Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Parameterized Model Checking of Fault-Tolerant Distributed Algorithms

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

by

Annu Gmeiner MU
Registration Number 0927448

to the Faculty of Informatics
at the Vienna University of Technology

Advisors: Univ.-Prof. Dipl.-Ing. Dr. Helmut Veith
Privatdoz. Dipl.-Ing. Dr. Josef Widder

The dissertation has been reviewed by:

Prof. Helmut Veith   Prof. Natasha Sharygina

Vienna, 5th October, 2015

Annu Gmeiner

Erklärung zur Verfassung der Arbeit

Annu Gmeiner MU
Antonie-Alt-Gasse 8/2/10, Vienna, 1100-Austria

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, 5 Oktober, 2015

Annu Gmeiner

Acknowledgements

First of all I would like to thank Prof. Helmut Veith for accepting me as his PhD student and for being such a great supervisor. Without his constant guidance, support and motivation, I would not have been able to finish this thesis. I thank him for being extremely patient and understanding while I had to take my time with the thesis due to personal reasons. He always remained approachable in spite of his busy schedule and was happy to offer help and guidance in both professional and non-professional matters.

I would now like to thank Prof. Ulrich Schmid for introducing me to the area of distributed algorithms through one of the best lectures I have attended. He introduced me to my current research area. I thank him for the guidance he offered me and the numerous discussions about various topics in distributed algorithms. I would like to thank Prof. Hannes Werthner for founding the PhD School of Informatics and funding me through it for the first two years of my PhD career. I also thank him for organizing great lectures and seminars as a part of the PhD School, which introduced me to the research area I was interested in.

I thank Prof. Thomas Eiter and Stefan Woltran for funding me through the Doctoral School of Mathematical Logic in Computer Science and the constant guidance with the various bureaucratic issues of the PhD program.

I would also like to thank Beatrix Forsthuber and Eva Nedoma for helping with the various administrative issues at the institute and Clarissa Schmid for providing assistance with administrative matters of the PhD School of Informatics.

I would now like to thank my FORSYTE colleagues, especially Igor Konnov and Josef Widder, who I worked together with in this project. I thank Josef for agreeing to be my second supervisor. Both Josef and Igor have taken a lot of time and effort to help me with finishing my thesis. They have constantly and patiently answered all my possibly silly questions. They have supported me by closely working together with me when I was under extreme time pressure to finish my thesis. Without their prolonged support it would have been impossible for me to finish writing this thesis. I also thank them for making the whole task a bit lighter by creating a nice atmosphere at work with their funny sense of humour and encouragement while I was weighed down by the circumstances.

I thank Julia Demyanova for being such a great office mate and friend. I would like to also thank all the other FORSYTE colleagues for the very constructive feedback to my practice talk. I also thank my PhD School and MLICS doctoral college colleagues for the support and the great discussions that we have had.

Next, I thank Mark Guttenbrunner for the support and help during the first two years of the PhD School without which I would have been able to be at this stage. I would like to thank Eugen Jiresch for all the logic lectures he took me to, in the beginning of the PhD school, not to forget the real-time translation to English. I thank Christoph Roschger for the interesting discussions over the many lunches and cups of coffee. I also would like to thank other friends who made my academic life a lot better and funnier, like Markus Huber, Barbara Donko, Stefan Ruemmele and Worarath Krathu.

I would like to thank Andreas Holzer for all the great technical and non-technical discussions we have had during my time at FORSYTE. I thank him for being the fine person that he is, for taking time to explaining all the silly questions that I had about the different papers I read. I would like to thank Florian Zuleger for being such a great friend, for being the perfect best man, for the time and patience for the numerous visits to choose the nicest wedding dress, for all the concerts that we have been to, for all the support through my agonies of writing and finishing the thesis. I thank both Andi and Florian for making the first conference that I attended such a memorable event.

Last but the most important, I would like to thank my family for tolerating the many frequent lows and highs I had through out my PhD career. I would like to thank my parents for continuously inspiring me to finish what I started and all the emotional support they managed to shower on me in spite of being physically so far away from me.

