.17s
FloodMinOmit, $k = 1$	4	6	2	1	0.06s	0.04s	0.01s	0.01s
FloodMinOmit, $k = 2$	6	12	4	1	0.05s	0.08s	0.01s	0.03s
FloodSet	7	14	4	2	0.10s	0.90s	0.06s	0.15s
kSetOmit, $k = 1$	8	24	4	3	1.09s	1min8s	0.23s	0.81s
kSetOmit, $k = 2$	12	54	6	–	t.o.	t.o.	–	–
PhaseKing	34	72	12	4	3.67s	15.80s	0.25s	1.47s
ByzKing	34	72	12	4	3.73s	38.50s	0.24s	2.26s
HybridKing	60	138	13	8	t.o.	t.o.	28.82s	5min33s
OmitKing	52	128	13	8	1h15min	t.o.	9.08s	1min27s
PhaseQueen	24	42	8	3	0.40s	4.72s	0.06s	0.50s
ByzQueen	24	42	8	3	0.53s	10.6s	0.08s	0.61s
HybridQueen	42	80	9	–	t.o.	t.o.	–	–
OmitQueen	36	72	9	3	0.57s	8.87s	0.27s	1.21s

and PhaseQueen, we tweaked the resilience condition, and introduced more faults than expected by the algorithm, e.g., by setting $n \geq 3t$ instead of $n > 3t$ for Byzantine faults. For FairCons, FloodMin, FloodMinOmit, and FloodSet, we checked executions without a clean round condition. For all of the erroneous encodings, our tool produces counterexamples in seconds, as can be seen in Table 5.3.

SMT Solvers. In our evaluation, we used Z3 and CVC4 as back-end SMT solvers. To obtain the results presented in the tables above, we ran both solvers with their default configurations. We observed that on our benchmarks, Z3 generally performs better than CVC4.

When computing the diameter, as an input to the SMT solvers, we give an SMT-LIB file that encodes the negation of the diameter query (5.6), similar to the one we constructed in the examples in Section 5.3.2.

Table 5.2: Results for our benchmarks where Theorem 5.3 is applicable: chain and $|\mathrm{CP}_{\mathcal{R}}|$ are the longest chain and the number of c-propositions in each STA, respectively. We report on the time it took both solvers to check reachability using the bound obtained by Theorem 5.3.

algorithm	chain	$\lvert\mathrm{CP}_{\mathcal{R}}\rvert$	BMC time with $\lvert\mathrm{CP}_{\mathcal{R}}\rvert \cdot$ chain	
			Z3	CVC4
SAB	2	4	0.13s	0.26s
HybridSAB	2	4	0.16s	0.42s
OmitSAB	2	4	0.15s	0.45s

Table 5.3: Results for the erroneous encodings our benchmarks, available at [Stoa]. We injected two types of errors: wrong resilience condition (e.g., $n \geq 3t$) and no presence of clean round (denoted by ¬clean in the table). We report on the time it took the solver Z3 (we omit the results with CVC4 for space reasons) to (i) compute the diameter using SMT, and (ii) check reachability using the diameter computed using the SMT procedure from Section 5.3.1.

algorithm	error	d	(i) Z3 d time	(ii) Z3 BMC time	violation
SAB	$n \geq 3t$	2	0.08s	0.03s	unforgeability
OmitSAB	$n \geq 2t$	2	0.03s	0.03s	unforgeability
FairCons	¬clean	2	0.15s	0.03s	agreement
FloodMin, $k = 1$	¬clean	2	0.07s	0.03s	agreement
FloodMin, $k = 2$	¬clean	2	0.15s	0.03s	k-agreement
FloodSet	¬clean	2	0.15s	0.03s	agreement
FloodMinOmit, $k = 1$	¬clean	1	0.03s	0.01s	agreement
FloodMinOmit, $k = 2$	¬clean	1	0.05s	0.01s	k-agreement
PhaseKing	$n \geq 3t$	8	12.85s	2.01s	agreement
PhaseQueen	$n \geq 4t$	6	1.4s	0.55s	agreement

The size of the SMT-LIB files is proportional to the number of locations, rules, guards, and the diameter. For example, the SMT-LIB file for the simplest benchmark, FloodMinOmit, for $k = 1$ and d $= 1$, has 184 lines of code, while the most complicated benchmark, HybridKing for d $= 8$ has 11691 lines of code.

We tried to understand how the SMT solvers deal with the negation of (5.6). With CVC4, we tried disabling the option `cbqi`, which stands for counterexample-based quantifier instantiation [RKK17], and obtained `unknown` for all of our benchmarks. With Z3, we tried disabling the options `mbqi` and `ematching`, which stand for model-based quantifier instantiation and E-matching, respectively, but did not get an output `unknown`, which points out that Z3 does not use these techniques to deal with the quantified query. By

enabling verbose output while running Z3, the solver reports that it uses the `qsat` procedure, which performs quantifier projection [BJ15].

5.7 Discussion and Related Work

In this chapter, we showed a reduction from the parameterized model checking problem for safety properties of synchronous fault-tolerant distributed algorithms, which can be encoded using synchronous threshold automata, to the parameterized bounded reachability problem. We showed undecidability of the parameterized (unbounded) reachability problem by reduction from the halting problem of two-counter machines [Min67]. We presented a semi-decision procedure for computing a bound on the diameter, which can be used as completeness threshold for parameterized bounded reachability. We also showed that for a specific class of synchronous threshold automata, we are able to establish a theoretical bound on the diameter, similar to the asynchronous case [KVW17, KVW15].

Although the majority of our benchmarks have synchronous threshold automata that are neither monotonic nor 1-cyclic, our SMT-based procedure can still automatically compute a diameter bound for most of them. Finding other classes of synchronous threshold automata for which one could derive the diameter bounds is a subject of future work.

It is remarkable that our semi-decision procedure computes finite bounds both for algorithms that run for parameterized number of rounds (such as, e.g., FloodMin, which runs for $\lfloor t/k \rfloor + 1$ rounds), as well as for algorithms that run for infinitely many rounds (such as, e.g., SAB, whose pseudocode, presented in Figure 4.1, contains an infinite loop). Moreover, these finite bounds are constants, and are independent of the parameters.

For the algorithms that assume that every execution has a clean round, our method requires that the verification engineer encodes the clean round condition as input. This is similar to the pattern-based verification conditions that we introduced in Chapter 3. One direction for future work is to use derive the clean round condition automatically from the environment, by analyzing the different fault models and environment assumptions.

The 1-cyclicity condition is reminiscent of flat counter automata [LS05]. In Figure 5.2, we show a possible translation of a synchronous threshold automaton to a counter automaton (similar to the translation for asynchronous threshold automata from [KKW18]). We note that the counter automaton is not flat, due to the presence of the outer loop, which models a transition to the next round. By knowing a bound d on the diameter (e.g., by Theorem 5.3), one can flatten the counter automaton by unfolding the outer loop d times. We also experimented with the tool FAST [BLP06] on two of our benchmarks: SAB and FloodMin for $k = 1$, depicted in Figures 4.1 and 4.2 respectively. FAST terminated on SAB, but took significantly longer than our tool on the same machine (i.e., hours rather than seconds). FAST ran out of memory when checking FloodMin.

In this chapter, we considered a class of safety properties, which happened to be challenging. In [ACJT96], safety of finite-state transition systems over infinite data domains was reduced to backwards reachability checking using a fixpoint computation, as long as the

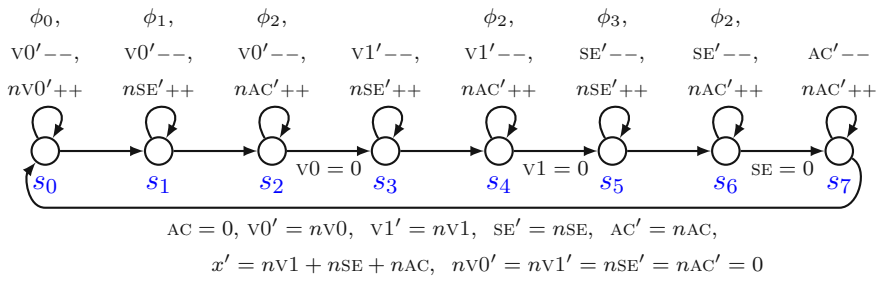


Figure 5.2: A counter automaton for the STA in Figure 4.1, with $\varphi_0 \equiv x < t + 1$, $\varphi_1 \equiv x + f \geq t + 1$, $\varphi_2 \equiv x + f \geq n - t$, $\varphi \equiv x < n - t$, where x counts the number of processes in locations $\text{V1}, \text{SE}, \text{AC}$; and n, t, f are counters for the parameters. On a path from s_0 to s_7, the counters $\ell \in \{\text{V0}, \text{V1}, \text{SE}, \text{AC}\}$ are emptied, while the counters $n\ell$ are populated. This models the transitions from one location to another in the current round.

transition systems are well-structured. It would be interesting to put our results in this context. A decidability result for liveness properties of parameterized timed networks was obtained in [ARZS15], employing linear programming for the analysis of vector addition systems with a parametric initial state. We suggest to investigate the use of similar ideas for analyzing liveness properties of synchronous threshold automata, as well as completeness thresholds for general safety and liveness in the future.

CHAPTER 6

Synchronous Threshold Automata with Receive Message Counters

As discussed throughout this thesis, in the synchronous computation model, the actions that a process takes locally depend on the messages that the process has received by the other processes in the system. Often, a process checks whether a quorum has been obtained (e.g., majority, two-thirds, etc.) by counting the number of messages it has received. Obtaining a quorum means that the number of *received* messages has to pass a given threshold, which should guarantee that it is safe for a correct process to take an action, and move to a new local state. In Chapter 4, we introduced synchronous threshold automata, that allowed us to model synchronous fault-tolerant distributed algorithms based on the observation that from the viewpoint of enabled transitions in a transition system, we may substitute the check whether a quorum of messages has been *received* with a check whether enough messages have been *sent*. For many algorithms, this translation, including the proper modeling of the faulty process behavior, is straightforward and can easily be done manually.

However, other algorithms have more complicated guard expressions over the number of received messages than the guard expressions we have seen so far in the algorithms FloodMin or SAB. Consider, for example, the pseudocode of the algorithm PhaseQueen, presented in Figure 6.1. The algorithm operates in phases, with two rounds per phase. In the first round, all processes broadcast their value stored in the variable `v` (line 4), receive messages from the other processes (line 5), and count how many messages with value 0 (line 6) and value 1 (line 7) were received in the round. If a process received more than $2t$ messages with value 1, then it sets its value to 1 (line 9), otherwise it sets its value to 0 (line 11). In the second round, a specific process i is chosen to be a queen if the number of the current phase is equal to i (line 13). In this round, only the queen broadcasts a message (line 14). Each process receives the queen's value $\mathtt{v}_q$ (line 15), and checks if in the first round, it received less than $n - t$ messages with value equal to their

```
1  v := input({0, 1})
2  for each phase 1 through t + 1 do {
3    /* round 1: full message exchange */
4    broadcast v
5    receive messages from other processes
6    C[0] := number of received 0's
7    C[1] := number of received 1's
8    if C[1] > 2t then
9      v := 1
10   else
11     v := 0
12   /* round 2: queen's broadcast */
13   if phase = i then
14     broadcast v
15   receive queen's message v_q
16   if C[v] < n − t then
17     v := v_q
18 }
```

Figure 6.1: The pseudocode of PhaseQueen, code for process i

own value v. If this is the case, the process sets its value to the value v_q received from the queen (line 17).

Other algorithms that have similar, more complicated guard expressions include PhaseKing, as well as the variants of PhaseQueen and PhaseKing for different fault models, including hybrid faults, where it is assumed that processes may fail according to different fault assumptions. The interplay of the different fault models is a challenge for doing manual abstractions. For example, in the hybrid faults variants HybridKing and HybridQueen of these two algorithms that we considered, the messages that processes receive are either from correct, send-omission faulty, or Byzantine faulty processes. Hence, to get the correct guards over the number of sent messages (which are either sent by correct or send-omission-faulty processes), one needs to take into account the messages sent by Byzantine-faulty processes as well (which are modeled implicitly in the synchronous threshold automata framework, see Section 4.1.2).

In this chapter, we introduce a *new variant* of synchronous threshold automata that allows expressing guards over the receive variables, and thus is a formalization which captures the constructs that appear in the pseudocode found in the literature. We further propose an explicit encoding of the implicit assumptions imposed by the computation and fault models, by adding constraints to the environment assumption Env, which are specific to the respective fault model and can be reused for different algorithms.

6.1 Process and Environment Specification: Receive Synchronous Threshold Automaton

We generalize synchronous threshold automata, defined in Chapter 4, to contain receive variables and define guards over these receive variables.

Definition 6.1 (Receive synchronous threshold automaton)**.** A *receive synchronous threshold automaton*, or *receive* STA, is the tuple $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$, where:

- $\mathcal{L}$ is a finite set of *locations*,
- $\mathcal{I} \subseteq \mathcal{L}$ is a non-empty set of *initial locations*,
- $\mathcal{R}^{\Delta}$ is a finite set of *rules*,
- Δ is the finite set of *receive variables*,
- Π is a finite set of *parameters*,
- RC is a *resilience condition*,
- Env^{Δ} is an *environment assumption*. □

The locations $\mathcal{L}$, initial locations $\mathcal{I}$, rules $\mathcal{R}^{\Delta}$, and receive variables Δ constitute the process specification. The environment specification is defined by the environment assumption Env^{Δ}. The parameters and resilience condition are defined as for synchronous threshold automata, presented in Chapter 4. We discuss the remaining components of the receive STA in detail below.

6.1.1 Process Specification: Locations, Rules and Receive Variables

The locations, initial locations, and sending of messages in a receive STA are defined as for STA, discussed in Chapter 4. We now introduce the receive variables Δ, and the rules $\mathcal{R}^{\Delta}$, which contain expressions over receive variables.

Definition 6.2 (Receive variables)**.** The set Δ contains *receive variables* $\mathsf{nr}(m)$ that range over $\mathbb{N}$, where $m \in \mathcal{M}$. □

A receive variable $\mathsf{nr}(m)$ stores the number of messages of type $m \in \mathcal{M}$ that were received by a process. Thus, $|\Delta| = |\mathcal{M}|$, as in Δ there is exactly one receive variable $\mathsf{nr}(m)$ per message type $m \in \mathcal{M}$. Initially, each receive variable is assigned the value 0. As they are used to count the number of received messages in a round, their value depends on the number of messages sent in the given round, as we will discuss in Section 6.1.2.

Let $M \subseteq \mathcal{M}$ denote a set of message types, and let $\#M$ denote the total number of messages of types $m \in M$, received by some process. Observe that the notation $\#M$ is a

shorthand for $\sum_{m \in M} \mathsf{nr}(m)$. We will use these two notations interchangeably. Further, when M is a singleton set, that is, when $M = \{m\}$, we will simply use the notation $\mathsf{nr}(m)$ to denote $\#\{m\}$. For the purpose of expressing guards over the receive variables $\mathsf{nr}(m)$, for $m \in \mathcal{M}$, we define *r-propositions*.

Definition 6.3 (Receive propositions)**.** We define *r-propositions*, which are expressions of the form:

$$\#M \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b, \text{ such that } M \subseteq \mathcal{M}, \boldsymbol{a} \in \mathbb{Z}^{|\Pi|}, b \in \mathbb{Z}$$

We denote by RP the set of r-propositions. □

The intended meaning of the r-propositions is to check whether the total number of messages of types $m \in M$ received by some process i is greater than or equal to a linear combination of the parameters. Formally, they are evaluated in tuples $(\mathbf{d}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$ is a vector of values assigned to each receive variable $\mathsf{nr}(m)$, for $m \in \mathcal{M}$, and $\mathbf{p} \in \mathbf{P}_{RC}$.

Definition 6.4 (Semantics of r-propositions)**.** Given a tuple $(\mathbf{d}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$ is a vector of values assigned to each receive variable $\mathsf{nr}(m)$, for $m \in \mathcal{M}$, and $\mathbf{p} \in \mathbf{P}_{RC}$, we define:

$$(\mathbf{d}, \mathbf{p}) \models \#M \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{iff} \quad \sum_{m \in M} \mathbf{d}[m] \geq \boldsymbol{a} \cdot \mathbf{p} + b$$ □

Similarly to the way we defined rules of STA in Chapter 4, the rules $r^{\Delta} \in \mathcal{R}^{\Delta}$ in rSTA are tuples $r^{\Delta} = (\mathit{from}, \mathit{to}, \varphi)$, where $r^{\Delta}.\mathit{from}, r^{\Delta}.\mathit{to} \in \mathcal{L}$ are locations, and $r^{\Delta}.\varphi$ is a *receive guard*, defined below.

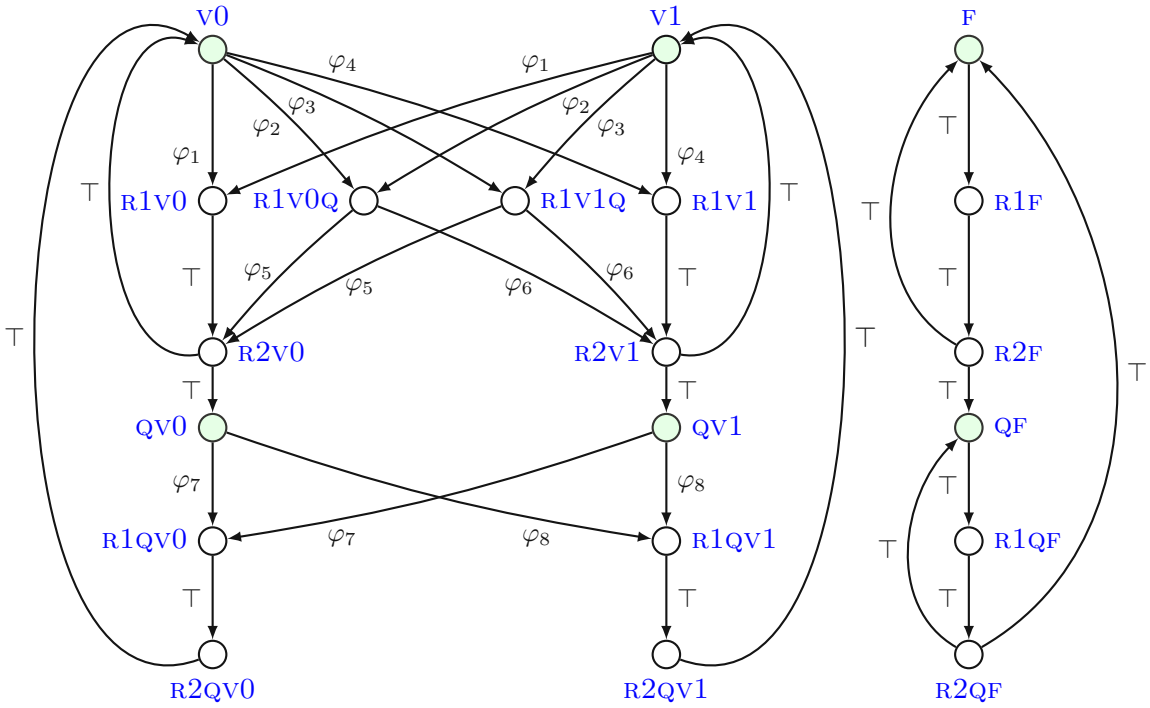
Definition 6.5 (Receive guard)**.** A *receive guard* is a Boolean combination of c-propositions and r-propositions. □

We now define the formal semantics of the receive guards $r^{\Delta}.\varphi$, for $r^{\Delta} \in \mathcal{R}^{\Delta}$, which are evaluated in tuples $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$.