I would like to thank Karl Gmeiner without whose support I would not have been able to manage writing this thesis while being on maternity leave. First of all, I thank him for making our PhD school period so much fun, for the numerous discussions we had about almost everything, and for being such a great friend and husband. I thank Karl for the all the emotional support and for being there whenever I needed him, for managing a job and a family single-handedly while I was busy with my thesis, for standing me through the extreme lows, for motivating me endlessly, for always being successful in showing me the positive side of everything. Thank you for such a beautiful family that we have.

I would also like to thank my darling daughter Sharon, for all the love and patience she has showed while the thesis took all the time that I should have had for her.

Lastly, I would like to thank my son Arthur, who is way too young to remember any of these.

Abstract

Distributed algorithms play an inevitable role in our day to day life. They are omnipresent, with their applications extending to our household equipments, medical equipments, air traffic control, space missions, Internet and such numerous fields of application. A mere peek at their application areas reveals the importance of their reliable operation even in the presence of faults in the underlying system. Fault-Tolerant Distributed Algorithms (FTDAs) were introduced to increase the reliability of distributed algorithms. FTDAs are designed in a way that the system functions correctly even in the presence of a certain number of faults in the system. However, in order to fulfill the purpose of an FTDA, it is necessary to ensure that no man made errors have crept into its design. In other words, it needs to be verified that the FTDA satisfies its specifications.

The common practice in the distributed algorithms community is to express such algorithms using pseudo code backed by no formal semantics and prove their correctness by mathematical reasoning in natural language. The semantics of the algorithm and the assumptions made on the environment are usually not clear from the pseudo code and expert knowledge of the algorithm is a necessity to understand and reason about it. Moreover, the non-determinism introduced by concurrency, varying processor speeds and message delays, a limited view of local states etc., make manual correctness proof of such algorithms an extremely creative, time consuming and error prone task. The inclusion of fault tolerance introduces an additional dimension of non-determinism due the presence of faults. Since manual correctness proofs can be erroneous, verifying these algorithms formally is the only way to ensure their correctness. Automatic verification eliminates the need for in depth knowledge about the algorithm to be verified. Our aim is to automatically verify a widely used class of FTDAs, where the processes send messages to each other and take actions when the number of messages received crosses a certain threshold. We call this class of algorithms, threshold-based FTDAs.

The first step towards formally verifying an algorithm is formalizing it. Thus, at first we introduce a formal framework which captures the behavior of threshold-based FTDAs. We introduce extended Control Flow Automata (CFA) for this purpose. The extended CFA serves as a simple mathematical representation of the algorithm. We use the SPIN model checker to verify the algorithms. To this end, we extend PROMELA, which is the input language of SPIN in order to capture the various features of threshold-based FTDAs, such as non-determinism, multiple parameters etc., and model these algorithms

using extended PROMELA.

There are various formal verification techniques in existence. We choose model checking because model checking provides an automated technique of verification with minimum user intervention. FTDAs are parameterized in the number of correct process as well as the maximum number of faulty processes. Moreover, the actual number of faulty processes in an FTDA can take any value less than or equal to the maximum number of faulty processes. Thus the system to be verified has multiple parameters. That is, in order to ensure the correctness of an FTDA we have to consider all the possible values these parameters can take. This gives rise to an infinite family of systems with unbounded states to be verified and thus we have a Parameterized Model Checking Problem (PMCP) at hand. We prove that our problem is undecidable and thus there is no general solution to it.

We introduce an abstraction method called Parameterized Interval Abstraction (PIA) to reduce our family of unbounded systems to a single finite state system. To this end, we apply the PIA to the family of unbounded-state systems in two levels. We first get rid of the unbounded variables from the individual systems of the family by mapping them to their corresponding abstract values of a finite domain. We then eliminate the parameters from the resultant system by using PIA on the number processes in each location. As a result we get a single finite state system, which can be verified using a conventional model checker like SPIN. We use refinement techniques to eliminate spurious counter examples caused by the undesired behavior in the resultant finite state system introduced by the abstraction. To validate our formalization and abstraction method, we present several case studies where safety and liveness properties of different threshold-based FTDAs are verified. We also present simulation proofs for both levels of the abstraction to show that our abstraction method is sound.

Abstract

Verteilte Algorithmen spielen eine wichtige Rolle in unserem täglichen Leben. Sie sind allgegenwärtig, mit Anwendungen, die von Haushaltsgegenständen, medizinische Geräte, Flugverkehrskontrolle, Weltraummissionen bis zum Internet und noch weiter reichen. Schon ein kleiner Blick auf die Anwendungsfelder zeigt, wie wichtig die verlässliche Funktionalität auch im Fehlerfall des grundlegenden Systems ist. Fehlertolerante verteilte Algorithmen (FTDAs) wurden eingeführt, um die Zuverlässigkeit verteilter Algorithmen zu erhöhen. FTDAs wurden so entwickelt, dass das System bis zu einer bestimmten Anzahl von Fehlern korrekt funktioniert. Nichtsdestotrotz ist es nötig, sicherzustellen, dass bei der Entwicklung des FDTAs Fehler vermieden werden. Mit anderen Worten, man muss verifizieren, dass der FTDA seine Spezifikation erfüllt.

In der gängigen Literatur werden solche Algorithmen meist informell in Pseudo-Code definiert und ihre Korrektheit wird durch mathematisches Schließen in natürlicher Sprache bewiesen. Die Semantik des Algorithmus und die Annahmen über die Umgebung sind im Allgemeinen nicht aus dem Pseudo-Code erkennbar sodass die Analyse der Algorithmen fundiertes Wissen über dieselben erfordert. Darüber hinaus sind informelle Korrektheitsbeweise solcher Algorithmen komplex, zeitaufwändig und fehleranfällig wegen des Nicht-Determinismus, der sich aus Nebenläufigkeit, unterschiedliche Prozessorgeschwindigkeiten usw. ergibt. Durch das Einbeziehen von Fehlertoleranz muss auch das nicht-deterministische Auftreten von Fehlern berücksichtigt werden. Da solche händischen Korrektheitsbeweise fehlerbehaftet sein können ist die formale Verifizierung solcher Algorithmen die einzige Möglichkeit um auszuschließen, dass bei der Entwicklung Fehler eingebaut wurden. Automatische Verifizierung beseitigt die Notwendigkeit von tiefergehendem Wissen über den Algorithmus. Unser Ziel ist, sogenannte threshold-based FTDAs automatisch zu verifizieren. In dieser Klasse von FTDAs tauschen Prozesse Nachrichten miteinander aus und führen bestimmte Aktionen aus, nachdem die Zahl der eingehenden Nachrichten einen Schwellenwert überschritten hat.

Der erste Schritt um einen Algorithmus formal zu verifizieren ist, diesen selbst zu formalisieren. Aus diesem Grund führen wir ein formales Framework ein, das das Verhalten von threshold-based FTDAs abbildet. Zu diesem Zweck definieren wir erweitere Kontrollfluss-Automaten (CFAs). Diese erweiterten CFAs stellen eine einfache mathematische Repräsentation des Algorithmus dar. Wir benutzen den Spin Model Checker um den Algorithmus zu verifizieren. Dazu erweitern wir Promela, die Eingabesprache von Spin um die unterschiedlichen Eigenschaften von threshold-based FTDAs abzubilden wie Nicht-Determinismus, mehrere Parameter usw. und modellieren die Algorithmen im erweiteren PROMELA.

Es gibt unterschiedliche Möglichkeiten zur formalen Verifizierung. Wir nutzen Model-Checking weil es automatisiert mit minimaler Benutzerinteraktion ablaufen kann. FTDAs sind parametrisiert sowohl bei der Anzahl der Prozesse als auch bei der maximalen Zahl an fehlerhaften Prozessen. Darüber hinaus kann die tatsächliche Zahl an fehlerhaften Prozessen in einem FTDA einen beliebigen Wert kleiner oder gleich der maximalen Anzahl an fehlerhaften Prozessen annehmen. Daher muss das System mit mehreren unterschiedlichen Parametern verifiziert werden. Um die Korrektheit eines FTDAs sicherzustellen müssen wir alle möglichen Werte dieser Parameter berücksichtigen. Das führt zu einer unendlich großen Familie von System mit unbegrenzt vielen Zuständen, die verifiziert werden müssen und somit erhalten wir ein parametrisiertes Model-Checking-Problem (PMCP). Wir beweisen, dass unser Problem unentschiedbar ist und somit keine allgemeine Lösung existiert.