Definition 6.6 (Semantics of receive guards)**.** Given a tuple $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$ is a vector of valuations of the receive variables $\mathsf{nr}(m)$, for $m \in \mathcal{M}$, $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of counters, and $\mathbf{p} \in \mathbf{P}_{RC}$ is an admissible valuation of the parameter vector $\boldsymbol{\pi}$, we evaluate c-propositions and r-propositions as follows:

$$(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{iff} \quad (\boldsymbol{\kappa}, \mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{(cf. Definition 4.8)}$$
$$(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) \models \#M \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{iff} \quad (\mathbf{d}, \mathbf{p}) \models \#M \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{(cf. Definition 6.4)}$$

The semantics of the Boolean connectives is standard. □



$$\mathcal{L} = \{\text{V0}, \ldots, \text{R2QF}\} \qquad \mathcal{I} = \{\text{V0}, \text{V1}, \text{QV0}, \text{QV1}, \text{F}, \text{QF}\}$$
$$\mathcal{R}^{\Delta} = \{r_1^{\Delta}, \ldots, r_{34}^{\Delta}\} \qquad \mathcal{M} = \{m_0, m_1, m_{q0}, m_{q1}\}$$
$$\varphi_1 \equiv \mathsf{nr}(m_1) \leq 2t \wedge \mathsf{nr}(m_0) \geq n - t \qquad \varphi_5 \equiv \mathsf{nr}(m_{q0}) \geq 1$$
$$\varphi_2 \equiv \mathsf{nr}(m_1) \leq 2t \wedge \mathsf{nr}(m_0) < n - t \qquad \varphi_6 \equiv \mathsf{nr}(m_{q1}) \geq 1$$
$$\varphi_3 \equiv \mathsf{nr}(m_1) > 2t \wedge \mathsf{nr}(m_1) \geq n - t \qquad \varphi_7 \equiv \mathsf{nr}(m_1) \leq 2t$$
$$\varphi_4 \equiv \mathsf{nr}(m_1) > 2t \wedge \mathsf{nr}(m_1) < n - t \qquad \varphi_8 \equiv \mathsf{nr}(m_1) > 2t$$
$$\Pi = \{n, t, f\} \qquad RC \equiv n > 4t \wedge t \geq f$$
$$Q = \{\text{QV0}, \ldots, \text{R2QV1}, \text{QF}, \text{R1QF}, \text{R2QF}\} \quad \text{(queen locations)}$$
$$F = \{\text{F}, \text{R1F}, \text{R2F}, \text{QF}, \text{R1QF}, \text{R2QF}\} \quad \text{(faulty locations)}$$
$$\mathsf{sent}(m_0) = \{\text{V0}, \text{QV0}\} \qquad \mathsf{sent}(m_1) = \{\text{V1}, \text{QV1}\}$$
$$\mathsf{sent}(m_{q0}) = \{\text{R1QV0}\} \qquad \mathsf{sent}(m_{q1}) = \{\text{R1QV1}\}$$

Figure 6.2: The rSTA for the algorithm PhaseQueen. We omit the rule names, and label the rules by their guards. The definition of Env^{Δ} will follow in Section 6.1.2.

Example 6.1. Figure 6.2 shows the rSTA of the algorithm PhaseQueen, whose pseudocode is given in Figure 6.1.

To model the behavior of the processes running this algorithm using a receive STA, we proceed as follows. First, we have locations that encode the behavior of the correct processes which are not a queen in the current phase:

- V0 and V1, which encode that a process has the value 0 and 1, respectively,
- R1V0 and R1V1, which encode that after the end of the first round, a process sets its value to 0 and 1, respectively, and that it has received at least $n - t$ messages that have its value in the first round (that is, the condition from line 16 evaluates to false),
- R1V0Q and R1V1Q, which encode that after the end of the first round, a process sets its value to 0 and 1, respectively, and that it has received less than $n - t$ messages that have its value in the first round, and will use the message received from the queen to update its value at the end of the second round (that is, the condition from line 16 evaluates to true),
- R2V0 and R2V1, which encode that after the end of the second round, a process sets its value to 0 and 1, respectively.

From the locations R2V0 and R2V1, we have outgoing rules that bring the process back to the beginning of the next phase, that is, to the locations V0 and V1, respectively. Additionally, a process might move from the locations R2V0 and R2V1 to QV0 and QV1, respectively, and thus become a queen in the next phase. The locations QV0, QV1, R1QV0, R1QV1, R2QV0, R2QV1 capture the behavior of a correct process acting as a queen in the current phase.

In the case of the algorithm PhaseQueen, it can happen that in some phase, a faulty process acts as a queen. To capture this behavior, we introduce locations which are populated by a single Byzantine-faulty process, namely the locations $F = \{\text{F}, \ldots, \text{R2QF}\}$. The Byzantine faulty process may act as a queen in some phase, if it moves from the location R2F to the location R2QF.

Processes in locations V0, QV0 send messages of type m_0, that is, messages containing the value 0. Similarly, processes in locations V1, QV1 send messages of type m_1, containing the value 1. The message types m_{q0} and m_{q1} are used to encode that the queen in the current phase sent a message with value 0 and 1, respectively. When a Byzantine-faulty process is in location R1QF, which encodes that it is a queen and that it performs the queen broadcast, it can send a message of either type m_{q0} or m_{q1}, that is, a message containing either value 0 or value 1.

The receive guards $\varphi_1, \ldots, \varphi_8$ express conditions over the number of received messages of a certain message type, and capture conditions which appear in the pseudocode. Observe

that by $\mathsf{nr}(m_1)$, we denote the number of messages containing the value 1 that a process received in the first round of the phase. This corresponds to the number stored in the variable $\mathsf{C}[1]$ in the pseudocode (line 7). Similarly, $\mathsf{nr}(m_{q0})$ denotes the number of messages containing the value 0 received by the queen in the second round of the phase. For example, the receive guard φ_2 is satisfied if a process received:

- not more than $2t$ messages of type m_1, that is, if the process takes the **else** branch in line 10, and
- less than $n - t$ messages of type m_0, that is, if the condition on line 16 evaluates to true.

The rules that move processes to the location R1V0Q are guarded by the receive guard φ_2. This receive guard encodes that a process sets its value to 0 at the end of the first round, and that it will use the queen's message to update its value at the end of the second round.

In the next section, we will introduce the constraints imposed by the environment, and will discuss the environment assumption Env^{Δ} of the rSTA used to encode the algorithm PhaseQueen. □

6.1.2 Environment Specification: Environment Assumption

Modeling faults in rSTA uses the same methodology as for STA, which we defined in Section 4.1.2. In addition to introducing locations and rules to model the behavior of the faulty processes, as well as constraints on the number of processes in given locations, to faithfully model the faulty environment of the receive STA, we will introduce constraints on the values of the receive variables. Generally, these constraints express that the number of received messages is in the range from the number of messages sent by *correct* processes to the total number of sent messages (sent by both correct and faulty processes).

To define the constraints in the environment assumption Env^{Δ}, we introduce e-propositions, which we use to encode the relation between the number of received and sent messages.

Definition 6.7 (Environment propositions)**.** We define *e-propositions*, which are expressions of the form:

$$\#M \geq \#L + \boldsymbol{a} \cdot \boldsymbol{\pi} + b, \text{ such that } M \subseteq \mathcal{M}, L \subseteq \mathcal{L}, \boldsymbol{a} \in \mathbb{Z}^{|\Pi|}, b \in \mathbb{Z}$$

We denote by EP the set of e-propositions. □

The e-propositions are evaluated in tuples $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$, formally defined below.

Definition 6.8 (Semantics of e-propositions)**.** Given a tuple $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$ is a vector of valuations of the receive variables $\mathsf{nr}(m)$, for $m \in \mathcal{M}$, $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of counters, and $\mathbf{p} \in \mathbf{P}_{RC}$ is an admissible valuation of the parameter

vector $\boldsymbol{\pi}$, we evaluate e-propositions as follows:

$$(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) \models \#M \geq \#L + \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{iff} \quad \sum_{m \in M} \mathbf{d}[m] \geq \sum_{\ell \in L} \boldsymbol{\kappa}[\ell] + \boldsymbol{a} \cdot \mathbf{p} + b \qquad \square$$

Definition 6.9 (Environment assumption Env^{Δ})**.** The environment assumption Env^{Δ} is a conjunction of c-propositions, e-propositions and their negations.

We denote by $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$, and $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$ the sub-formulas of Env^{Δ} consisting only of c-propositions, and e-propositions (and their negations), respectively. $\square$

In the environment assumption Env^{Δ}, the c-propositions restrict the number of processes in certain locations, while the e-propositions restrict the values of the receive variables by relating them to the number of sent messages of the same type.

Recall the mapping sent, defined in Definition 4.5, which assigns to every message type $m \in \mathcal{M}$ the set $\mathsf{sent}(m) \subseteq \mathcal{L}$ of locations, where correct processes send a message of type m. The synchronous computation model imposes a lower bound on the number of received messages. Namely, as every message sent by a correct process in one round is received in the same round, the number $\mathsf{nr}(m)$ of received messages of type $m \in \mathcal{M}$ is bounded from below by the number $\#\mathsf{sent}(m)$ of messages of type m, sent by correct processes. Irrespective of the fault model, the environment assumptions contain the following constraints:

(E1) $\#\mathsf{sent}(m) \leq \mathsf{nr}(m)$, for each $m \in \mathcal{M}$.

By Definition 6.9, we have that:

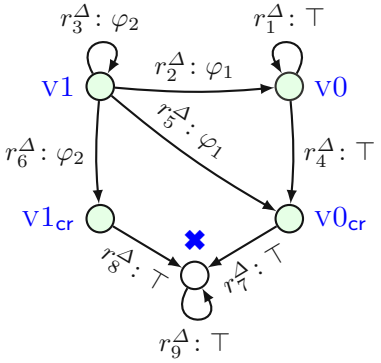
$$\mathsf{Env}^{\Delta} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}} \wedge \mathsf{Env}^{\Delta}_{\mathrm{EP}}$$

where:

$$\mathsf{Env}^{\Delta}_{\mathrm{CP}} \equiv \mathsf{C1} \wedge \mathsf{C2} \wedge \mathsf{Env}_{\mathrm{CP},*} \quad \text{and} \quad \mathsf{Env}^{\Delta}_{\mathrm{EP}} \equiv \mathsf{E1} \wedge \mathsf{Env}_{\mathrm{EP},*} \quad \text{for } * \in \{\mathsf{cr}, \mathsf{so}, \mathsf{byz}\}$$

The subformula $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$, which is a conjunction c-propositions and their negations, contains the constraints that we already defined for the synchronous threshold automata without receive variables in Section 4.1.2. To capture the relationship between the number of sent and received messages, in the following, for each of the fault models we consider in this thesis, we define the constraints $\mathsf{Env}_{\mathrm{CP},*}$, where $* \in \{\mathsf{cr}, \mathsf{so}, \mathsf{byz}\}$.

Crash Faults. For crash faults, the environment constraint $\mathsf{Env}_{\mathrm{EP},\mathsf{cr}}$ is the conjunction of e-propositions and their negations that restricts the values of the receive variables under the crash fault model as follows. First, recall the mapping $\mathsf{sent}_{\mathsf{cr}}$, defined in Section 4.1.2, which defines the set of crash locations where a crashed process sends a message of type $m \in \mathcal{M}$. For every message type $m \in \mathcal{M}$, we have that the number of received



$$
\begin{array}{rcl}
\mathcal{R}^\Delta & = & \{r_1^\Delta, \ldots, r_9^\Delta\} \\
\varphi_1 & \equiv & \mathsf{nr}(m_0) \geq 1 \\
\varphi_2 & \equiv & \mathsf{nr}(m_0) < 1 \\
\mathsf{Env}_{\mathsf{cr}}^\Delta & \equiv & \bigwedge_{\ell \in \mathcal{L}} \#\{\ell\} \geq 0 \\
& & \wedge\ \#\mathcal{L} = n \\
& & \wedge\ \#\{\mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}, ✖\} \leq f \\
& & \wedge\ \#\{\mathrm{v0}\} \leq \mathsf{nr}(m_0) \leq \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \\
& & \wedge\ \#\{\mathrm{v1}\} \leq \mathsf{nr}(m_1) \leq \#\{\mathrm{v1}, \mathrm{v1}_{\mathsf{cr}}\}
\end{array}
$$

Figure 6.3: The receive STA encoding the loop body of the FloodMin algorithm for $k = 1$, whose pseudocode is given in Figure 2.1, and which tolerates crash faults.

messages of type m for each process is a value, which is bounded from above by the number $\#(\mathsf{sent}(m) \cup \mathsf{sent}_{\mathsf{cr}}(m))$ of messages of type m, sent by the correct processes and the processes flagged as crashed in the current round. That is, $\mathsf{Env}_{\mathrm{EP},\mathsf{cr}}$ is the following constraint:

$$\mathsf{Env}_{\mathrm{EP},\mathsf{cr}} \equiv \bigwedge_{m \in \mathcal{M}} \mathsf{nr}(m) \leq \#(\mathsf{sent}(m) \cup \mathsf{sent}_{\mathsf{cr}}(m))$$

Example 6.2. Figure 6.3 depicts the rSTA for the algorithm FloodMin, for $k = 1$, whose pseudocode is presented in Figure 2.1.

Recall Example 4.3 on page 110, where we encoded the algorithm FloodMin, for $k = 1$, using an STA. We identified the sets $\mathcal{L}_{\mathsf{corr}} = \{\mathrm{v0}, \mathrm{v1}\}$ of correct locations, $\mathcal{L}_{\mathsf{cr}} = \{\mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}\}$ of crash locations, $\mathcal{M} = \{m_0, m_1\}$ of message types, the mappings sent and $\mathsf{sent}_{\mathsf{cr}}$, and the environment constraint $\#\{\mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}, ✖\} \leq f$, specific for the crash fault model.

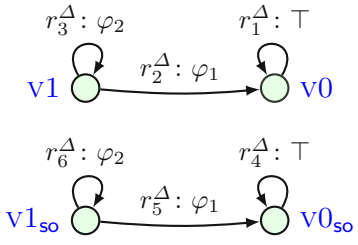
In the rSTA, we have two receive guards:

- $\varphi_1 \equiv \mathsf{nr}(m_0) \geq 1$, which checks if a process received at least one message of type m_0,
- $\varphi_2 \equiv \mathsf{nr}(m_0) < 1$, which checks if a process did not receive messages of type m_0.

The environment assumption $\mathsf{Env}_{\mathsf{cr}}^\Delta$ differs from the environment assumption Env from Figure 4.2 in the constraints that restrict the values of the receive variables $\mathsf{nr}(m)$, for $m \in \mathcal{M}$. In particular, for the algorithm FloodMin, for $k = 1$, we have the following constraints:

$$\#\{\mathrm{v0}\} \leq \mathsf{nr}(m_0) \leq \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \quad \text{and} \quad \#\{\mathrm{v1}\} \leq \mathsf{nr}(m_1) \leq \#\{\mathrm{v1}, \mathrm{v1}_{\mathsf{cr}}\}$$

which restrict the number $\mathsf{nr}(m_0)$ (resp. $\mathsf{nr}(m_1)$) of received messages of type m_0 (resp. m_1) to a value in the range from the number of processes in location v0 (resp. v1) to the number of processes in locations $\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}$ (resp. $\mathrm{v1}, \mathrm{v1}_{\mathsf{cr}}$). □



$$
\begin{aligned}
\mathcal{R}^\Delta &= \{r_1^\Delta, \ldots, r_6^\Delta\} \\
\varphi_1 &\equiv \mathsf{nr}(m_0) \geq 1 \\
\varphi_2 &\equiv \mathsf{nr}(m_0) < 1 \\
\mathsf{Env}_{\text{so}}^\Delta &\equiv \textstyle\bigwedge_{\ell \in \mathcal{L}} \#\{\ell\} \geq 0 \\
&\quad \wedge \#\mathcal{L} = n \\
&\quad \wedge \#\{\text{v0}, \text{v1}\} = n - f \\
&\quad \wedge \#\{\text{v0}_{\text{so}}, \text{v1}_{\text{so}}\} = f \\
&\quad \wedge \#\{\text{v0}\} \leq \mathsf{nr}(m_0) \leq \#\{\text{v0}, \text{v0}_{\text{so}}\} \\
&\quad \wedge \#\{\text{v1}\} \leq \mathsf{nr}(m_1) \leq \#\{\text{v1}, \text{v1}_{\text{so}}\}
\end{aligned}
$$

Figure 6.4: The receive STA encoding the loop body of the FloodMinOmit algorithm for $k = 1$, whose pseudocode is given in Figure 2.1, and which tolerates send omission faults.