Wir führen eine Abstraktionsmethode ein, Parameterized Interval Abstraction (PIA), um diese Familie an unbegrenzten Systemen zu einem einzige Finite-State-System zu reduzieren. Dazu wenden wir PIA auf zwei Ebenen auf diese Systeme an. Zunächst entfernen wir ungebundene Variablen von den einzelnen Systemen, indem wir sie auf ihrem entsprechenden abstrakten Wert in einem endlichen Bereich abbilden. Danach eliminieren wir die Parameter des sich daraus ergebenden Systems indem wir PIA auf die Zahlen an Prozessen an jeder Stelle anwenden. Als Ergebnis erhalten wir ein einziges Finite-State-System, das mit einem gewöhnlichen Model-Checker wie SPIN verifiziert werden kann. Wir benutzen Verfeinerungstechniken um falsche Gegenbeispiele zu eliminieren, die durch ungewolltes Verhalten im Endsystem verursacht werden, das durch die Abstraktion eingeführt wurde. Um unsere Formalisierung und Abstraktionsmethode zu bestätigen zeigen wir zahlreiche Fallstudien, bei denen Safety- und Lifeness-Eigenschaften unterschiedlicher FTDAs verifiziert werden. Wir präsentieren zusätzlich Simulationsbeweise für beide Ebenen der Abstraktion um zu zeigen, dass unsere Abstraktionsmethode gültig ist.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Aim

The aim of this thesis is to develop a novel method to enable automated verification of an important class of Fault-Tolerant Distributed Algorithms (FTDAs), called threshold-based FTDAs and thus make a substantial improvement to the current state of the art in the area of formal verification of fault-tolerant distributed algorithms (FTDAs).

FTDAs are in essence distributed algorithms designed to ensure correctness of distributed systems even in the presence of a certain number of faulty processes, thereby increasing the reliability of distributed systems.

We concentrate on an important and widely used class of fault-tolerant distributed algorithms, that we name threshold-based fault-tolerant distributed algorithms. Threshold-based FTDAs uses threshold-guards on the number of messages received by the processes to decide action to be taken by the corresponding process [16, 52, 77, 71, 8, 69, 37, 19, 16]. They are used in many safety critical applications [64]. The algorithms we consider have multiple parameters expressing the number of processes and faults. That is, we have to verify a family of systems with all the possible values the parameters can take.

1.2 Motivation

To quote Leslie Lamport, "A distributed system is one in which the failure of a computer which you did not even know existed can render your own computer unusable" [65]. A distributed system consists of a collection of loosely coupled processes, that coordinate with each other to achieve a common goal [8]. The Internet is one of the most well-known examples of a distributed system. A distributed algorithm defines a protocol for the behavior of each process in such a system. Distributed algorithms have been studied extensively [71, 8], and the central problems are well understood.

The problems in distributed algorithms differ from the fundamental problems in sequential (that is, non-distributed) algorithms. The central problems in distributed algorithms are posed by the inevitable uncertainty of any local view of the global state, originating from unknown/varying processor speeds, communication delays, and failures. Pivotal services in distributed systems, such as mutual exclusion, routing, consensus, clock synchronization, leader election, atomic broadcasting, and replicated state machines, must hence be designed to cope with this uncertainty. Moreover, it is a common practice to represent distributed algorithms using a pseudo language. Thus, they are difficult to reason about and understand, when compared to non-distributed algorithms.

1.2.1 Classification of distributed algorithms

We can classify distributed algorithms into different types based on the following attributes:

Communication method: The interprocess communication models are

- Shared memory model, where the processes communicate with each other by changing the value of shared variables.
- Message passing model, where the processes communicate by sending messages to each other.