Send Omission Faults. The environment constraint $\mathsf{Env}_{\text{EP,so}}$ for the send omission fault model restricts the values of the receive variables under the send omission fault model. Recall the mapping $\mathsf{sent}_{\text{so}}$, defined in Section 4.1.2, which defines the set of send-omission locations where a send-omission-faulty process sends a message of type $m \in \mathcal{M}$. Similarly to the crash fault model, for every message type $m \in \mathcal{M}$, in the environment constraint $\mathsf{Env}_{\text{EP,so}}$ we have that the number of received messages of type m for each process is some value less than or equal to the number $\#(\mathsf{sent}(m) \cup \mathsf{sent}_{\text{so}}(m))$ of messages of type m, sent by the correct and the send-omission-faulty processes:

$$\mathsf{Env}_{\text{EP,so}} \equiv \bigwedge_{m \in \mathcal{M}} \mathsf{nr}(m) \leq \#(\mathsf{sent}(m) \cup \mathsf{sent}_{\text{so}}(m))$$

Example 6.3. Figure 6.4 depicts the rSTA for the algorithm FloodMinOmit, for $k = 1$, whose pseudocode is presented in Figure 2.1.

Recall Example 4.4, where we encoded the algorithm FloodMinOmit, for $k = 1$, using an STA, and identified the sets $\mathcal{L}_{\text{corr}} = \{\text{v0}, \text{v1}\}$ of correct locations, $\mathcal{L}_{\text{so}} = \{\text{v0}_{\text{so}}, \text{v1}_{\text{so}}\}$ of send-omission locations, $\mathcal{M} = \{m_0, m_1\}$ of message types, the mappings sent and $\mathsf{sent}_{\text{so}}$, and the environment constraint $(\#\{\text{v0}, \text{v1}\} = n - f \wedge \#\{\text{v0}_{\text{so}}, \text{v1}_{\text{so}}\} = f)$, specific for the send omission fault model.

In the rSTA for FloodMinOmit, for $k = 1$, we have the receive guards φ_1 and φ_2, which are defined in the same way in the rSTA for the algorithm FloodMin, for $k = 1$, which tolerates crash faults.

While the guards φ_1 and φ_2 are syntactically the same, both for the version that tolerates crash and the version that tolerates send omission faults, the constraints that define the relation between the number of received and sent messages differ. That is, for the algorithm FloodMinOmit, for $k = 1$, we have following constraints:

$$\#\{\text{v0}\} \leq \mathsf{nr}(m_0) \leq \#\{\text{v0}, \text{v0}_{\text{so}}\} \quad \text{and} \quad \#\{\text{v1}\} \leq \mathsf{nr}(m_1) \leq \#\{\text{v1}, \text{v1}_{\text{so}}\}$$

which restrict the number $\mathsf{nr}(m_0)$ (resp. $\mathsf{nr}(m_1)$) of received messages of type m_0 (resp. m_1) to a value between the number of processes in location v0 (resp. v1) and the number of processes in locations $\mathrm{v0}, \mathrm{v0}_{\mathsf{so}}$ (resp. $\mathrm{v1}, \mathrm{v1}_{\mathsf{so}}$). □

Byzantine Faults. As discussed in Section 4.1.2, in the case of Byzantine faults, the threshold automaton encodes the behavior of a correct process only, and the environment captures the effect that the Byzantine-faulty processes have on the correct ones. We did not introduce a mapping that defined the locations where Byzantine-faulty processes sent locations, as we did in the case of crash and send omission faults. Rather, for Byzantine faults, we overapproximated the number of messages sent by Byzantine-faulty processes by the parameter f, which denotes the number of faults.

This means that we do not have additional new locations or rules, and that $\mathsf{Env}_{\mathrm{CP},\mathsf{byz}} \equiv \top$. The effect that the Byzantine-faulty processes have on the correct processes is encoded using the formula $\mathsf{Env}_{\mathrm{EP},\mathsf{byz}}$, which defines constraints which bound the values of the receive variables. These constraints motivated by some typical implicit assumptions on the design of distributed algorithms that we illustrate in the following.

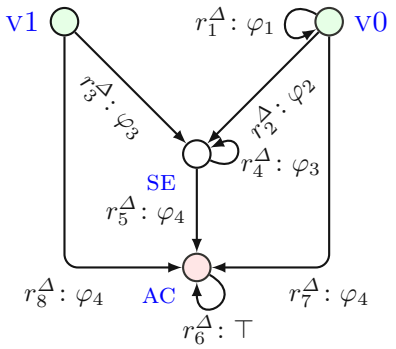
Consider the r-proposition $\#\{m_0, m_1\} \geq n - t$, that checks the total number of received messages of types m_0 and m_1 passes the threshold $n - t$. If a Byzantine-faulty process sends both messages of types m_0 and m_1 to the same correct process, then the correct process can locally detect that the Byzantine-faulty process misbehaved, as each correct process is supposed to send only one of those messages (due to our assumption that a correct process in a given location cannot send messages of more than one type in a round). Thus, the correct process i can safely filter out one or both of those messages, and count at most one of them towards the $n - t$ threshold. In the rSTA, the received messages are stored in receive variables $\mathsf{nr}(m) \in \Delta$, for $m \in \mathcal{M}$, which are counters, and as such, cannot track messages received from individual processes. To capture this (implicit) filtering in our model, we introduce constraints in $\mathsf{Env}_{\mathrm{EP},\mathsf{byz}}$, which state that the number of received messages of both types m_0 and m_1 does not exceed the number of messages of types m_0 and m_1 sent by correct processes, together with the messages possibly sent by the f Byzantine-faulty processes.

In general, the environment constraint $\mathsf{Env}_{\mathrm{EP},\mathsf{byz}}$ for Byzantine faults is:

$$\mathsf{Env}_{\mathrm{EP},\mathsf{byz}} \equiv \bigwedge_{r^\Delta \in \mathcal{R}^\Delta} \#M(r^\Delta.\varphi) \leq \#\Big(\bigcup_{m \in M(r^\Delta.\varphi)} \mathsf{sent}(m) \Big) + f$$

where $M(r^\Delta.\varphi) \subseteq \mathcal{M}$ are the message types of the receive variables that occur in a receive guard $r^\Delta.\varphi$, for $r^\Delta \in \mathcal{R}^\Delta$.

Observe that for every rule $r^\Delta \in \mathcal{R}^\Delta$, the number $\#(\bigcup_{m \in M(r^\Delta.\varphi)} \mathsf{sent}(m)) + f$ denotes the total number of sent messages of types $M(r^\Delta.\varphi)$, since there are exactly $\bigcup_{m \in M(r^\Delta.\varphi)} \mathsf{sent}(m)$ messages of types $M(r^\Delta.\varphi)$ sent by the correct processes, and at



$$
\begin{aligned}
\mathcal{R}^\Delta &= \{r_1^\Delta, \ldots, r_8^\Delta\} \\
\varphi_1 &\equiv \mathsf{nr}(m_{\mathsf{ECHO}}) < t+1 \\
\varphi_2 &\equiv \mathsf{nr}(m_{\mathsf{ECHO}}) \geq t+1 \\
\varphi_3 &\equiv \mathsf{nr}(m_{\mathsf{ECHO}}) < n-t \\
\varphi_4 &\equiv \mathsf{nr}(m_{\mathsf{ECHO}}) \geq n-t \\
\mathsf{Env}^\Delta_{\mathsf{byz}} &\equiv \textstyle\bigwedge_{\ell \in \mathcal{L}} \#\{\ell\} \geq 0 \\
&\quad \wedge \#\mathcal{L} = n - f \\
&\quad \wedge \#\{\text{V1}, \text{SE}, \text{AC}\} \leq \mathsf{nr}(m_{\mathsf{ECHO}}) \\
&\qquad \leq \#\{\text{V1}, \text{SE}, \text{AC}\} + f
\end{aligned}
$$

Figure 6.5: The receive STA encoding the loop body of the algorithm SAB whose pseudocode is given in Figure 4.1, and which tolerates Byzantine faults.

most f messages of types $M(r^\Delta.\varphi)$ sent by the Byzantine-faulty processes. The constraint $\mathsf{Env}_{\mathsf{EP,byz}}$ thus captures the filtering of messages of different types, sent by the same Byzantine-faulty processes.

That is, our example r-proposition $\#\{m_0, m_1\} \geq n - t$ induces the following constraint:

$$\#\{m_0, m_1\} \leq \#(\mathsf{sent}(m_0) \cup \mathsf{sent}(m_1)) + f$$

Example 6.4. Consider the rSTA of the algorithm SAB, presented in Figure 6.5. The pseudocode of the algorithm SAB is given in Figure 4.1.

Recall that the algorithm SAB has a single message type, m_{ECHO}, and that the processes in locations V1, SE, or AC send a message of type m_{ECHO}. In the rSTA, we have four receive guards:

- $\varphi_1 \equiv \mathsf{nr}(m_{\mathsf{ECHO}}) < t + 1$ and $\varphi_3 \equiv \mathsf{nr}(m_{\mathsf{ECHO}}) < n - t$, which check if a process received less than $t + 1$ and $n - t$ messages with value ECHO, respectively,
- $\varphi_2 \equiv \mathsf{nr}(m_{\mathsf{ECHO}}) \geq t + 1$ and $\varphi_4 \equiv \mathsf{nr}(m_{\mathsf{ECHO}}) \geq n - t$, which check if a process received at least $t + 1$ and $n - t$ messages with value ECHO, respectively.

Observe that we have $M(\varphi_i) = \{m_{\mathsf{ECHO}}\}$, for $1 \leq i \leq 4$. The guards precisely encode the conditions in the pseudocode: for example, φ_2 captures the condition on line 7 in Figure 4.1, and φ_1 its negation (that is, the implicit **else** branch).

In the environment assumption Env^Δ, in addition to the environment constraints already present in the STA of the algorithm SAB (see Example 4.5), we have the constraint that restricts the value of the receive variable $\mathsf{nr}(m_{\mathsf{ECHO}})$:

$$\#\{\text{V1}, \text{SE}, \text{AC}\} \leq \mathsf{nr}(m_{\mathsf{ECHO}}) \leq \#\{\text{V1}, \text{SE}, \text{AC}\} + f$$

which states that the number of received messages containing the value ECHO ranges between the number of correct processes that sent a value ECHO, and the total number of (correct and Byzantine-faulty) processes that sent a value ECHO. □

Remark on Algorithms with a Coordinator. In the algorithms that tolerate Byzantine faults, and where no process acts as a coordinator, such as the algorithm SAB, we have that the number of participating processes is equal to the number of correct processes, that is, $N(\boldsymbol{\pi}) = n - f$. When modeling algorithms that tolerate Byzantine faults where a process acts as a coordinator in some round, we need to take into account that in some phase, the coordinator will be Byzantine. Thus, we add locations $\mathcal{L}_{\mathsf{byz}} \subseteq \mathcal{L}$ for a single Byzantine-faulty process, disjoint from the locations that encode the behavior of the correct processes. The newly introduced locations do not encode any values of the local variables, they ensure that the Byzantine process (which may become a coordinator) moves synchronously with the other processes. In the rSTA for the algorithm PhaseQueen (Figure 6.2), we defined $\mathcal{L}_{\mathsf{byz}} = F = \{\text{F}, \ldots, \text{R2QF}\}$. As we model the behavior of a single Byzantine process explicitly, we have $N(\boldsymbol{\pi}) = n - f + 1$ for algorithms with a coordinator that tolerate Byzantine faults.

In this case, we define the constraints $\mathsf{Env}_{\text{CP,co}}$, which restrict the number of processes in given locations. We also identify locations $\mathcal{L}_{\mathsf{co}} \subseteq \mathcal{L}$, which only a (correct or Byzantine) coordinator is allowed to populate. The environment constraint $\mathsf{Env}_{\text{CP,co}}$ for Byzantine-tolerant algorithms with a coordinator is:

$$\mathsf{Env}_{\text{CP,co}} \equiv \#\mathcal{L}_{\mathsf{co}} = 1 \wedge \#\mathcal{L}_{\mathsf{byz}} = 1$$

where $\#\mathcal{L}_{\mathsf{co}} = 1$ (resp. $\#\mathcal{L}_{\mathsf{byz}} = 1$) ensures that there is exactly one process in the coordinator locations $\mathcal{L}_{\mathsf{co}}$ (resp. in the Byzantine locations $\mathcal{L}_{\mathsf{byz}}$).

Additionally, we have message types $m_{\mathsf{co}} \in \mathcal{M}$ that model the coordinator messages, and denote by ℓ_F the location where the Byzantine process performs the coordinator broadcast. The constraint $\mathsf{Env}_{\text{EP,co}}$ states that the number of received coordinator messages of type m_{co} does not exceed the total number of coordinator messages of type m_{co} sent by the correct and Byzantine coordinators:

$$\mathsf{Env}_{\text{EP,co}} \equiv \mathsf{Env}_{\text{EP,byz}} \wedge \bigwedge_{m_{\mathsf{co}} \in \mathcal{M}} \mathsf{nr}(m_{\mathsf{co}}) \leq \#(\mathsf{sent}(m_{\mathsf{co}}) \cup \{\ell_F\})$$

where the formula $\mathsf{Env}_{\text{EP,byz}}$ is defined for Byzantine faults, and we assume that it does not contain constraints that restrict the values of the receive variables $\mathsf{nr}(m_{\mathsf{co}})$, as the constraints $\mathsf{nr}(m_{\mathsf{co}}) \leq \#(\mathsf{sent}(m_{\mathsf{co}}) \cup \{\ell_F\})$ are tighter than the ones defined by $\mathsf{Env}_{\text{EP,byz}}$.

Example 6.5. Recall the rSTA for the algorithm PhaseQueen, discussed in Example 6.1. Its environment assumption $\mathsf{Env}^{\Delta}_{\mathsf{co}}$ is the formula:

$$\begin{aligned}\mathsf{Env}^{\Delta}_{\mathsf{co}} \equiv \bigwedge_{\ell\in\mathcal{L}} \#\{\ell\} \geq 0 \wedge \#\mathcal{L} = n - f + 1 \wedge \#Q = 1 \wedge \#F = 1\\ \wedge \#\mathsf{sent}(m_0) \leq \mathsf{nr}(m_0) \wedge \#\mathsf{sent}(m_1) \leq \mathsf{nr}(m_1)\\ \wedge \#\mathsf{sent}(m_{q0}) \leq \mathsf{nr}(m_{q0}) \wedge \#\mathsf{sent}(m_{q1}) \leq \mathsf{nr}(m_{q1})\\ \wedge \mathsf{nr}(m_1) \leq \#\mathsf{sent}(m_1) + f\\ \wedge \mathsf{nr}(m_0) + \mathsf{nr}(m_1) \leq \#\mathsf{sent}(m_0) \cup \mathsf{sent}(m_1) + f\\ \wedge \mathsf{nr}(m_{q0}) \leq \#(\mathsf{sent}(m_{q0}) \cup \{\text{R1QF}\}) \wedge \mathsf{nr}(m_{q1}) \leq \#(\mathsf{sent}(m_{q1}) \cup \{\text{R1QF}\})\end{aligned}$$

In addition to the constraints on the number of processes in certain locations (C1) and (C2), defined in Section 4.1.2, for PhaseQueen we have the two additional constraints $\#Q = 1$ and $\#F = 1$, that restrict the number of processes in the queen locations $\mathcal{L}_{\mathsf{co}} = Q = \{\text{QV0}, \ldots, \text{R2QF}\}$ to one process, and the number of processes in the Byzantine-faulty locations $\mathcal{L}_{\mathsf{byz}} = F = \{\text{F}, \ldots, \text{R2QF}\}$ to one process, respectively. These constraints are used to model that there is a single process that acts as a queen in each phase, and that there is at most one process that is Byzantine-faulty.

The environment assumption $\mathsf{Env}^{\Delta}_{\mathsf{co}}$ also contains constraints that restrict the values of the receive variables $\mathsf{nr}(m_0), \ldots, \mathsf{nr}(m_{q1})$. Let the message type m_{q0} model a message with value 0 sent by the process acting as queen in the current phase. Then, the value $\#\mathsf{sent}(m_{q0})$ is the number of messages with value 0 sent by a correct queen in the current phase. As there is exactly one queen in each phase, $\#\mathsf{sent}(m_{q0})$ is at most one, since it may be the case that in the current phase, we have a Byzantine-faulty queen. In the rSTA of the algorithm PhaseQueen, the location R1QF encodes that a Byzantine-faulty queen performs the queen broadcast. Thus, in the environment assumption, we have that the number of received messages with value 0 by the queen is less than or equal to the number of queen messages with value 0 sent by either correct or a Byzantine-faulty queen, that is, $\#\mathsf{sent}(m_{q0}) \leq \mathsf{nr}(m_{q0}) \leq \#(\mathsf{sent}(m_{q0}) \cup \{\text{R1QF}\})$. We have a similar constraint for the message type m_{q1}, which models a queen message with value 1. For the message types that do not model a queen message, namely m_0 and m_1, we have the constraints defined by $\mathsf{Env}_{\text{EP},\mathsf{byz}}$ for Byzantine faults. □

6.2 Synchronous System Specification: Synchronous Transition System

Let $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$ be a receive STA, and $\mathbf{p} \in \mathbf{P}_{RC}$ an admissible valuation of the parameter vector $\boldsymbol{\pi}$. The synchronous transition system, defined below, represents a system of $N(\mathbf{p})$ processes, whose behavior is modeled using the rSTA.

Definition 6.10 (System $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$)**.** A *synchronous transition system* (or *system*), w.r.t. an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$ and an $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$ is the triple $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p}) = \langle S(\mathbf{p}), S_0(\mathbf{p}), T(\mathbf{p})\rangle$, where

- $S(\mathbf{p})$ is the set of *states*,
- $S_0(\mathbf{p})$ is the set of *initial states*,
- $T(\mathbf{p})$ is the *transition relation*. □

We will define the states, initial states, and transition relation of the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ in the remainder of this section.

Definition 6.11 (Parameterized system $\mathsf{STS}(\mathsf{rSTA}, \boldsymbol{\pi})$)**.** We denote by $\mathsf{STS}(\mathsf{rSTA}, \boldsymbol{\pi})$ the *parameterized synchronous transition system*, which is used to represent the infinite family $\{\mathsf{STS}(\mathsf{rSTA}, \mathbf{p}) \mid \mathbf{p} \in \mathbf{P}_{RC}\}$ of finite-state synchronous transition systems. □

States. We proceed by defining the states of the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$, for a given rSTA and admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$. Recall Definition 6.9, where we defined the environment assumption Env^{Δ} as the conjunction $\mathsf{Env}^{\Delta}_{\mathrm{CP}} \wedge \mathsf{Env}^{\Delta}_{\mathrm{EP}}$.

Definition 6.12 (States)**.** A *state* $s \in S(\mathbf{p})$ is a tuple $s = \langle \boldsymbol{\ell}, \mathbf{nr}_1, \ldots, \mathbf{nr}_{N(\mathbf{p})}, \mathbf{p} \rangle$, where:

- $\boldsymbol{\ell} \in \mathcal{L}^{N(\mathbf{p})}$ is an $N(\mathbf{p})$-dimensional vector of locations,
- $\mathbf{nr}_i \in \mathbb{N}^{|\mathcal{M}|}$, for $1 \leq i \leq N(\mathbf{p})$, is a vector of valuations of the receive variables $\mathsf{nr}(m)$, with $m \in \mathcal{M}$, for each process i,

such that $s \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$. □

In a state $s \in S(\mathbf{p})$, the vector $\boldsymbol{\ell}$ of locations is used to store the current location $s.\boldsymbol{\ell}[i] \in \mathcal{L}$ for each process i, with $1 \leq i \leq N(\mathbf{p})$, while the vector $\mathbf{nr}_i \in \mathbb{N}^{|\mathcal{M}|}$ stores the values of the receive variables for each process i. Further, each state $s \in S(\mathbf{p})$ satisfies the conjunct $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$ of the environment assumption Env^{Δ} which contains only c-propositions and their negations. The formula $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$ is used to restrict the number of processes in certain locations. We will use the formula $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$, which is also a part of the environment assumption Env^{Δ}, when we define the transition relation below.

To formally define that a state $s \in S(\mathbf{p})$ satisfies the environment constraints $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$, we define the semantics of c-propositions w.r.t. states $s \in S(\mathbf{p})$. Let $\mathsf{counters}_{\mathbf{p}} : S(\mathbf{p}) \times \mathcal{L} \to \mathbb{N}$ denote a mapping that maps a state $s \in S(\mathbf{p})$ and a location $\ell \in \mathcal{L}$ to the number of processes that are in location ℓ in the state s, that is, $\mathsf{counters}_{\mathbf{p}}(s, \ell) = |\{i \mid 1 \leq i \leq N(\mathbf{p}) \wedge s.\boldsymbol{\ell}[i] = \ell\}|$. Further, let $\boldsymbol{\kappa}(s) \in \mathbb{N}^{|\mathcal{L}|}$ denote the $|\mathcal{L}|$-dimensional vector of counters w.r.t. the state $s \in S(\mathbf{p})$, where for every location $\ell \in \mathcal{L}$, we have that $\boldsymbol{\kappa}(s)[\ell]$ stores the number of processes that are in location ℓ in the state s, that is, $\boldsymbol{\kappa}(s)[\ell] = \mathsf{counters}_{\mathbf{p}}(s, \ell)$.

Definition 6.13 (Semantics of propositions, $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$ w.r.t. states)**.** We evaluate the c-propositions in states $s \in S(\mathbf{p})$ as follows:

$$s \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \qquad \text{iff} \qquad (\boldsymbol{\kappa}(s), s.\mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$$

A state $s \in S(\mathbf{p})$ satisfies the environment constraints $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$, that is, $s \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$ iff $(\boldsymbol{\kappa}(s), s.\mathbf{p}) \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$. □

Initial States. In an initial state $s_0 \in S_0(\mathbf{p})$, the vector $\boldsymbol{\ell}$ of locations stores only initial locations, i.e., $\boldsymbol{\ell}[i] \in \mathcal{I}$, for $1 \leq i \leq N(\mathbf{p})$, and all the receive variables for all processes are initialized to 0, formalized below.

Definition 6.14 (Initial states)**.** A state $s_0 = \langle \boldsymbol{\ell}, \mathbf{nr}_1, \ldots, \mathbf{nr}_{N(\mathbf{p})}, \mathbf{p} \rangle$ is *initial*, i.e., $s_0 \in S_0(\mathbf{p}) \subseteq S(\mathbf{p})$, if:

1. $s_0.\boldsymbol{\ell} \in \mathcal{I}^{N(\mathbf{p})}$,
2. $s_0.\mathbf{nr}_i[m] = 0$, for $1 \leq i \leq N(\mathbf{p})$ and $m \in \mathcal{M}$. □

Example 6.6. Consider the rSTA of the algorithm PhaseQueen given in Figure 6.2. Suppose $\mathbf{p} = \langle 9, 2, 2 \rangle$; for this vector, we have $\mathbf{p} \in \mathbf{P}_{RC}$, since $\mathbf{p}[n] > 4 \cdot \mathbf{p}[t] \wedge \mathbf{p}[t] \geq \mathbf{p}[f]$, as $9 > 4 \cdot 2 \wedge 2 \geq 2$. An initial state $s_0 \in S_0(\mathbf{p})$ is the following state (we omit $\mathbf{p}$):

$$s_0.\boldsymbol{\ell} \quad \begin{bmatrix} \text{V0} & \text{V0} & \text{V0} & \text{V1} & \text{V1} & \text{V1} & \text{QV1} & \text{F} \end{bmatrix}$$

	$s_0.\mathbf{nr}_1$	$s_0.\mathbf{nr}_2$	$s_0.\mathbf{nr}_3$	$s_0.\mathbf{nr}_4$	$s_0.\mathbf{nr}_5$	$s_0.\mathbf{nr}_6$	$s_0.\mathbf{nr}_7$	$s_0.\mathbf{nr}_8$
m_0	0	0	0	0	0	0	0	0
m_1	0	0	0	0	0	0	0	0
m_{q0}	0	0	0	0	0	0	0	0
m_{q1}	0	0	0	0	0	0	0	0

There are $N(\mathbf{p}) = \mathbf{p}[n] - \mathbf{p}[f] + 1 = 8$ participating processes. In the state s_0, we have three processes in location V0, i.e. $\mathsf{counters}_{\mathbf{p}}(s_0, \text{V0}) = 3$, three processes in location V1, i.e., $\mathsf{counters}_{\mathbf{p}}(s_0, \text{V1}) = 3$, one process in the queen location QV1, i.e., $\mathsf{counters}_{\mathbf{p}}(s_0, \text{QV1}) = 1$, and one process in the Byzantine-faulty location F, i.e., $\mathsf{counters}_{\mathbf{p}}(s_0, \text{F}) = 1$. The remaining locations contain no processes. Thus, the environment constraint $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$ for the algorithm PhaseQueen, given in Example 6.5, holds.

Further, the locations V0, V1, QV0, and F are initial locations, and the values of all receive variables for all processes are set to 0. □

Transition relation. We now define the transition relation $T(\mathbf{p}) \subseteq S(\mathbf{p}) \times S(\mathbf{p})$. where we will use the environment constraint $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$ to restrict the values of the receive variables. A transition $(s, s') \in T(\mathbf{p})$ encodes one round in the execution of the distributed algorithm. Recall that according to the synchronous computation model, in a round, the processes send messages, receive messages, and update their variables based on the received messages. Further, all the messages sent in the current round are received in the same round. The sending of messages is captured by processes being in certain locations, while we will encode the receiving of messages by restricting the values of the receive variables using the environment constraint $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$. The process variable updates are

captured by moving processes from one location to another, based on the values of the receive variables.

Definition 6.15. The transition relation $T(\mathbf{p})$ is a binary relation $T(\mathbf{p}) \subseteq S(\mathbf{p}) \times S(\mathbf{p})$, where a pair $(s, s') \in S(\mathbf{p}) \times S(\mathbf{p})$ of states is a *transition* in $T(\mathbf{p})$, i.e., $(s, s') \in T(\mathbf{p})$ iff for every process i, with $1 \leq i \leq N(\mathbf{p})$:

1. $0 \leq s'.\mathbf{nr}_i[m] \leq N(\mathbf{p})$, for every message type $m \in \mathcal{M}$, such that $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models \mathsf{Env}^{\Delta}_{\mathrm{EP}}$,
2. there exists $r^{\Delta} \in \mathcal{R}^{\Delta}$ such that:
   - $s.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{from}$,
   - $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models r^{\Delta}.\varphi$,
   - $s'.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{to}$.
3. $s'.\mathbf{p} = s.\mathbf{p}$,
4. $s' \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$. □

In a transition from a state $s \in S(\mathbf{p})$ to a state $s' \in S(\mathbf{p})$ using the relation $T(\mathbf{p})$, the receive variables and locations of each process are updated. To update the receive variables, for each message type $m \in \mathcal{M}$ and each process i, with $1 \leq i \leq N(\mathbf{p})$, the value $s'.\mathbf{nr}_i[m]$ of the receive variable $\mathsf{nr}(m)$ of process i is assigned a value non-deterministically, such that the environment constraint $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$ is satisfied. By constraining the value of the receive variable $\mathsf{nr}(m)$ of process i to a value in the range from 0 to $N(\mathbf{p})$, we ensure that the number of received messages of type m is non-negative, and that it does not exceed the number of participating processes. Further, we require that the new values of the receive variables for process i and the number of sent messages by the processes (given by the number of processes in certain locations, stored in the vector $\boldsymbol{\kappa}(s)$) satisfy the environment constraint $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$. This ensures that the receive variables of each process are assigned values that satisfy the constraints of the environment assumption, which, in the case of the synchronous computation model, captures that all messages sent by correct processes in a round are received in the same round, and that the number of messages of type m, received by process i, is bounded by above by the total number of messages of type m, sent by both correct and faulty processes. To update the locations, each process picks a rule $r^{\Delta} \in \mathcal{R}^{\Delta}$ that it applies to update its location. A rule $r^{\Delta} \in \mathcal{R}^{\Delta}$ can be applied by a process i if the process i is in the location $r^{\Delta}.\mathit{from}$ in the state s, and if the newly assigned values of the receive variables of process i in the state s' satisfy the receive guard $r^{\Delta}.\varphi$. If this is the case, the process i updates its location to $r^{\Delta}.\mathit{to}$ in the state s'. The parameter values remain unchanged, and we additionally require that the state s' satisfies the environment constraints $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$, i.e., that it is a valid state.

Example 6.7. Consider again the rSTA of the algorithm PhaseQueen given in Figure 6.2 and suppose that $\mathbf{p} = \langle 9, 2, 2 \rangle$. The state s_1, given below, is in relation $T(\mathbf{p})$ with the state s_0, presented in Example 6.6.

$$
\begin{array}{cc}
s_1.\ell & \begin{bmatrix} \text{R1V1} & \text{R1V0Q} & \text{R1V1} & \text{R1V0Q} & \text{R1V1} & \text{R1V1} & \text{R1QV1} & \text{R1F} \end{bmatrix} \\
& \begin{array}{cccccccc} s_1.\mathbf{nr}_1 & s_1.\mathbf{nr}_2 & s_1.\mathbf{nr}_3 & s_1.\mathbf{nr}_4 & s_1.\mathbf{nr}_5 & s_1.\mathbf{nr}_6 & s_1.\mathbf{nr}_7 & s_1.\mathbf{nr}_8 \end{array} \\
m_0 & \begin{bmatrix} 3 & 5 & 3 & 5 & 3 & 4 & 4 & 3 \end{bmatrix} \\
m_1 & \begin{bmatrix} 5 & 4 & 6 & 4 & 6 & 5 & 5 & 4 \end{bmatrix} \\
m_{q0} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\
m_{q1} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}
\end{array}
$$

In s_1, the values of the receive variables that count messages of type m_0 and m_1, sent out in the first round of the phase, are updated. In Example 6.1, we defined $\mathsf{sent}(m_0) = \{\text{V0}, \text{QV0}\}$ and $\mathsf{sent}(m_1) = \{\text{V1}, \text{QV1}\}$. In the state s_0, we have $\#\mathsf{sent}(m_0) = \mathsf{counters}_{\mathbf{p}}(s_0, \text{V0}) = 3$ correct processes that send a message of type m_0 and $\#\mathsf{sent}(m_1) = \mathsf{counters}_{\mathbf{p}}(s_0, \text{V1}) + \mathsf{counters}_{\mathbf{p}}(s_0, \text{QV1}) = 4$ correct processes that send a message of type m_1. The environment assumption $\mathsf{Env}^{\Delta}_{\text{EP}}$ restricts the values of the receive variables as follows:

$$
\begin{aligned}
\#\mathsf{sent}(m_0) &\leq \mathsf{nr}(m_0), && \text{i.e., } 3 \leq s_1.\mathbf{nr}_i[m_0] \\
\#\mathsf{sent}(m_1) &\leq \mathsf{nr}(m_1), && \text{i.e., } 4 \leq s_1.\mathbf{nr}_i[m_1] \\
\mathsf{nr}(m_1) &\leq \#\mathsf{sent}(m_1) + f, && \text{i.e., } s_1.\mathbf{nr}_i[m_1] \leq 6 \\
\mathsf{nr}(m_0) + \mathsf{nr}(m_1) &\leq \#\mathsf{sent}(m_0) + \#\mathsf{sent}(m_1) + f, && \text{i.e., } s_1.\mathbf{nr}_i[m_0] + s_1.\mathbf{nr}_i[m_1] \leq 9
\end{aligned}
$$

for each $1 \leq i \leq N(\mathbf{p})$. Using this assignment of values to the receive variables, processes 2 and 4 satisfy the guard φ_2, processes 1, 3, 5, and 6 satisfy the guard φ_4, and process 7 satisfies the guard φ_8. Thus, to update their locations in the state s_2, processes 2 and 4 apply the rule, guarded by φ_2, that moves them from V0 and V1, respectively, to R1V0Q. Similarly, processes 1, 3, 5, and 6 move to R1V1, process 7 moves to R1QV1, and process 8 moves to the only possible location it can move to, that is, to R1F. □

Temporal Logic for Specifying Properties To express properties of synchronous fault-tolerant distributed algorithms modeled using rSTA, we use the same temporal logic and atomic propositions as for STA, defined in Chapter 4. That is, we use LTL properties over c-propositions.

Example 6.8. We formalize the safety properties *Validity* and *Agreement* for the algorithm PhaseQueen, whose pseudocode is given in Figure 6.1, and whose rSTA is depicted in Figure 6.2.

- *Validity.* A value that is not an initial value of any correct process is not a value that is decided on. We express this using two formulas: one that checks if all correct processes have an initial value different than 0, and an analogous formula that checks the same condition for the value 1.

$$
\begin{aligned}
\#\{\text{V0}, \text{QV0}\} = 0 &\rightarrow \mathbf{G}\,(\mathit{decided} \rightarrow \#\{\text{V0}, \text{QV0}\} = 0) \\
\#\{\text{V1}, \text{QV1}\} = 0 &\rightarrow \mathbf{G}\,(\mathit{decided} \rightarrow \#\{\text{V1}, \text{QV1}\} = 0)
\end{aligned}
$$

- *Agreement.* No two correct processes decide on different values.

$$\mathbf{G}\left(\mathit{decided} \rightarrow (\#\{\text{v0}, \text{qv0}\} = 0 \vee \#\{\text{v1}, \text{qv1}\} = 0)\right)$$

The flag *decided* is true when the algorithm has ran for $t + 1$ phases. □

6.3 Discussion

In this chapter, we introduced a new variant of synchronous threshold automata, which facilitates the step of producing a formal model of a synchronous fault-tolerant distributed algorithm. By adding receive variables and expressing guards over them, we obtain a formalization which is closer to the pseudocode found in the literature. This formalization eliminates the possibility of introducing bugs in the manual encoding of the algorithm when coming up with the correct guard expressions. That is, the synchronous threshold automata with receive variables can be seen as having a one-to-one correspondence to the pseudocode of the algorithm they are modeling.

For the purpose of modeling *asynchronous* fault-tolerant distributed algorithms, threshold automata with receive variables were introduced in [SKWZ20]. In contrast to the synchronous computation model, which is the focus of this thesis, the asynchronous computation model does not impose any bound on the time it takes for a message to be delivered. That is, in the environment assumption of the asynchronous threshold automata with receive variables, no lower bound is imposed on the values of the receive variables.

The synchronous threshold automata with receive variables introduced in this chapter stand at the middle ground between the two process and environment specifications we introduced in Chapters 2 and 4. On the one hand, they allow storing the received messages explicitly, as was done using the process neighborhood variables in Chapter 2. On the other hand, they encode local transitions as guarded rules from one location to another, as was done in the standard STA, defined in Chapter 4. In the next chapter, we will construct an STA from a given rSTA, which will allow us to apply the bounded model checking technique, introduced in Chapter 5, to verify the safety properties of an algorithm encoded using rSTA.

CHAPTER 7

Eliminating Receive Message Counters

In this chapter, we propose an automated method to translate guard expressions over the *local receive* variables into guard expressions over the number of *globally sent* messages. The input is a synchronous threshold automaton with receive variables, rSTA, whose rules and environment assumption contain expressions over the receive variables, as defined in Chapter 6. The output is a synchronous threshold automaton STA, where the receive variables are eliminated, and where the guards and environment assumption contain expressions over the number of sent messages, as defined in Chapter 4.

Eliminating Receive Variables. We argued that the root cause that an action that a process takes becomes enabled is not the fact that the number of *received* messages has passed a certain threshold (which is information local to a process), but rather the fact that enough processes have *sent* messages (which is a fact global to the system). This leads to redundancy in the formalization of the process behavior: the information about whether an action is enabled is present in the global state of the system, as well as in the local state of the processes. In [JKS+13], it was shown that this redundancy may lead to spurious counterexamples when applying abstraction-based model checking, which makes abstraction-based model checking impractical even for very simple benchmarks.

In Section 7.1, we remove this redundancy by eliminating the receive variables in an rSTA, using quantifier elimination for Presburger arithmetic [Pre29, Coo72, Pug92]. A guard expression over the number of sent messages is obtained by applying quantifier elimination to a given receive guard, strengthened by additional constraints that impose bounds on the values of the receive variables, which are existentially quantified. The output of quantifier elimination is a quantifier-free expression over the number of sent messages, and constitutes a valid input to the bounded model checking technique we introduced in Chapter 5.