Timing model: Based on the timing model, distributed algorithms can be classified into

- Synchronous Algorithms: In the *synchronous model*, the processes are perfectly synchronized. That is, conceptually they take steps in perfect synchrony with each other. This model is simple but expensive to achieve in reality [71].
- Asynchronous Algorithms: In an *asynchronous* algorithm there is no upper bound to the message delay and relative process speeds [46]. Process steps can be arbitrarily interleaved. These features of asynchronous algorithms gives rise to non-determinism.
- Partially Synchronous Algorithms: *Partially synchronous* algorithms are more restricted than asynchronous algorithms. Usually, an upper bound on the relative process speeds and/or on the message delay is used to restrict the behavior of the processes [37, 33].

Given below is the pseudo code of a distributed algorithm which solves consensus:

```
choose $v_i$
send $v_i$ to all
wait until received messages from all processes
decide on smallest received value
```

Listing 1: pseudo code for consensus algorithm

In the example above, every process sends its value to all processes in the system and then waits until it has received messages from all the processes in the system. Then it decides on the smallest value that it has received. Now assume that one process in the system is faulty and sends conflicting messages to different subsets of processes. This can give rise to disagreement, that is, a situation where different processes decide on different values. Thus, with a single faulty process in the system, the algorithm fails to function correctly.

Since distributed algorithms find their applications in a wide range of fields which include safety critical applications like the aerospace, flight control systems, medical equipments, automobiles etc [64], it is necessary to ensure their reliability even in the presence of faults in the system. This is done using FTDAs. The idea is to have redundant processes in the system to ensure that the system functions correctly even if some of the processes are faulty. FTDAs define the behavior of the individual processes which compose such systems.

1.2.2 Fault-Tolerant Distributed Algorithms

FTDAs constitute a core topic of the distributed algorithms community with a rich body of results [71, 8]. An FTDA is designed such that the distributed system keeps working correctly even though some processes in the system have failed. The algorithm given below is an example for an FTDA, where n denotes the number of processes in the system and t denotes the maximum number of faulty processes allowed. The algorithm is explained in detail in Chapter 2 and is used as a case study throughout the thesis.

```
code for a correct process i

v_i in { false, true }
accept_i in { false, true } <- false

CODE

if v_i and not sent ⟨echo⟩ before
then send ⟨echo⟩ to all;

if received ⟨echo⟩ from at least t+1 distinct processes and not sent
  ⟨echo⟩ before
then
  send ⟨echo⟩ to all;

if received ⟨echo⟩ from at least n−t distinct processes
then accept_i <- true;
```

Listing 2: Refer to Algorithm 2.1 and Section 2.4.1 for details

Fault assumptions: The maximum number of the faulty processes that can be present in the system such that the system functions correctly, depends on the fault model assumed. Of the different kinds of faults, a *Byzantine fault* [80] is the most unrestricted one. A Byzantine faulty process does not have to stick to the protocol and has no assumptions on its behavior. Thus it can perform any action, including sending incorrect messages to a subset of processes. A *clean crash fault* is the most restricted one, where the faulty process just stops working. Fault models like *send omission* and *receive omission* [53] lie in between Byzantine and crash fault, where the faulty process fails to send or receive some messages occasionally. Faults can also be classified as *transient* or *permanent*. A permanent fault, as the name suggests, is permanent where a process fails once and forever, while a transient fault is temporary.

From a more operational viewpoint, FTDAs typically consist of multiple processes that communicate by message passing over a completely connected communication graph. Since a sender can be faulty, a receiver cannot wait for a message from a specific sender process. Therefore, most FTDAs use counters to reason about their environment. For instance if a process receives a certain message m from more than t distinct processes, it can conclude that at least one of the senders is non-faulty.

For example, the pseudo code given in Listing 2 shows the core logic of the broadcasting algorithm from [90] which tolerates Byzantine faults.

Threshold-based Fault-Tolerant Distributed Algorithms

In this thesis we concentrate on verifying the correctness of threshold-based FTDAs. In addition to the standard execution of actions, which are guarded by some predicate on the local state, most basic distributed algorithms (cf. [71, 8]) add existentially or universally guarded commands involving received messages:

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

(a) existential guard

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

(b) universal guard

Depending on the content of the message `<m>`, the function `action` performs a local state transition and possibly sends messages to one or more processes. Such constructs can be found, e.g., in (non-fault-tolerant) distributed algorithms for constructing spanning trees, flooding, mutual exclusion, or network synchronization [71]. If we try to build such a fault-tolerant distributed algorithm using the construct of Example (a) in the presence of Byzantine faults, the (local state of the) receiver process would be corrupted if the received message `<m>` originates in a faulty process. A faulty process could hence contaminate a correct process. On the other hand, if one tried to use the construct of Example (b), a correct process would wait forever (starve) when a faulty process omits to send the required message. To overcome these problems, FTDAs typically require assumptions on the maximum number of faults, and employ suitable thresholds for the number of messages which can be expected to be received by correct processes. Assuming that the system consists of n processes among which at most t may be faulty, *threshold-guarded commands* such as the following are typically used by fault-tolerant distributed algorithms:

```
if received <m> from n−t distinct processes
then action(m);
```

(c) threshold guard

Assuming that thresholds are functions of the parameters n and t, threshold guards are just generalization of quantified guards as given in Examples (a) and (b). In Example (c), a process waits to receive $n - t$ messages from distinct processes. As there are at least $n-t$ correct processes, the guard cannot be blocked by faulty processes, which avoids the problems of Example (b). In the distributed algorithms literature, we can find a variety of different thresholds: Typical numbers are $\lceil n/2 + 1 \rceil$ (for majority [37, 77]), $t + 1$ (to wait for a message from at least one correct process [90, 37]), or $n - t$ (in the Byzantine case [90, 6] to wait for at least $t + 1$ messages from correct processes, provided $n > 3t$).

In the setting of Byzantine fault tolerance, it is important to note that the use of threshold-guarded commands implicitly rests on the assumption that a receiver can distinguish messages from different senders. In practice, this can be achieved e.g. by using point-to-point links between processes or by message authentication. What is important here is that Byzantine faulty processes are only allowed to exercise control on their own messages and computations, but not on the messages sent by other processes and the computation of other processes.

Resilience condition: The maximum number of faults allowed in a system is restricted by a resilience condition. The resilience condition depends on the failure model assumed and problem to be solved. In case of Byzantine failure, an example of the resilience condition is $n > 3t$ [80] and $f \leq t$, where n is the number of processes in the system, t is the upper threshold of the number of faults allowed in the system and f is the number of faulty processes in the system at any point of time.

1.2.3 Motivation for Automated Verification

Since the intention of FTDAs is to make distributed systems more reliable in presence of faults, it is of utmost importance to ensure that they satisfy their specifications or in layman's terms, that they do what they are designed to do. In the distributed algorithms community, the correctness of algorithms is proved manually using reasoning in natural language [8, 71]. Since the algorithms are represented in pseudo code the semantic details are difficult to be extracted without having an expert knowledge of the algorithm. Moreover, faults introduce additional dimensions of uncertainty and parameterization: the unknown number of faulty processes in the system and the non-determinism introduced by the faulty behavior of processes. With all these factors being involved, it is a very time consuming and creative task to develop manual correctness proofs for these algorithms and it is not surprising that errors creep into such proofs in some cases [70]. Hence our goal is to ensure the correctness of these algorithms using automated verification techniques.

1.2.4 Model Checking

There are several formal verification methods in existence, such as model checking, theorem proving, abstract interpretation etc. [15, 38, 30]. In its original formulation [23], model checking was concerned with verifying the correctness of finite state systems. If we have a finite system model S and a specification φ in temporal logic a model checker checks whether $S \models \varphi$.

We choose model checking for a number of reasons. Model checking provides a high degree of automation, which is preferable in our case because that eliminates the necessity of having in-depth knowledge about the nitty-gritties of the FTDA under consideration. In model checking, the properties to be satisfied are checked against the system model and if there is a violation, a counterexample is reported showing the path along which the property is violated. Such a counterexample is enormously helpful in debugging.