Soundness and Completeness. In Section 7.2, we show that this method is sound and complete. That is, we show the existence of a bisimulation relation between the the counter system induced by the produced synchronous threshold automaton and the system induced by the original receive synchronous threshold automaton. Thus, eliminating the receive message counters preserves temporal properties.

Experimental Evaluation. We specified the control flow of several fault-tolerant distributed algorithms from the literature using receive synchronous threshold automata, whose guards are expressions over the receive variables. We implemented our quantifier-elimination-based technique in a prototype, which we used to obtain the corresponding translated guards. We compared the automatically generated automata to the existing manually encoded automata and found flaws in several manually encoded automata, such as wrong guards or missing rules.

We used our bounded model checking tool, which implements the technique presented in Chapter 5, on the resulting synchronous threshold automata and verified their safety properties. We report on the experimental results in Section 7.3.

7.1 Abstracting rSTA to STA

Given an rSTA, our goal is to construct an STA, which differs from the rSTA only in the guards on its rules and the environment assumption. For each rule $r^{\Delta} \in \mathcal{R}^{\Delta}$ in the rSTA, whose guard $r^{\Delta}.\varphi$ is a receive guard, we will construct a rule $r \in \mathcal{R}$ in the STA, such that the guard $r.\varphi$ is a Boolean combination of c-propositions. We will perform the abstraction in two steps: (i) we will strengthen each receive guard $r^{\Delta}.\varphi$, occurring on the rules $r^{\Delta} \in \mathcal{R}^{\Delta}$ of the rSTA, with the constraints imposed by the faulty environment and the synchronous computation model, encoded in the environment assumption Env^{Δ}, and (ii) we will eliminate the receive variables from the receive guards and environment assumptions of rSTA to obtain the guards and environment assumption of STA.

7.1.1 Guard Strengthening

Let $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$ be a receive STA, where the rules $r^{\Delta} \in \mathcal{R}^{\Delta}$ have guards containing expressions over the receive variables $\mathsf{nr}(m) \in \Delta$, and where the environment assumption $\mathsf{Env}^{\Delta} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}} \wedge \mathsf{Env}^{\Delta}_{\mathrm{EP}}$ is a conjunction of two environment constraints, $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$ and $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$, where the latter restricts the values of the receive variables. Recall that in Section 6.1.2, we defined different environment constraints $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$ for the different fault models. In general, these constraints express that for each message type $m \in \mathcal{M}$, the receive variable $\mathsf{nr}(m)$ is assigned a value which is greater or equal to the number of messages of type m sent by correct processes, and which is smaller or equal to the total number of messages of type m, sent by both correct and faulty processes (e.g., $\#\mathsf{sent}(m) \leq \mathsf{nr}(m) \leq \#\mathsf{sent}(m) + \#\mathsf{sent}_{\mathsf{cr}}(m)$ for crash faults). As a first step towards eliminating the receive variables from the receive guards, we strengthen the rules from

the set $\mathcal{R}^{\Delta}$, by adding the environment constraints $\mathsf{Env}^{\Delta}_{\mathrm{EP}}$ to their guards in order to bound the values of the receive variables.

Definition 7.1 (Strengthened rules)**.** Given a rule $r^{\Delta} \in \mathcal{R}^{\Delta}$, its *strengthened rule* is the rule $\widehat{r}^{\Delta} = \mathsf{strengthen}(r^{\Delta})$, such that

- $\widehat{r}^{\Delta}.\mathit{from} = r^{\Delta}.\mathit{from}$,
- $\widehat{r}^{\Delta}.\mathit{to} = r^{\Delta}.\mathit{to}$,
- $\widehat{r}^{\Delta}.\varphi = r^{\Delta}.\varphi \wedge \mathsf{Env}^{\Delta}$.

We denote by $\widehat{\mathcal{R}}^{\Delta} = \{\mathsf{strengthen}(r^{\Delta}) \mid r^{\Delta} \in \mathcal{R}^{\Delta}\}$ the set of strengthened rules in $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$. □

Example 7.1. Consider the receive guard $\varphi_1 \equiv \mathsf{nr}(m_0) \geq 1$, occurring on the rules of the rSTA for the algorithm FloodMin, depicted in Figure 6.3. Recall the environment assumption Env^{Δ} for the algorithm FloodMin, described in Example 6.2. The strengthened guard $\widehat{\varphi}_1$ is:

$$\begin{aligned}\widehat{\varphi}_1 &\equiv \varphi_1 \wedge \mathsf{Env}^{\Delta} \\ &\equiv \mathsf{nr}(m_0) \geq 1 \\ &\quad \wedge \bigwedge_{\ell \in \mathcal{L}} \#\{\ell\} \geq 0 \wedge \#\mathcal{L} = n \wedge \#\{\mathrm{v0}_{\mathsf{cr}}, \mathrm{v1}_{\mathsf{cr}}, \maltese\} \leq f \\ &\quad \wedge \#\{\mathrm{v0}\} \leq \mathsf{nr}(m_0) \leq \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \wedge \#\{\mathrm{v1}\} \leq \mathsf{nr}(m_1) \leq \#\{\mathrm{v1}, \mathrm{v1}_{\mathsf{cr}}\}\end{aligned}$$ □

7.1.2 Eliminating the Receive Variables

Let $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$ be a receive STA, and let $\widehat{\mathcal{R}}^{\Delta}$ be the set of strengthened rules, as defined in Definition 7.1. We construct an $\mathsf{STA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Pi, RC, \mathsf{Env})$ whose locations, initial locations, and parameters are the same as in rSTA. Before we define the rules $\mathcal{R}$ and environment assumption Env of the constructed STA, we define the mapping eliminate, which applies quantifier elimination to eliminate the receive variables.

Definition 7.2 (Eliminating Δ)**.** Let ϕ be a propositional formula over r-propositions, c-propositions, and e-propositions. Let $\boldsymbol{\delta} = \langle \mathsf{nr}(m_1), \ldots, \mathsf{nr}(m_{|\mathcal{M}|}) \rangle$ denote the $|\mathcal{M}|$-dimensional *receive variables vector*, and let QE denote the quantifier elimination procedure for Presburger arithmetic. The formula:

$$\mathsf{eliminate}(\phi) = \mathsf{QE}(\exists \boldsymbol{\delta}\ \phi)$$

is a quantifier-free formula, with no occurrence of the receive variables $\mathsf{nr}(m) \in \Delta$, for $m \in \mathcal{M}$, which is logically equivalent to the formula $\exists \boldsymbol{\delta}\ \phi$. □

To construct a rule $r \in \mathcal{R}$ of an STA, given a rule $r^{\Delta} \in \mathcal{R}^{\Delta}$ of an rSTA, we will apply the mapping eliminate to each guard of the strengthened rule $\widehat{r}^{\Delta} \in \widehat{\mathcal{R}}^{\Delta}$, where $\widehat{r}^{\Delta} = \mathsf{strengthen}(r^{\Delta})$. The result of quantifier elimination is a quantifier-free formula over c-propositions, which is logically equivalent to $\exists \boldsymbol{\delta}\ \widehat{r}^{\Delta}.\varphi$.

Definition 7.3 (Constructed $\mathcal{R}$)**.** Given a rule $r^{\Delta} \in \mathcal{R}^{\Delta}$ in an rSTA, its corresponding rule in the constructed STA is the rule $r = \mathsf{construct}(r^{\Delta}) \in \mathcal{R}$, such that:

- $r.\mathit{from} = r^{\Delta}.\mathit{from}$,
- $r.\mathit{to} = r^{\Delta}.\mathit{to}$,
- $r.\varphi = \mathsf{eliminate}(\widehat{r}^{\Delta}.\varphi)$, where $\widehat{r}^{\Delta} = \mathsf{strengthen}(r^{\Delta})$. □

Example 7.2. Recall that in Example 7.1, we presented the strengthened guard $\widehat{\varphi}_1$ of the guard φ_1, occurring on the rules of the rSTA for the algorithm FloodMin (Figure 6.3).

The result of applying quantifier elimination to the formula $\exists \boldsymbol{\delta}\ \widehat{\varphi}_1$ is the following quantifier-free formula:

$$\begin{aligned}\mathsf{eliminate}(\widehat{\varphi}_1) &\equiv \mathsf{QE}(\exists \boldsymbol{\delta}\ \widehat{\varphi}_1)\\ &\equiv \#\{\mathrm{v0}, \mathrm{v0}_{\mathsf{cr}}\} \geq 1 \wedge \#\{\mathrm{v0}_{\mathsf{cr}}\} \geq 0 \wedge \#\{\mathrm{v1}_{\mathsf{cr}}\} \geq 0 \wedge \mathsf{Env}_{\mathrm{CP},\mathsf{cr}}\end{aligned}$$

where $\mathsf{Env}_{\mathrm{CP},\mathsf{cr}}$ are the constraints of the environment assumption for the crash fault model that do not contain receive variables. □

Recall that $\mathsf{Env}^{\Delta} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}} \wedge \mathsf{Env}^{\Delta}_{\mathrm{EP}}$. We obtain the environment assumption of an STA as follows.

Definition 7.4 (Constructed Env)**.** Given an environment assumption $\mathsf{Env}^{\Delta} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}} \wedge \mathsf{Env}^{\Delta}_{\mathrm{EP}}$ of an rSTA, the environment assumption Env of the constructed STA is the formula $\mathsf{Env} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}}$. □

The following proposition is a consequence of the definition of eliminate and quantifier elimination.

Proposition 7.1. *For every strengthened rule $\widehat{r}^{\Delta} \in \widehat{\mathcal{R}}^{\Delta}$ and every tuple $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$, $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$, and $\mathbf{p} \in \mathbf{P}_{RC}$, we have:*

$$(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) \models \widehat{r}^{\Delta}.\varphi \quad \textit{implies} \quad (\boldsymbol{\kappa}, \mathbf{p}) \models \mathsf{eliminate}(\widehat{r}^{\Delta}.\varphi)$$ □

Note that the converse of Proposition 7.1 does not hold in general. That is, in the case when $\widehat{r}^{\Delta}.\varphi$ is a receive guard, $(\boldsymbol{\kappa}, \mathbf{p}) \models \mathsf{eliminate}(\widehat{r}^{\Delta}.\varphi)$ does not imply that $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) \models \widehat{r}^{\Delta}.\varphi$, for every $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$. However, by quantifier elimination, we have that $(\boldsymbol{\kappa}, \mathbf{p}) \models \mathsf{eliminate}(\widehat{r}^{\Delta}.\varphi)$ implies $(\boldsymbol{\kappa}, \mathbf{p}) \models \exists \boldsymbol{\delta}\ \widehat{r}^{\Delta}.\varphi$.

7.2 Soundness and Completeness

In this section, we show that this construction of an STA is sound and complete. That is, given an rSTA and an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$, we show that there exists a bisimulation relation between the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$, induced by rSTA and $\mathbf{p}$, and the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, induced by the constructed STA and $\mathbf{p}$. The existence of a bisimulation implies that $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ and $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ satisfy the same CTL^* formulas [BK08].

As we saw in Chapters 4 and 6, we express the properties of synchronous fault-tolerant distributed algorithms using LTL formulas over c-propositions. Let CP denote the set of c-propositions. We define two labeling functions, $\lambda_{S(\mathbf{p})}$ and $\lambda_{\Sigma(\mathbf{p})}$, where the function $\lambda_{S(\mathbf{p})} : S(\mathbf{p}) \to 2^{\mathrm{CP}}$ assigns to a state $s \in S(\mathbf{p})$ the set of c-propositions from CP that hold in s, and the function $\lambda_{\Sigma(\mathbf{p})} : \Sigma(\mathbf{p}) \to 2^{\mathrm{CP}}$ is defined analogously.

We recall the definition of a bisimulation relation from [BK08], and adapt it to the two systems $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p}) = \langle S(\mathbf{p}), S_0(\mathbf{p}), T(\mathbf{p}) \rangle$ and $\mathsf{CS}(\mathsf{STA}, \mathbf{p}) = \langle \Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p}) \rangle$. Recall the Definitions 6.15 and 4.15, which define the transition relations of the $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ and $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, respectively.

Definition 7.5 (Bisimulation [BK08])**.** A binary relation $B(\mathbf{p}) \subseteq S(\mathbf{p}) \times \Sigma(\mathbf{p})$ is a *bisimulation relation* if:

1. for every initial state $s_0 \in S_0(\mathbf{p})$, there exists an initial configuration $\sigma_0 \in I(\mathbf{p})$ such that $(s_0, \sigma_0) \in B(\mathbf{p})$,
2. for every initial configuration $\sigma_0 \in I(\mathbf{p})$, there exists an initial state $s_0 \in S_0(\mathbf{p})$ such that $(s_0, \sigma_0) \in B(\mathbf{p})$,
3. for every $(s, \sigma) \in B(\mathbf{p})$ it holds that:
   a) $\lambda_{S(\mathbf{p})}(s) = \lambda_{\Sigma(\mathbf{p})}(\sigma)$,
   b) for every state $s' \in S(\mathbf{p})$ such that $(s, s') \in T(\mathbf{p})$, there exists a transition $tr \in Tr(\mathbf{p})$ and a configuration $\sigma' \in \Sigma(\mathbf{p})$ such that $(\sigma, tr, \sigma') \in R(\mathbf{p})$ and $(s', \sigma') \in B(\mathbf{p})$.
   c) for every transition $tr \in Tr(\mathbf{p})$ and configuration $\sigma' \in \Sigma(\mathbf{p})$ such that $(\sigma, tr, \sigma') \in R(\mathbf{p})$, there exists a state $s' \in S(\mathbf{p})$ such that $(s, s') \in T(\mathbf{p})$ and $(s', \sigma') \in B(\mathbf{p})$.

To show that there exists a bisimulation relation between the two systems $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ and $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, we introduce an *abstraction mapping* from the set $S(\mathbf{p})$ of states of $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ to the set $\Sigma(\mathbf{p})$ of configurations of $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$.

Definition 7.6 (Abstraction mapping $\alpha_{\mathbf{p}}$)**.** The *abstraction mapping* $\alpha_{\mathbf{p}} : S(\mathbf{p}) \to \Sigma(\mathbf{p})$ maps $s \in S(\mathbf{p})$ to $\sigma \in \Sigma(\mathbf{p})$, such that:

$$\sigma = \alpha_{\mathbf{p}}(s) \quad \text{iff} \quad \sigma = (\boldsymbol{\kappa}(s), s.\mathbf{p}) \qquad \square$$

The following lemma is a consequence of the definition of the abstraction mapping $\alpha_{\mathbf{p}}$ and the semantics of c-propositions. It shows that a state and its abstraction satisfy the same c-propositions.

Lemma 7.1. *Let $s \in S(\mathbf{p})$, $\sigma \in \Sigma(\mathbf{p})$, such that $\sigma = \alpha_{\mathbf{p}}(s)$. For every c-proposition $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ from* CP*, it holds that:*

$$s \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \qquad \textit{iff} \qquad \sigma \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$$

Proof. Suppose $s \in S(\mathbf{p})$ and $\sigma \in \Sigma(\mathbf{p})$ such that $\sigma = \alpha_{\mathbf{p}}(s)$. Using Definition 7.6, we have:

$$\begin{aligned} s \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \;\; \text{iff} \;\; & (\boldsymbol{\kappa}(s), s.\mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \\ \text{iff} \;\; & (\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \\ \text{iff} \;\; & \sigma \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \end{aligned}$$

□

The main result of this section is stated in the theorem below.

Theorem 7.1. *Let* $\mathsf{STS}(\mathit{rSTA}, \mathbf{p})$ *be the system induced by a given* rSTA *and an arbitrary* $\mathbf{p} \in \mathbf{P}_{RC}$*. Let* $\mathsf{CS}(\mathit{STA}, \mathbf{p})$ *be the counter system induced by the* STA *constructed from* rSTA *and the admissible valuation* $\mathbf{p}$*. The binary relation* $B(\mathbf{p}) = \{(s, \sigma) \mid s \in S(\mathbf{p}), \sigma \in \Sigma(\mathbf{p}), \sigma = \alpha_{\mathbf{p}}(s)\}$ *is a bisimulation relation.*

Proof. We show that the binary relation $B(\mathbf{p}) = \{(s, \sigma) \mid s \in S(\mathbf{p}), \sigma \in \Sigma(\mathbf{p}), \sigma = \alpha_{\mathbf{p}}(s)\}$ satisfies all the conditions in Definition 7.5.

1. Let $s_0 \in S_0(\mathbf{p})$ be an arbitrary initial state. By the definition of the abstraction mapping, we have that $\sigma_0 = \alpha_{\mathbf{p}}(s_0) = (\boldsymbol{\kappa}(s_0), s_0.\mathbf{p})$, where for every $\ell \in \mathcal{I}$ we have $\sigma_0.\boldsymbol{\kappa}[\ell] = |\{i \mid s_0.\boldsymbol{\ell}[i] = \ell\}|$. For every $\ell \in \mathcal{L} \setminus \mathcal{I}$ it holds that $|\{i \mid s_0.\boldsymbol{\ell}[i] = \ell\}| = 0$. Since $s_0 \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$ by definition, as a consequence of Lemma 7.1, we have $\sigma_0 \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$. Since $\mathsf{Env} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}}$, we obtain $\sigma_0 \models \mathsf{Env}$, hence $\sigma_0 \in \Sigma(\mathbf{p})$ is a valid configuration. Further, by the definition of initial configurations, we have $\sigma_0 = \alpha_{\mathbf{p}}(s_0) \in I(\mathbf{p})$. Thus $(s_0, \sigma_0) \in B(\mathbf{p})$.

2. Let $\sigma_0 \in I(\mathbf{p})$ be an arbitrary initial configuration. We construct a tuple $s_0 = \langle \boldsymbol{\ell}, \mathbf{nr}_1, \ldots, \mathbf{nr}_{N(\mathbf{p})}, \mathbf{p} \rangle$ with $s_0.\boldsymbol{\ell} \in \mathcal{I}^{N(\mathbf{p})}$ and $s_0.\mathbf{nr}_i[m] = 0$, for $1 \leq i \leq N(\mathbf{p})$ and $m \in \mathcal{M}$, such that $s_0.\mathbf{p} = \sigma_0.\mathbf{p}$. For every $s_0.\boldsymbol{\ell}[i]$, with $1 \leq i \leq N(\mathbf{p})$, we choose a location $\ell \in \mathcal{I}$, such that the resulting tuple $s_0.\boldsymbol{\ell}$ satisfies the constraint $\sigma_0.\boldsymbol{\kappa}[\ell] = \mathsf{counters}_{\mathbf{p}}(s_0, \ell)$. By the definition of the abstraction mapping, we have $\sigma_0 = \alpha_{\mathbf{p}}(s_0)$. From $\sigma_0 \models \mathsf{Env}$ and $\mathsf{Env} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}}$, by Lemma 7.1, we get $s_0 \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$, hence s_0 is a valid state, that is $s_0 \in S(\mathbf{p})$. By the definition of initial states, $s_0 \in S_0(\mathbf{p})$, and thus, $(\sigma_0, s_0) \in B(\mathbf{p})$.