State space explosion is the main challenge of model checking, where the state space grows exponentially with system parameters like the number of processes in the system, number of variables etc. There exist different methods to handle the state space explosion. Abstraction is an effective method to handle state space explosion problem, where a system with a smaller state space is derived from the concrete system, such that the abstract system simulates the concrete system [26]. Thus, the properties only need to be checked in the abstract system. If the abstract system satisfies a property, by construction the concrete system satisfies it. Otherwise, it is checked if the counterexample is spurious or not. If the counterexample is found to be spurious, then the abstraction is refined. If the counterexample is not spurious, the system does not satisfy the property. CEGAR (Counterexample Guided Abstraction Refinement) [25] is a method which refines the abstraction automatically based on the counterexample, if the counterexample is found to be spurious. Partial order reduction [81, 50, 94] exploits the fact that the order of transitions does not affect some properties, to reduce the state space. Predicate abstraction [51, 25] uses theorem provers and SAT/SMT solvers to compute an abstraction of a concrete system. Counter abstraction [35], abstracts away the number of processes by introducing an abstract counter which represents the number of processes in a each local state. Environment abstraction [28], abstracts away the behaviour of the environment of a process.

Theorem proving being another alternative method generates a mathematical proof which shows that the property is satisfied by the system model. Automated Theorem Provers like Isabelle [79], VAMPIRE [84], PVS [85] needs high user interaction in the form of input lemmas to guide the system through the proof. This demands the user to have expert knowledge of the algorithm to be verified. On the other hand, testing, which is not a formal verification method, is completely automated, but it is not complete. Thus model checking is an ideal choice for us, since it is a good balance between completeness and automation.

1.2.5 Parameterized Model Checking

Finite-state models are, however, not always an adequate modeling formalism for software and hardware. Distributed algorithms are parameterized in one or more variables. Thus, the conventional finite state model checker can only verify such algorithms for small system instances with a fixed number of processes. To verify the algorithm for all possible values of a parameter, we need parameterized model checking. Let S_n be a system which is parameterized in the number of processes n and φ be a specification expressed in temporal logic. Then a parameterized model checker checks if $\forall n \geq 0 \; . \; S_n \models \varphi$.

Some FTDAs are described using the parameters n, t and f. Thus, FTDAs are more difficult than the standard setting of parameterized model checking because *a certain number t of the n processes can be faulty.* Importantly, the upper bound t for the faulty processes, which is essentially a fraction of n, is also a parameter. Thus, for example, we have to reason about all systems with $n - f$ non-faulty and f faulty processes, where $f \leq t$ and $n > 3t$, for an FTDA which tolerates Byzantine faults. Now let us take a look at the challenges involved in parameterized model checking of FTDAs.

1.2.6 Challenges

Only very few fault-tolerant distributed algorithms have been automatically verified. This is be because many aspects of distributed algorithms still pose research challenges for model checking. Given below are the five most pressing issues in model checking distributed algorithms:

- There is no commonly agreed-upon distributed computing model, but rather many variants, which differ in subtle details. Moreover, distributed algorithms are usually described in pseudo code, typically using different (alas unspecified) pseudo code languages, which obfuscates the relation to the underlying computing model.

- For many applications, the size of the distributed system, that is, the number of participants is a priori unknown. Hence, the design and verification of distributed algorithms should work for all system sizes. That is, distributed systems are parameterized by construction.

- The inherent concurrency and the uncertainty caused by partial failure lead to many sources of non-determinism, which makes reasoning about distributed algorithms extremely difficult.

- Correctness of distributed algorithms depends on the assumptions made on the environment. Various assumptions made on the environment are for example, only a certain fraction of the processes faulty, the interleaving of steps is restricted, the message delays are bounded etc. Such assumptions are usually spread throughout the literature and explained in natural language. Hence they cannot be extracted from the pseudo code.

Considering all these facts, it is not a surprise that formal verification of FTDAs finds itself in a very premature state of research.

We have identified the two main problems to be attacked in order to be able to model check FTDAs: the formalization problem and verification problem.

Formalization problem

Formalizing the algorithm represented in pseudo code is a major challenge which we have to overcome to be able to formally verify the algorithm. A clear understanding of the semantics of the algorithm is unavoidable for ensuring the correct execution of the algorithm. Thus, a formal framework for specifying the algorithms is necessary to be able to verify them.