3. Suppose $(s, \sigma) \in B(\mathbf{p})$. Then $\sigma = \alpha_{\mathbf{p}}(s)$.

   a) By Lemma 7.1 and the definition of the labeling functions, we obtain $\lambda_{S(\mathbf{p})}(s) = \lambda_{\Sigma(\mathbf{p})}(\sigma)$.

   b) Suppose $(s, s') \in T(\mathbf{p})$. By the definition of the transition relation $T(\mathbf{p})$, we have that $s' \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$ and for every i, where $1 \leq i \leq N(\mathbf{p})$, we have $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models \mathsf{Env}^{\Delta}_{\mathrm{EP}}$. Further, for every process i, for $1 \leq i \leq N(\mathbf{p})$, there exists a rule $r^{\Delta} \in \mathcal{R}^{\Delta}$ such that $s.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{from}$, $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models r^{\Delta}.\varphi$, and $s'.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{to}$. Let $\mathcal{R}^{\Delta}|_{(s,s')} \subseteq \mathcal{R}^{\Delta}$ denote the set of rules that the processes applied in the transition $(s, s') \in T(\mathbf{p})$. For every rule $r^{\Delta} \in \mathcal{R}^{\Delta}|_{(s,s')}$, let:

$$\mathsf{total}(r^{\Delta}) = |\{i \mid s.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{from}, (s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models r^{\Delta}.\varphi, s'.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{to}\}|$$

   denote the number of processes that apply the rule r^{Δ} in the transition $T(\mathbf{p})$. We construct a transition $tr \in \mathit{Tr}(\mathbf{p})$, with $tr : \mathcal{R} \to \mathbb{N}$ as follows:

$$tr(r) = \begin{cases} 0 & \text{if } r^{\Delta} \notin \mathcal{R}^{\Delta}|_{(s,s')} \\ \mathsf{total}(r^{\Delta}) & \text{otherwise} \end{cases} \quad \text{for } r \in \mathcal{R} \text{ and } r = \mathsf{construct}(r^{\Delta})$$

   To show that tr is a transition which is enabled in σ, we need to show that (i) for every rule $r \in \mathcal{R}$ with a non-zero factor, we have that $\sigma \models r.\varphi$, and (ii) σ is the origin $o(tr)$ of the transition tr.
   For (i), consider an arbitrary $r^{\Delta} \in \mathcal{R}^{\Delta}|_{(s,s')}$, and let $r = \mathsf{construct}(r^{\Delta})$. In the transition tr, we have $tr(r) > 0$. From $r^{\Delta} \in \mathcal{R}^{\Delta}|_{(s,s')}$, we have that there exists a process i, with $1 \leq i \leq N(\mathbf{p})$, such that $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models r^{\Delta}.\varphi$. By our assumption, $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models \mathsf{Env}^{\Delta}_{\mathrm{EP}}$, and hence, by the definition of the strengthened guards, $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models \widehat{r}^{\Delta}.\varphi$. By Proposition 7.1, we obtain $(\boldsymbol{\kappa}(s), s.\mathbf{p}) \models \mathsf{eliminate}(\widehat{r}^{\Delta}.\varphi)$. From this, the definition of the abstraction mapping, and since $r.\varphi = \mathsf{eliminate}(\widehat{r}^{\Delta}.\varphi)$, we get $\sigma \models r.\varphi$.
   For (ii), recall that for every $\ell \in \mathcal{L}$, we have $o(tr).\boldsymbol{\kappa}[\ell] = \sum_{r.\mathit{from}=\ell} tr(r)$. That is, for every $\ell \in \mathcal{L}$, we have:

$$\begin{aligned} o(tr).\boldsymbol{\kappa}[\ell] = \sum_{r.\mathit{from}=\ell} tr(r) &= \sum_{\substack{r.\mathit{from}=\ell \\ r=\mathsf{construct}(r^{\Delta})}} \mathsf{total}(r^{\Delta}) \\ &= |\{i \mid s.\boldsymbol{\ell}[i] = \ell\}| = \mathsf{counters}_{\mathbf{p}}(s, \ell) = \boldsymbol{\kappa}(s)[\ell] = \sigma.\boldsymbol{\kappa}[\ell] \end{aligned}$$

   What remains to show is that $g(tr) = \alpha_{\mathbf{p}}(s')$. By the definition of the goal of a transition tr, for every $\ell \in \mathcal{L}$, we have:

$$\begin{aligned} g(tr).\boldsymbol{\kappa}[\ell] = \sum_{r.\mathit{to}=\ell} tr(r) &= \sum_{\substack{r.\mathit{to}=\ell \\ r=\mathsf{construct}(r^{\Delta})}} \mathsf{total}(r^{\Delta}) \\ &= |\{i \mid s'.\boldsymbol{\ell}[i] = \ell\}| = \mathsf{counters}_{\mathbf{p}}(s', \ell) = \boldsymbol{\kappa}(s')[\ell] \end{aligned}$$

By the definition of the abstraction mapping, we get $g(tr) = \sigma' = \alpha_{\mathbf{p}}(s')$. Since $s' \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$ and since $\mathsf{Env} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}}$, by Lemma 7.1, we have $\sigma' \models \mathsf{Env}$, hence $\sigma' \in \Sigma(\mathbf{p})$ is a valid configuration, and by the definition of the transition relation $R(\mathbf{p})$, we have $(\sigma, \sigma') \in R(\mathbf{p})$. Thus $(s', \sigma') \in B(\mathbf{p})$.

c) Suppose $(\sigma, tr, \sigma') \in R(\mathbf{p})$. By the definition of the transition relation $R(\mathbf{p})$, we have $\sigma = o(tr)$ and $\sigma' = g(tr)$. Let $\mathcal{R}|_{tr}$ denote the set of rules $r \in \mathcal{R}$ with non-zero factors in the transition tr, that is, $r \in \mathcal{R}|_{tr}$ iff $tr(r) > 0$. We will build a transition $(s, s') \in T(\mathbf{p})$ as follows.

Partition the set $P = \{i \mid 1 \leq i \leq N(\mathbf{p})\}$ of processes into mutually disjoint sets $I(s, r) \subseteq P$, for $r \in \mathcal{R}|_{tr}$, such that $I(s, r) = \{i \mid s.\boldsymbol{\ell}[i] = r.\mathit{from}\}$ and $|I(s, r)| = tr(r)$. That is, the set $I(s, r)$ contains the processes that are in location $r.\mathit{from} = r^{\Delta}.\mathit{from}$ in the state s, and to which we will apply the rule r^{Δ} in the transition $(s, s') \in T(\mathbf{p})$, where $r = \mathsf{construct}(r^{\Delta})$.

W.l.o.g., pick an arbitrary $r \in \mathcal{R}|_{tr}$. By the definition of $I(s, r)$, we assume that for $r^{\Delta} \in \mathcal{R}^{\Delta}$, such that $r = \mathsf{construct}(r^{\Delta})$, we have $s.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{from}$, for $i \in I(s, r)$. From $r \in \mathcal{R}|_{tr}$ and $tr(r) > 0$, we have that $\sigma \models r.\varphi$. Further, as $\sigma = \alpha_{\mathbf{p}}(s)$, by Lemma 7.1, we get $s \models r.\varphi$. By Definitions 7.2 and 7.3, we have that $r.\varphi = \mathsf{eliminate}(\widehat{r}^{\Delta}.\varphi) = \mathsf{QE}(\exists \boldsymbol{\delta}\, \widehat{r}^{\Delta}.\varphi)$, and by quantifier elimination, $s \models \exists \boldsymbol{\delta}\, \widehat{r}^{\Delta}.\varphi$. For every process $i \in I(s, r)$, for which we assume $s.\boldsymbol{\ell}[i] = r.\mathit{from} = r^{\Delta}.\mathit{from}$, we pick a valuation $\mathbf{d}_i \in \mathbb{N}^{|\mathcal{M}|}$, such that $s \models \widehat{r}^{\Delta}.\varphi[\mathbf{d}_i/\boldsymbol{\delta}]$, i.e., $(\boldsymbol{\kappa}(s), s.\mathbf{p}) \models \widehat{r}^{\Delta}.\varphi[\mathbf{d}_i/\boldsymbol{\delta}]$. We build a tuple $s' = \langle \boldsymbol{\ell}, \mathbf{nr}_1, \ldots, \mathbf{nr}_{N(\mathbf{p})}, \mathbf{p} \rangle$, where $s'.\mathbf{p} = s.\mathbf{p}$ and:

$$s'.\boldsymbol{\ell}[i] = r.\mathit{to} = r^{\Delta}.\mathit{to} \text{ and } s'.\mathbf{nr}_i = \mathbf{d}_i, \text{ for each } i \in I(s, r)$$

Hence, we obtain $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models r^{\Delta}.\varphi$ and $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models \mathsf{Env}^{\Delta}_{\mathrm{EP}}$, for $i \in I(s, r)$, since $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models \widehat{r}^{\Delta}.\varphi$, and $\widehat{r}^{\Delta}.\varphi = r^{\Delta}.\varphi \wedge \mathsf{Env}^{\Delta}_{\mathrm{EP}}$. We repeat this step for each $r \in \mathcal{R}|_{tr}$, and obtain a tuple s', where for every rule $r^{\Delta} \in \mathcal{R}^{\Delta}$, we have that if $tr(r) > 0$, where $r = \mathsf{construct}(r^{\Delta})$, then $tr(r)$ processes are in location $r^{\Delta}.\mathit{from}$ in s', and satisfy the guard $r^{\Delta}.\varphi$.

To show that $\sigma' = \alpha_{\mathbf{p}}(s')$, from the construction of s' we have $tr(r) = |\{i \mid s'.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{to}\}|$, where $r = \mathsf{construct}(r^{\Delta})$. From $\sigma' = g(tr)$ and the definition of the goal of a transition, for every location $\ell \in \mathcal{L}$, we have:

$$\sigma'.\boldsymbol{\kappa}[\ell] = \sum_{r.\mathit{to}=\ell} tr(r) = \sum_{\substack{r.\mathit{to}=\ell \\ r=\mathsf{construct}(r^{\Delta})}} |\{i \mid s'.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{to}\}|$$
$$= |\{i \mid s'.\boldsymbol{\ell}[i] = \ell\}| = \mathsf{counters}_{\mathbf{p}}(s', \ell)$$

Thus, by the definition of the abstraction mapping, we obtain $\sigma' = \alpha_{\mathbf{p}}(s')$.

What remains to show is that the constructed tuple s' satisfies the environment constraint $\mathsf{Env}^{\Delta}_{\mathrm{CP}}$. From $\sigma' \models \mathsf{Env}$ and $\mathsf{Env} \equiv \mathsf{Env}^{\Delta}_{\mathrm{CP}}$, we have $\sigma' \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$. By Lemma 7.1, we get $s' \models \mathsf{Env}^{\Delta}_{\mathrm{CP}}$, and thus $s' \in S(\mathbf{p})$ is a valid state. Further, by the definition of the transition relation $T(\mathbf{p})$, we have that $(s, s') \in T(\mathbf{p})$. Thus $(s', \sigma') \in B(\mathbf{p})$.

Hence, $B(\mathbf{p})$ is a bisimulation relation. □

The existence of a bisimulation relation between the systems $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$, induced by an rSTA and $\mathbf{p} \in \mathbf{P}_{RC}$, and $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ implies that any CTL^* formula over the set CP of c-propositions that holds in the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, also holds in the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ [BK08, Corollary 7.27.]. Thus, as a consequence of Theorem 3.3 we have the following corollary.

Corollary 7.1 (Soundness and completeness)**.** *Let* $\mathsf{STS}(\mathit{rSTA}, \mathbf{p})$ *be the system induced by a given* rSTA *and an arbitrary* $\mathbf{p} \in \mathbf{P}_{RC}$*. Let* $\mathsf{CS}(\mathit{STA}, \mathbf{p})$ *be the counter system induced by the* STA *constructed from* rSTA *and the admissible valuation* $\mathbf{p}$*. Then, for every* CTL^* *formula* ϕ *over the set* CP *of c-propositions, we have that:*

$$\mathsf{STS}(\mathit{rSTA}, \mathbf{p}) \models \phi \quad \textit{iff} \quad \mathsf{CS}(\mathit{STA}, \mathbf{p}) \models \phi$$ □

LTL can be embedded into CTL^* [BK08, Theorem 6.83.]. Thus, we can verify the safety properties of systems $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ induced by an rSTA, using the bounded model checking approach from Chapter 5. In addition, if we obtain a counterexample as output from applying bounded model checking to the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, where STA is constructed from an rSTA, then we can conclude that this counterexample is non-spurious, that is, it can be replayed in the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ induced by the rSTA.

7.3 Experimental Evaluation

To show the usefulness of translating rSTA to STA, we conducted a set of experiments, where we:

- encoded synchronous fault-tolerant distributed algorithms from the literature using rSTA,
- implemented the method from Section 7.1 in a prototype that produces the corresponding STA,
- compared the output to the existing manual encodings, which were artifacts of the experimental evaluation from Chapter 5, and
- verified the properties of the generated STA using the prototype implementing the bounded model checking technique from Chapter 5.

Encoding algorithms with rSTA. To encode synchronous fault-tolerant distributed algorithms as rSTA, we extended the synchronous threshold automata encoding, which we used in the experimental evaluation in Chapter 5, to support declarations of receive variables and constraints about the relationship between the number of sent and received messages, given by the environment assumption. The algorithms we encoded are listed

Table 7.1: The algorithms we encoded as rSTA and the results of applying the verification technique from Chapter 5 to the STA obtained by translating the rSTA. The column QE states the time needed to produce a STA, given a rSTA as input, with Z3 automating the quantifier elimination step. The column ⇒ states if all, some, or none of the automatically generated STA guards imply the guards of the manually produced STA, used as benchmarks in Chapter 5. We report on the time it took the solvers Z3 and CVC4 to (i) check the guard implications (only Z3), (ii) compute the diameter using SMT for the automatically generated STA, and (iii) check the safety properties of the automatically generated STA using the SMT-based procedure from Chapter 5.

algorithm	QE	⇒	(i) ⇒ time	d	(ii) d time		(iii) BMC time	
	Z3		Z3		Z3	CVC4	Z3	CVC4
SAB	0.16s	all	0.18s	2	0.09s	0.26s	0.03s	0.03s
HybridSAB	0.39s	all	0.41s	2	0.14s	0.75s	0.03s	0.06s
OmitSAB	0.34s	all	0.36s	2	0.11s	0.69s	0.03s	0.05s
FairCons	0.25s	all	0.44s	2	0.17s	2.82s	0.07s	0.16s
FloodMin, $k = 1$	0.10s	all	0.19s	2	0.07s	0.25s	0.06s	0.11s
FloodMin, $k = 2$	0.26s	all	0.35s	2	0.13s	1.72s	0.07s	0.19s
FloodMinOmit, $k = 1$	0.10s	all	0.13s	1	0.03s	0.03s	0.01s	0.01s
FloodMinOmit, $k = 2$	0.27s	all	0.26s	1	0.05s	0.08s	0.01s	0.03s
FloodSet	0.20s	all	0.31s	2	0.11s	0.71s	0.07s	0.17s
kSetOmit, $k = 1$	0.59s	all	0.52s	3	2.71s	53.36s	0.22s	0.85s
kSetOmit, $k = 2$	1.43s	all	1.18s	–	t.o.	t.o.	–	–
PhaseKing	1.19s	all	1.57s	4	3.53s	16.51s	0.24s	1.57s
ByzKing	1.16s	all	1.58s	4	1.92s	1min19s	0.27s	1.97s
HybridKing	3.59s	some	3.03s	4	0.33s	6.34s	0.18s	1.11s
OmitKing	3.09s	all	2.79s	4	0.26s	6.12s	0.15s	0.91s
PhaseQueen	0.42s	all	0.90s	3	0.37s	4.46s	0.04s	0.61s
ByzQueen	0.42s	all	0.91s	3	0.39s	17.15s	0.09s	0.58s
HybridQueen	1.34s	some	1.77s	3	0.13s	2.04s	0.05s	0.37s
OmitQueen	1.13s	all	1.56s	3	0.13s	2.18s	0.20s	0.46s

in Table 7.1. For each of them, there already existed a manually produced STA, which we constructed in order to run the experiments in Chapter 5. The manually produced rSTA and STA have the same structure w.r.t. locations and rules, and differ only in the guards that occur on the rules: in the rSTA, we have receive guards, which are Boolean combinations of r-propositions and c-propositions, while in the manually encoded STA, the guards are Boolean combinations of c-propositions.

Applying Quantifier Elimination. We implemented a script that parses the input rSTA and creates a STA whose rules have guards that are Boolean combinations of

c-propositions, according to the abstraction from Section 7.1. To automate the quantifier elimination step, we applied Z3 [dMB08] tactics for quantifier elimination [Bjø10, BJ15], to formulas of the form $\exists \boldsymbol{\delta}\, \hat{r}^{\Delta}.\varphi$, where $\hat{r}^{\Delta}.\varphi \equiv r^{\Delta}.\varphi \wedge \mathsf{Env}^{\Delta}$ is the strengthened guard of the receive guard $r^{\Delta}.\varphi$, for $r^{\Delta} \in \mathcal{R}^{\Delta}$. For all our benchmarks, the STA is generated within seconds, as reported in Table 7.1.

Analyzing the Generated STA. We also performed a comparison between the guards of the automatically generated STA and the manually encoded STA. Syntactically, the automatically generated guards are larger and not human-readable, as they contain additional constraints, obtained by eliminating the receive variables from the environment assumption Env^{Δ}. Semantically, we used Z3 to check whether the guards for the automatically generated STA imply the guards of the manually encoded STA from Chapter 5.

For each automatically generated guard φ_{auto} over c-propositions, we check whether its corresponding guard φ_{man} over c-propositions from the manual encoding is implied by φ_{auto}, for all values of the parameters and number of sent messages. To check this, let $M_i = \mathsf{sent}(m_i)$, for $m_i \in \mathcal{M}$ and $1 \leq i \leq |\mathcal{M}|$, denote the set of locations where processes send messages of type m_i, and observe that both φ_{auto} and φ_{man} are formulas where M_i are free variables, ranging over the set $\mathbb{N}$ of natural numbers. We say that φ_{auto} implies φ_{man} iff the following formula is valid:

$$\forall \mathbf{p} \in \mathbf{P}_{RC}\ \forall M_1 \ldots \forall M_{|\mathcal{M}|}\ \varphi_{\mathsf{auto}}(M_1, \ldots, M_{|\mathcal{M}|}) \rightarrow \varphi_{\mathsf{man}}(M_1, \ldots, M_{|\mathcal{M}|})$$

We automate the validity check of the above formula using an SMT solver, such as Z3, to check the unsatisfiability of its negation:

$$\exists \mathbf{p} \in \mathbf{P}_{RC}\ \exists M_1 \ldots \exists M_{|\mathcal{M}|}\ \varphi_{\mathsf{auto}}(M_1, \ldots, M_{|\mathcal{M}|}) \wedge \neg \varphi_{\mathsf{man}}(M_1, \ldots, M_{|\mathcal{M}|})$$

With this check we are able to either verify that the earlier, manually encoded STA faithfully model the benchmark algorithms, or detect discrepancies, which we investigated further. Our translation technique produces the strongest possible guards, due to our soundness and completeness result. Hence, we expected that the implication holds for all the guards of all the benchmarks we considered. This is however not the case for the algorithms HybridKing and HybridQueen, which are designed to tolerate hybrid faults, in particular, send omissions and Byzantine faults.

For these two algorithms, we found that one automatically generated guard does not imply its corresponding manual guard. By manual inspection, we concluded that this is due to a flaw in the manual encoding. More precisely, the guard in question checks whether the process did not receive any message from the king (resp. queen) in the current phase, i.e., if the king (resp. queen) performed a send omission in the current phase. If this is the case, the process sets the king (resp. queen) value to its own value (resp. to 0). As a Byzantine-faulty process can perform send omissions as well, in the

hybrid fault model, a process may not receive a king (resp. queen) message if the king (resp. queen) is either send-omission or Byzantine-faulty.

In the rSTA for the algorithms HybridKing and HybridQueen, this is captured with the guard $\mathsf{nr}(m_{k0}) = 0 \wedge \mathsf{nr}(m_{k1}) = 0$ and $\mathsf{nr}(m_{q0}) = 0 \wedge \mathsf{nr}(m_{q1}) = 0$, where m_{k0}, m_{k1} and m_{q0}, m_{q1} are the types of the message that the phase king and queen send, respectively. The translated versions of these two guards check whether the number of messages sent by a correct king and queen, respectively, is set to 0, and whether the number of messages sent by a (send-omission or Byzantine) faulty king and queen, respectively, is greater than 0. That is, we have the following two automatically generated guards:

$$\#\mathsf{sent}(m_{k0}) = 0 \wedge \#\mathsf{sent}(m_{k1}) = 0 \wedge \#(\mathsf{sent}(m_{ko0}) \cup \mathsf{sent}(m_{ko1}) \cup \{\ell_F\}) > 0$$
$$\#\mathsf{sent}(m_{q0}) = 0 \wedge \#\mathsf{sent}(m_{q1}) = 0 \wedge \#(\mathsf{sent}(m_{qo0}) \cup \mathsf{sent}(m_{qo1}) \cup \{\ell_F\}) > 0$$

where m_{ko0}, m_{ko1} and m_{qo0}, m_{qo1} are the types of the message that the omission-faulty phase king and queen send, respectively, and ℓ_F is the location where the Byzantine-faulty king and queen perform their broadcast.

However, in the manual encoding of the algorithms HybridKing and HybridQueen, the guards that check whether a message from the king and queen has not been received, respectively, were encoded as follows:

$$\#(\mathsf{sent}(m_{ko0}) \cup \mathsf{sent}(m_{ko1})) > 0$$
$$\#(\mathsf{sent}(m_{qo0}) \cup \mathsf{sent}(m_{qo1})) > 0$$

The two manual guards did not check for the case when the Byzantine-faulty king and queen perform a send omission, respectively, and are thus wrong.

Further, the check whether a Byzantine-faulty king and queen performed as send omission was missing in the manual STA of the algorithms ByzKing and ByzQueen, respectively, which are the variants of HybridKing and HybridQueen, designed to tolerate only Byzantine faults. That is, we identified that there were rules missing from the earlier manual encodings of these algorithms as STA, that should be there. These rules were used to move processes to locations with a default value in phases where the king and queen were Byzantine faulty and performed a send omission. By adding these rules and writing the appropriate manual guards (in both cases, the manual guard is $\{\ell_F\} > 0$), we fixed the manual encodings and were able to prove that every guard of the automatically generated STA implies its corresponding guard in the manually encoded STA.

Model Checking of Safety Properties. The STA we obtained as output of our translation procedure can be given as input to the bounded-model-checking-based approach, presented in Chapter 5. Due to the soundness and completeness of our approach, we are able to solve the parameterized model checking problem of safety properties for algorithms encoded as rSTA by solving its dual parameterized reachability problem for the STA obtained by translating rSTA.

As was the case in Chapter 5, the experiments were run on a machine with 2,8 GHz Quad-Core Intel(R) Core(TM) i7 CPU and 16GB of RAM. The results of applying our SMT-based procedure to the automatically generated STA are presented in Table 7.1. The timeout, denoted by t.o. in the table, was set to 24 hours.

By comparing the experimental results in Table 5.1 on page 154 and Table 7.1, we observe that for the algorithms SAB, HybridSAB, OmitSAB, FairCons, FloodMin, for $k = 1$ and $k = 2$, FloodMinOmit, for $k = 1$ and $k = 2$, kSetOmit, for $k = 2$, FloodSet, PhaseKing, and PhaseQueen, our SMT-based procedure from Chapter 5 performs similarly on both the manually encoded and the automatically generated STA. For the other algorithms, we notice the following differences:

- computing the diameter for the automatically generated STA of kSetOmit, with $k = 1$, is slightly slower with Z3 and slightly faster with CVC4 than computing the diameter for the manually encoded STA;
- Z3 performs better while computing the diameter for the automatically generated STA than for the manually encoded STA of both ByzKing and ByzQueen, while CVC4 performs worse. Note that for the results presented in Table 5.1, the manual encodings of ByzKing and ByzQueen have missing rules. By adding the appropriate rules to the manually encoded STA and running the procedure that computes the diameter, we notice that computing the diameter on the automatically generated STA is still faster with both solvers than on the fixed manual encodings;
- both Z3 and CVC4 are able to compute the diameter for the automatically generated STA of HybridKing and HybridQueen within seconds, in contrast to both solvers running into timeouts when computing the diameter of the manually encoded STA of both algorithms. Further, the computed diameter $\mathsf{d} = 4$ for the automatically generated STA of HybridKing is smaller than the diameter 8, computed for the manually encoded STA of HybridKing on a more powerful machine;
- computing the diameter with Z3 is significantly faster for the automatically generated STA than for the manually encoded STA of OmitKing. CVC4 is able to compute the diameter for the automatically generated STA of OmitKing, while for the manually encoded one it timed out. Further, the computed diameter $\mathsf{d} = 4$ for the automatically generated STA of OmitKing is smaller than the diameter 8, computed for the manually encoded one;
- both Z3 and CVC4 perform better when computing the diameter for the automatically generated STA of OmitQueen, than for the manually encoded one.

For all algorithms, we note that bounded model checking with both Z3 and CVC4 performs similarly for both automatically generated and manually encoded STA.

We attribute the better performance of the technique from Chapter 5 on the automatically generated STA to the fact that the automatically generated guards contain additional

constraints, coming from the environment assumption, which help guide the SMT solvers and do not let them diverge in their search. Moreover, not only do we obtain the diameter bounds faster, we also obtain better bounds for the automatically generated STA of some benchmarks. These findings confirm the conjecture that manual encoding of distributed algorithms is a tedious and error-prone task and suggest that there is a real benefit of producing guards automatically.

7.4 Discussion

In this chapter, we presented an automated technique that abstracts synchronous threshold automata with receive variables to synchronous threshold automata whose guards are Boolean combinations of c-propositions. The abstraction technique translates guards expressed over the receive variables, to guards over the number of sent messages, where no receive variables occur. The translation incorporates the relationship between the sent and received messages in a system operating in a faulty environment, under the synchronous computation model, which we explicitly encoded using the environment assumption Env^{Δ}.

We showed that the translation of rSTA to STA is sound and complete, which implies that the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ and the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ satisfy the same temporal properties. Thus, checking temporal properties of the counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ can be used as a sound and complete procedure for checking temporal properties of the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$. To this end, we used the bounded model checking technique from Chapter 5. In this way, we establish a fully automated pipeline, that for a given algorithm: (1) starts from a formal model that captures its pseudocode, (2) produces a formal model suitable for verification, and (3) automatically verifies its safety properties. Our technique thus closes the gap between the original description of an algorithm and the model of algorithm given as an input to a verification tool.

The abstraction from threshold automata with receive variables to threshold automata with no receive variables presented in this chapter was first introduced in [SKWZ20] for the *asynchronous* case. As asynchronous threshold automata are out of the scope of this thesis, in this chapter, we adapted the approach in [SKWZ20] to the *synchronous* case. We now highlight the main differences between the two approaches.

First, the computation model is different. In the asynchronous computation model, in each transition of a system, only a single process takes a step, while in the synchronous computation model, all processes take a step in a transition. Further, in the asynchronous computation model, there are no limitations on the delivery of messages at the end of a round. Hence, the lower bound on the number of received messages, given in the synchronous model by the number of sent messages by correct processes, is only *eventually* satisfied in the asynchronous model, and thus is not used in the process of eliminating the receive variables from the receive guards. The fact that eventually all messages sent by correct processes are received is used as a fairness constraint for verifying liveness in the asynchronous case, which is something we have not explored in the synchronous case.

Finally, in the experimental evaluation, we have concluded that in the synchronous case, the additional constraints that occur on the translated guards improve the performance of the SMT-based safety verification procedure introduced in Chapter 5. On the contrary, in [SKWZ20], the model checker ByMC [KW18] performed significantly worse on the automatically generated asynchronous threshold automata, than on the manually encoded asynchronous threshold automata from the benchmark repository [Kon]. This is due to the succinctness of the manual guards, which contain less guard propositions than the automatically generated ones. The search space that the model checker ByMC explores is proportional to the number of guard propositions occurring in the threshold automaton. Hence, the search space that ByMC explores for the automatically generated threshold automaton is larger than the search space for the manually encoded one.

CHAPTER 8

Conclusions

In this thesis, we focused on developing automated techniques for parameterized verification of synchronous fault-tolerant distributed algorithms. The algorithms we analyzed were synchronous, in the sense that the processes execute the algorithm in lock-step and take steps simultaneously. Further, in all the algorithms we considered, the processes used broadcast as a communication primitive, where each process sends a message to all other processes. As an assumption of the synchronous computation model, the messages sent in one round are delivered in the same round, that is, a process can use the absence of a message to detect if another process is faulty. Another common characteristic of the algorithms we analyzed is that the processes are indistinguishable – they all have the same local variables and follow the same protocol. In other words, we did not consider algorithms where processes are distinguishable by an identifier. This, together with the broadcast communication, implies that the systems of n processes we are interested verifying are symmetric.

We extensively used symmetry in the approaches presented in this thesis, as it allowed us to simplify the reasoning about the synchronous systems. On the one hand, we used symmetry in the abstraction-based method, where we stored concrete information about the behavior of a small number of processes, and abstracted the behavior of the remaining processes in the system. On the other hand, we used symmetry in the modeling process of algorithms with coordinators, such as, e.g., the algorithm PhaseQueen, using (receive) synchronous threshold automata. There, we abstracted the fact that process i acts as a coordinator in phase i by a constraint in the environment assumption which ensures that there is at most one process in the dedicated coordinator locations.

The techniques we proposed in this thesis were inspired by existing parameterized verification approaches, which were introduced for asynchronous systems. For fault-tolerant distributed algorithms, most of the existing work on parameterized verification focuses on asynchronous algorithms, which operate under interleaving semantics, i.e., where the processes take steps in an arbitrary order and there is no guarantee on message delivery.

Although asynchronous algorithms more closely reflect real-world distributed systems, there are limitations to which kinds of problems can be solved in a purely asynchronous setting (e.g., see [FLP85]). Contrary to asynchronous algorithms, synchronous algorithms have a timing and communication model that is an overapproximation of what happens in the real world, and have been used in real-time systems where timing guarantees are of importance. Still, the simplified timing and communication assumptions do not make parameterized verification of synchronous fault-tolerant distributed algorithms less of a challenge.

Below, we summarize the results of this thesis in the context of the two research challenges we introduced in Chapter 1, namely the formalization and the verification challenge. We also list directions for future work.

8.1 Formalization Challenge

To address the formalization challenge, we proposed three different formal models for synchronous fault-tolerant distributed algorithms. We used them to faithfully model: (i) the process behavior, described using pseudocode, and (ii) the environment in which the processes operate, capturing the fault and communication assumptions.

Process and Environment Variables and Functions. The first formalization that we proposed in Chapter 2 defined the process specification as a set of process variables and functions that update these variables. We defined control and neighborhood process variables, which were used to store information local to a process and information about other processes, respectively. Introducing neighborhood variables as a part of the local state allowed us to model early deciding/stopping consensus algorithms, such as EDAC and ESC, where the processes compare the messages they received in two consecutive rounds. The process functions defined how processes send messages and how they update their variables. The control state update function was parameterized by the number n of processes, the upper bound t on the number of faults, and the round number r. We proposed a finite characterization of this parameterized function using a finite set of guarded assignments. The environment variables that we defined in Chapter 2 are specific for the crash fault model. They were used to non-deterministically flag processes as crashed, and non-deterministically deliver messages from the crashed to the other processes.

The process and environment specifications defined in this way represent a model that is very close to the pseudocode. However, we cannot directly apply a model checker to the parameterized system obtained as a composition of n process specification and an environment specification. To be able to obtain a model checking results, we performed several abstraction steps, which we defined in Chapter 3. These abstraction steps produced neighborhood and environment variables of fixed size and abstracted away the round number, which could grow unboundedly. One key step in producing the abstract model is the definition of the abstract version of the guarded assignments, which is done

syntactically and can be easily automated. As we currently define an abstract system manually, it would be of interest to automate this process, and generate an abstract system automatically, by automatically generating abstract guarded assignments given their concrete counterparts. Another direction for future work is to propose process and environment variables and functions for different fault models. We anticipate that this would mostly involve defining new environment variables, specific to the fault model, while we expect that the process variables for algorithms tolerating other types of faults may remain the same as for the algorithms tolerating crash faults.

Synchronous Threshold Automata. In the synchronous threshold automata modeling framework, we encoded the values of the process control variables using locations. We eliminated the need for process neighborhood variables by recognizing that the number of processes in a given location is equal the number of processes that sent a message, whose message type depends on the value of the control variables in this location. The processes move from one location to another by applying rules, which are guarded using linear arithmetic expressions over the number of processes in a set of locations and the parameters. The non-determinism due to faults is captured by having two rules outgoing of a location that are satisfied at the same time, whose guards check if: (i) only correct processes sent messages, or (ii) both correct and faulty processes sent messages of the same message types. The environment specification of the synchronous threshold automaton is defined by a constraint over the number of processes in given locations, which we called an environment assumption. By defining different environment assumptions and different shapes of the synchronous threshold automata, which depend on the fault model, we were able to model algorithms that tolerate crash, send omission, Byzantine, and hybrid faults.

Synchronous threshold automata are a formal model which we use as an input to our verification procedure, presented in Chapter 5. Since there are no neighborhood variables, we are no longer able to encode early deciding/stopping consensus algorithms using synchronous threshold automata. However, contrary to the process and environment variables and functions, with this approach we are able to capture other kinds of faults in addition to crash faults, and thus more algorithms. Investigating ways in which early deciding/stopping consensus algorithms can be modeled with synchronous threshold automata is an interesting direction for future work.

Receive Synchronous Threshold Automata. The challenge in producing synchronous threshold automata is coming up with the correct guard expressions on the rules, which is a non-trivial task sometimes. To address this, we proposed synchronous threshold automata with receive variables, where we store a receive message counter for each process and each message type. By introducing the receive variables, we ease the process of encoding an algorithm in the synchronous threshold automata framework, as we allowed the guards on the rules of the automaton to contain receive variables, and thus faithfully model the conditions occurring in the pseudocode. In this way, we

produced a formal model of an algorithm that is in a one-to-one correspondence with the pseudocode.

Instead of developing a dedicated parameterized verification technique for systems of n receive threshold automata, we used our existing bounded model checking procedure for systems of n synchronous threshold automata with no receive variables. To do so, we defined an automated translation procedure based on quantifier elimination, that eliminated the receive variables from the guards and environment assumption, and thus produced a synchronous threshold automaton whose guards and environment assumption are expressions that our bounded-model-checking based technique can handle. We thus bridged the gap between a formal model of an algorithm close to its pseudocode and a parameterized verification procedure.

The automated translation procedure gave us results in seconds, and uncovered several flaws in the existing synchronous threshold automata, which were encoded manually. We identified these flaws by comparing the manual encodings to the output of the translation procedure. Further, we noticed that bounded model checking performed better on the automatically generated synchronous threshold automata than on the manually encoded ones. As the difficulty of producing synchronous threshold automata has been lifted by introducing receive variables, in the future, it would be interesting to extend the set of benchmarks that we currently have. Adding new types of algorithms may introduce the need for extending the synchronous threshold automata with new features, and defining new automated translation procedures, which we leave for future work.

8.2 Verification Challenge

To address the verification challenge, in the spirit of existing parameterized verification approaches, we proposed: (i) a sound, but incomplete technique, based on abstraction, and (ii) a sound and complete technique, tailored to a specific class of problems, i.e., in our case, to a class of safety properties, which can be verified by showing that a bad state is not reachable in any execution.

Abstraction. In Chapter 3, we proposed an abstraction technique that allowed us to obtain first parameterized verification results for synchronous fault-tolerant distributed algorithms. The abstraction method we proposed is a sound, but an incomplete method. We used it to verify both safety and liveness properties of our benchmarks EDAC, ESC, FairCons, FloodMin, FloodSet, and NBAC. We introduced domain-specific pattern-based verification conditions which in fact are used as fairness conditions for verifying liveness. These conditions were needed in order to filter out traces leading to spurious counterexamples in the constructed abstract system. Using symmetry, we kept the values of the variables of a small number of processes in an abstract state to be equal to those in some corresponding concrete state. The remaining processes were identified based on the values of their control variables, and the abstract state stored whether there are zero or many processes in some control state. We thus fixed the size of the array variables

in the abstract state, which in a concrete global state depends on the parameter n. By applying an abstraction mapping that we proposed, any concrete system of any size can be mapped to an abstract system, defined as an overapproximation.

We also defined an abstract system constructively, by defining abstract versions of the system variables, global states, and transition relation. The main challenge in constructing the abstract transition relation was defining abstract versions of the guarded assignments and the update of the control states of the abstracted processes. By showing that the constructed abstract system simulates the abstract system obtained as an overapproximation induced by the abstraction mapping, we were able to verify properties of any concrete system by verifying properties of the constructed abstract system.

We used the constructive definition of the abstract system in order to conduct our experimental evaluation, where we encoded the abstract system using TLA+ and used the explicit-state model checker TLC to verify the properties, which took days to verify our benchmarks. In order to experiment with symbolic model checkers, one may consider running Apalache [KKT19], which is a new symbolic model checker for TLA+. Another option would be to encode the abstract system in another specification language for which a symbolic model checker already exists.

Regarding the abstraction technique itself, there are two open questions that may be investigated in the future. First, as the abstraction technique is tailored to the crash fault model, it would be interesting to investigate which adjustments should be done in order to capture send omission or Byzantine faults. Second, as we currently identify the pattern-based verification conditions manually, it would be beneficial to find means to generate them automatically. Moreover, identifying verification conditions for environments that model other kinds of faults is another possible direction for future work.

Bounded Model Checking. In Chapter 5, we proposed a procedure for parameterized verification of safety properties of systems of n synchronous threshold automata, based on bounded model checking. We proved that the parameterized reachability problem for these systems is in general undecidable, by a reduction from the halting problem of two counter machines. However, for systems for which we can compute a bound on the diameter, we are able to use it as a completeness threshold for bounded model checking. Due to the undecidability result, this bound does not always exist. We proposed a semi-decision procedure which we used to compute the diameter of our benchmarks, and which revealed that the diameters of synchronous threshold automata that model the synchronous fault-tolerant distributed algorithms we considered in this thesis are small. This makes the bounded model checking approach both complete and efficient. We also provided theoretical guarantees for a class of algorithms, whose synchronous threshold automata satisfy certain conditions. Most importantly, we showed that the bound on the diameter does not depend on the values of the parameters, but rather on the characteristics of the synchronous threshold automata.

We used this technique to verify the safety properties of our benchmarks FairCons, FloodMin for $k = 1$ and $k = 2$, FloodMinOmit for $k = 1$ and $k = 2$, FloodSet, kSetOmit

for $k = 1$, PhaseKing, HybridKing and its variants ByzKing and OmitKing, PhaseQueen, HybridQueen and its variants ByzQueen and OmitQueen, SAB, HybridSAB and its variant OmitSAB. We ran experiments where the synchronous threshold automaton given as input was either manually encoded, or was produced automatically from a receive threshold automaton using the translation procedure defined in Chapter 7. To automatically solve the decision problems, we used back-end SMT solvers, in particular the solvers Z3 and CVC4.

While we are currently only able to check a class of safety properties, we argue that this is enough for most synchronous fault-tolerant distributed algorithms, as (i) safety properties of the algorithms we considered in this thesis fall in this class, and (ii) liveness properties are termination properties. For algorithms where this is not the case, one would have to determine completeness thresholds for general safety and liveness properties, which is something we leave for future work. Another direction for future work is to provide theoretical guarantees for the existence of a bound on the diameter for algorithms whose synchronous threshold automata do not fall into the class of automata for which we currently guarantee that a bound exists.

Bibliography

[ACJT96] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General Decidability Theorems for Infinite-State Systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 313–321. IEEE Computer Society, 1996.

[AGOP16] Francesco Alberti, Silvio Ghilardi, Andrea Orsini, and Elena Pagani. Counter Abstractions in Model Checking of Distributed Broadcast Algorithms: Some Case Studies. In *Proceedings of the 31st Italian Conference on Computational Logic, Milano, Italy, June 20-22, 2016*, volume 1645 of *CEUR Workshop Proceedings*, pages 102–117. CEUR-WS.org, 2016.

[AK86] Krzysztof R. Apt and Dexter Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[ARS+18] Benjamin Aminof, Sasha Rubin, Ilina Stoilkovska, Josef Widder, and Florian Zuleger. Parameterized Model Checking of Synchronous Distributed Algorithms by Abstraction. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2018.

[ARZS15] Benjamin Aminof, Sasha Rubin, Florian Zuleger, and Francesco Spegni. Liveness of Parameterized Timed Networks. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 375–387. Springer, 2015.

[AW04] Hagit Attiya and Jennifer L. Welch. *Distributed Computing - Fundamentals, Simulations, and Advanced Topics (2. ed.)*. Wiley Series on Parallel and Distributed Computing. Wiley, 2004.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS*

'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCG89] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Reasoning about Networks with Many Identical Finite State Processes. *Inf. Comput.*, 81(1):13–31, 1989.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10ˆ20 States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BEJQ18] Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 372–391. Springer, 2018.

[BEL20] A. R. Balasubramanian, Javier Esparza, and Marijana Lazić. Complexity of Verification and Synthesis of Threshold Automata. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, volume 12302 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2020.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BGP] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Asymptotically Optimal Distributed Consensus. Technical report, Bell Labs. `http://plan9.bell-labs.co/who/garay/asopt.ps`.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards Optimal Distributed Consensus (Extended Abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 410–415. IEEE Computer Society, 1989.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJ15] Nikolaj Bjørner and Mikolás Janota. Playing with Quantified Satisfaction. In *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015.

[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.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular Model Checking. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.

[Bjø10] Nikolaj Bjørner. Linear Quantifier Elimination as an Abstract Decision Procedure. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2010.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BK14] Peter Bailis and Kyle Kingsbury. The Network is Reliable. *Commun. ACM*, 57(9):48–55, 2014.

[BKLW19] Nathalie Bertrand, Igor Konnov, Marijana Lazic, and Josef Widder. Verification of Randomized Consensus Algorithms Under Round-Rigid Adversaries. In *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*, volume 140 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[BLP06] Sébastien Bardin, Jérôme Leroux, and Gérald Point. FAST Extended Release. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 63–66. Springer, 2006.

[BSW11] Martin Biely, Ulrich Schmid, and Bettina Weiss. Synchronous Consensus Under Hybrid Process and Link Failures. *Theor. Comput. Sci.*, 412(40):5602–5630, 2011.

[Buc16] Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. Master's thesis, University of Guelph, 2016. `http://hdl.handle.net/10214/9769`.

[Bur06] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350. USENIX Association, 2006.

[BvGKJ17] Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. *Proc. ACM Program. Lang.*, 1(OOPSLA):110:1–110:27, 2017.

[BW20] A. R. Balasubramanian and Igor Walukiewicz. Characterizing Consensus in the Heard-Of Model. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CCM09] Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In *Reachability Problems, 3rd International Workshop, RP 2009, Palaiseau, France, September 23-25, 2009. Proceedings*, volume 5797 of *Lecture Notes in Computer Science*, pages 93–106. Springer, 2009.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. The TLA$^+$ Proof System: Building a Heterogeneous Verification Platform. In *Theoretical Aspects of Computing - ICTAC 2010, 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings*, volume 6255 of *Lecture Notes in Computer Science*, page 44. Springer, 2010.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[CHLT00] Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight Bounds for k-set Agreement. *J. ACM*, 47(5):912–943, 2000.

[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 *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[CLM89] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional Model Checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 353–362. IEEE Computer Society, 1989.

[CMP04] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*, pages 382–398. Springer, 2004.

[Coo72] David C Cooper. Theorem Proving in Arithmetic Without Multiplication. *Machine Intelligence*, 7(91-99):300, 1972.

[CS04] Bernadette Charron-Bost and André Schiper. Uniform Consensus is Harder than Consensus. *J. Algorithms*, 51(1):15–37, 2004.

[CS09] Bernadette Charron-Bost and André Schiper. The Heard-Of Model: Computing in Distributed Systems with Benign Faults. *Distributed Comput.*, 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 M. Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. Verification by Network Decomposition. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2004.

[CTV06] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Environment Abstraction for Parameterized Verification. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2006.

[CTV08] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008.

[DDMW19] Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. Communication-Closed Asynchronous Protocols. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 344–363. Springer, 2019.

[DHSZ03] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine Fault Tolerance, from Theory to Reality. In *Computer Safety, Reliability, and Security, 22nd International Conference, SAFECOMP 2003, Edinburgh, UK, September 23-26, 2003, Proceedings*, volume 2788 of *Lecture Notes in Computer Science*, pages 235–248. Springer, 2003.

[DHV+14] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A Logic-Based Framework for Verifying Consensus Algorithms. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2014.

[DHZ16] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 400–415. ACM, 2016.

[DM12] Henri Debrat and Stephan Merz. Verifying Fault-Tolerant Distributed Algorithms in the Heard-Of Model. *Arch. Formal Proofs*, 2012, 2012.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dt04] The Coq development team. *The Coq Proof Assistant Reference Manual*, 2004. Version 8.0.

[DTT14] Giorgio Delzanno, Michele Tatarek, and Riccardo Traverso. Model Checking Paxos in Spin. In *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014*, volume 161 of *EPTCS*, pages 131–146, 2014.

[EF82] Tzilla Elrad and Nissim Francez. Decomposition of Distributed Programs into Communication-Closed Layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.

[EN95] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about Rings. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 85–94. ACM Press, 1995.

[EN96] E. Allen Emerson and Kedar S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems (Extended Abstract). In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 1996.

[EN03] E. Allen Emerson and Kedar S. Namjoshi. On Reasoning About Rings. *Int. J. Found. Comput. Sci.*, 14(4):527–550, 2003.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and Model Checking. *Formal Methods Syst. Des.*, 9(1/2):105–131, 1996.

[FKL08] Dana Fisman, Orna Kupferman, and Yoad Lustig. On Verifying Fault Tolerance of Distributed Protocols. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2008.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[FZWK17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 328–343. ACM, 2017.

[GL03] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Comput.*, 16(1):1–20, 2003.

[God90] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.

[GREP20] Zeinab Ganjei, Ahmed Rezine, Petru Eles, and Zebo Peng. Verifying Safety of Parameterized Heard-Of Algorithms. In *Networked Systems - 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3-5, 2020, Proceedings*, volume 12129 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2020.

[GS97] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[HHK+17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM*, 60(7):83–92, 2017.

[JKS+13] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 201–209. IEEE, 2013.

[Jr.78] Sheldon B. Akers Jr. Binary Decision Diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[JRS11] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 245–256. IEEE Compute Society, 2011.

[KEH+20] Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive Sequentialization of Asynchronous Programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 227–242. ACM, 2020.

[KKT19] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.

[KKW18] Jure Kukovec, Igor Konnov, and Josef Widder. Reachability in Parameterized Systems: All Flavors of Threshold Automata. In *29th International*

Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China, volume 118 of *LIPIcs*, pages 19:1–19:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[KLVW17] Igor 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. ACM, 2017.

[Kon] Igor Konnov. Fault-Tolerant Benchmarks. Accessed: August, 2020.

[KP00] Yonit Kesten and Amir Pnueli. Control and Data Abstraction: The Cornerstones of Practical Formal Verification. *Int. J. Softw. Tools Technol. Transf.*, 2(4):328–342, 2000.

[KQH18] Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Synchronizing the Asynchronous. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[Krs05] Sava Krstić. Parametrized System Verification with Guard Strengthening and Parameter Abstraction. In *Automated Verification of Infinite-State Systems*, 2005.

[KS03] Daniel Kroening and Ofer Strichman. Efficient Computation of Recurrence Diameters. In *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.

[KVW14] Igor Konnov, Helmut Veith, and Josef Widder. On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, volume 8704 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2014.

[KVW15] Igor Konnov, Helmut Veith, and Josef Widder. SMT and POR Beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 85–102. Springer, 2015.

[KVW17] Igor V. Konnov, Helmut Veith, and Josef Widder. On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability. *Inf. Comput.*, 252:95–109, 2017.

[KW18] Igor Konnov and Josef Widder. ByMC: Byzantine Model Checker. In *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III*, volume 11246 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2018.

[Lam98] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lam11] Leslie Lamport. Byzantizing Paxos by Refinement. In *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2011.

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-Value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 357–370. ACM, 2016.

[Lei10] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[Lip75] Richard J. Lipton. Reduction: A New Method of Proving Properties of Systems of Processes. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, USA, January 1975*, pages 78–86. ACM Press, 1975.

[LKWB17] Marijana Lazić, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of Distributed Algorithms with Parameterized Threshold Guards. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 32:1–32:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[LR93] Patrick Lincoln and John M. Rushby. The Formal Verification of an Algorithm for Interactive Consistency under a Hybrid Fault Model. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece,*

June 28 - July 1, 1993, Proceedings, volume 697 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 1993.

[LS05] Jérôme Leroux and Grégoire Sutre. Flat Counter Automata Almost Everywhere! In *Automated Technology for Verification and Analysis, Third International Symposium, ATVA 2005, Taipei, Taiwan, October 4-7, 2005, Proceedings*, volume 3707 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2005.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MAK13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013.

[McM99] Kenneth L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 1999.

[McM01] Kenneth L. McMillan. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings*, volume 2144 of *Lecture Notes in Computer Science*, pages 179–195. Springer, 2001.

[McM03] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[MP20] Kenneth L. McMillan and Oded Padon. Ivy: A Multi-modal Verification Tool for Distributed Algorithms. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 190–202. Springer, 2020.

[MSB17] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff Bounds for Consensus Algorithms. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 217–237. Springer, 2017.

[Nak08] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. `https://bitcoin.org/bitcoin.pdf`.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NTK12] Tatsuya Noguchi, Tatsuhiro Tsuchiya, and Tohru Kikuno. Safety Verification of Asynchronous Consensus Algorithms with Model Checking. In *IEEE 18th Pacific Rim International Symposium on Dependable Computing, PRDC 2012, Niigata, Japan, November 18-19, 2012*, pages 80–88. IEEE Computer Society, 2012.

[OO14] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[OTT09] John W. O'Leary, Murali Talupur, and Mark R. Tuttle. Protocol Verification Using Flows: An Industrial Experience. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 172–179. IEEE, 2009.

[Pel93] Doron A. Peled. All from One, One for All: on Model Checking Using Representatives. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[PMP+16] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630. ACM, 2016.

[Pre29] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes Rendus du I congres de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980.

[Pug92] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Commun. ACM*, 35(8):102–114, 1992.

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-Counter Abstraction. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2002.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[Ray10] Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[RKK17] Andrew Reynolds, Tim King, and Viktor Kuncak. Solving Quantified Linear Arithmetic by Counterexample-Guided Instantiation. *Formal Methods Syst. Des.*, 51(3):500–532, 2017.

[Ser11] Amazon Web Services. Summary of the Amazon EC2 and Amazon RDS service disruption in the US East region. `https://aws.amazon.com/message/65648/`, April 2011.

[Ske81] Dale Skeen. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, USA, April 29 - May 1, 1981*, pages 133–142. ACM Press, 1981.

[SKWZ19] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, volume 11428 of *Lecture Notes in Computer Science*, pages 357–374. Springer, 2019.

[SKWZ20] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. Eliminating Message Counters in Threshold Automata. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, volume 12302 of *Lecture Notes in Computer Science*, pages 196–212. Springer, 2020.

[SKWZ21] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. Eliminating Message Counters in Synchronous Threshold Automata. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 2021.

[SRSP04] Wilfried Steiner, John M. Rushby, Maria Sorea, and Holger Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 189–198. IEEE Computer Society, 2004.

[ST87] T. K. Srikanth and Sam Toueg. Simulating Authenticated Broadcasts to Derive Simple Fault-Tolerant Algorithms. *Distributed Comput.*, 2(2):80–94, 1987.

[Stoa] Ilina Stoilkovska. Synchronous Threshold Automata. `https://github.com/istoilkovska/syncTA`. Encodings of synchronous fault-tolerant distributed algorithms as synchronous threshold automata. [Online; accessed February 2021].

[Stob] Ilina Stoilkovska. Synchronous TLA+ Benchmarks. `https://github.com/istoilkovska/synchronous-tla-benchmarks`. Synchronous fault-tolerant distributed algorithms encoded in TLA+. [Online; accessed February 2021].

[Suz88] Ichiro Suzuki. Proving Properties of a Ring of Finite-State Machines. *Inf. Process. Lett.*, 28(4):213–214, 1988.

[SWR02] Ulrich Schmid, Bettina Weiss, and John M. Rushby. Formally Verified Byzantine Agreement in Presence of Link Faults. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria, July 2-5, 2002*, pages 608–616. IEEE Computer Society, 2002.

[TKW20] Thanh-Hai Tran, Igor Konnov, and Josef Widder. Cutoffs for Symmetric Point-to-Point Distributed Algorithms. In *Networked Systems - 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3-5, 2020, Proceedings*, volume 12129 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2020.

[TLA] TLA+ Toolbox. http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html.

[TS11] Tatsuhiro Tsuchiya and André Schiper. Verification of Consensus Algorithms Using Satisfiability Solving. *Distributed Comput.*, 23(5-6):341–358, 2011.

[Val89] Antti Valmari. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[vGKB+19] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend Synchrony: Synchronous Verification of Asynchronous Distributed Programs. *Proc. ACM Program. Lang.*, 3(POPL):59:1–59:30, 2019.

[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 *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.