In the literature, a vast majority of distributed algorithms are described in pseudo code [89, 6, 95]. The intended semantics of the pseudo code is folklore knowledge among the distributed computing community. Researchers who have been working in this community have intuitive understanding of keywords like “send”, “receive”, or “broadcast”. For instance, inside the community it is understood that there is a semantical difference between “send to all” and “broadcast” in the context of fault tolerance. Moreover, the constraints on the environment are given in a rather informal way.

For example, let us consider the authenticated Byzantine model [37], Here it is assumed that faulty processes may behave arbitrarily. At the same time, it is assumed that there is some authentication service, which provides unbreakable digital signatures. In conclusion, it is thus assumed that faulty processes send messages as they like, *except* for the ones that look like messages sent by correct processes. However, inferring this kind of information about the behavior of faulty processes is a very intricate task.

Thus, a close familiarity with the distributed algorithms community is required to adequately model a distributed algorithm in preparation of formal verification. When the essential conditions are hidden between the lines, one cannot be sure that the algorithm being verified is the one that is actually intended by the authors. Hence there is a need for a formal framework to precisely express distributed algorithms along with their environment. Such a framework should not only be natural for distributed algorithms researchers, but also provide unambiguous and clear semantics. Since distributed algorithms come with a wide range of different assumptions, the framework has to be easily configurable to these situations.

The two major approaches that exist in formalizing distributed algorithms are I/O Automata [72, 62, 76] and TLA [67, 60, 68]. Both these approaches do not cater to our specific need for reason detailed in Section 2.7.

Verification problem (Parameterized Model Checking Problem)

The properties we want to verify are *safety and liveness properties.* Safety properties say that nothing bad can happen in the system, while liveness properties indicate something good will eventually happen. An example for a safety property for a consensus algorithm is, no two correct processes decide on conflicting values. Similarly, an example for a liveness property is, every correct process eventually decides on a value.

Availability of a formal framework still leaves the challenge of parametrized model checking open. The two challenges we face in our verification problem are, dealing with the parameter in the individual process codes, and parameterization in the number of processes in the system.

(i) Parameterized process codes: Unbounded variables are used to model the sending and reception of messages by the processes in our system. Since the number of process are not fixed, every process in our system has an unbounded state. We need an abstraction h, to eliminate the parameters within the process and thus transform the process model I to a bounded state model $h(I)$. The construction of h assures soundness, i.e., for a given specification logic we can assure by construction that $h(I) \models \varphi$ implies $I \models \varphi$. The major drawback of abstraction is incompleteness: if $h(I) \not\models \varphi$ then it does in general not follow that $I \not\models \varphi$. CEGAR addresses this problem by an adaptive procedure, which analyzes the abstract counterexample for $h(I) \not\models \varphi$ on $h(I)$ to find a concrete counterexample or obtain a better abstraction $h'(I)$. For abstraction to work in practice, it is crucial that the abstract domain from which h and h' are chosen is tailored to the problem class and possibly the specification. Abstraction thus is a semi-decision procedure whose usefulness has to be demonstrated by practical examples.

(ii) Parameterization on the number of processes: As we have seen earlier in this chapter, the number of processes in a distributed system is usually not a priori known. Moreover, fault tolerance requires process replication, which in turn depends on the fault model. Thus, an FTDA has multiple parameters, for instance, the number of processes in the system n, the maximum number of allowed faulty processes t and the number of actual faulty processes in the system f. Hence they are represented by an infinite class of structures $\mathbf{S} = S_1, S_2, \dots\}$ rather than a single structure, where $\mathbf{S}$ is parameterized in n, t and f.

1.3 Problem Statement

Given a parametrized system model which consists of an asynchronous parallel combination of n process models and a specification, our goal is to check if the system model satisfies the specification for all possible combinations of the parameter values defined by the resilience condition. Let $M(n,t,f)$ be the system model which consists of a parallel combination of n process models such that $f \leq t$. That is, $M(n,t,f) = M_1(n,t,f) \mid\mid M_2(n,t,f) \mid\mid ... \mid\mid M_n(n,t,f)$. Let F be a family of such systems with all possible combinations of the parameters n, t and f such that, the resilience condition is not violated. Then, if ϕ is a system specification represented as a Linear Temporal Logic (LTL) formula, we need to check if $F \models \phi$.

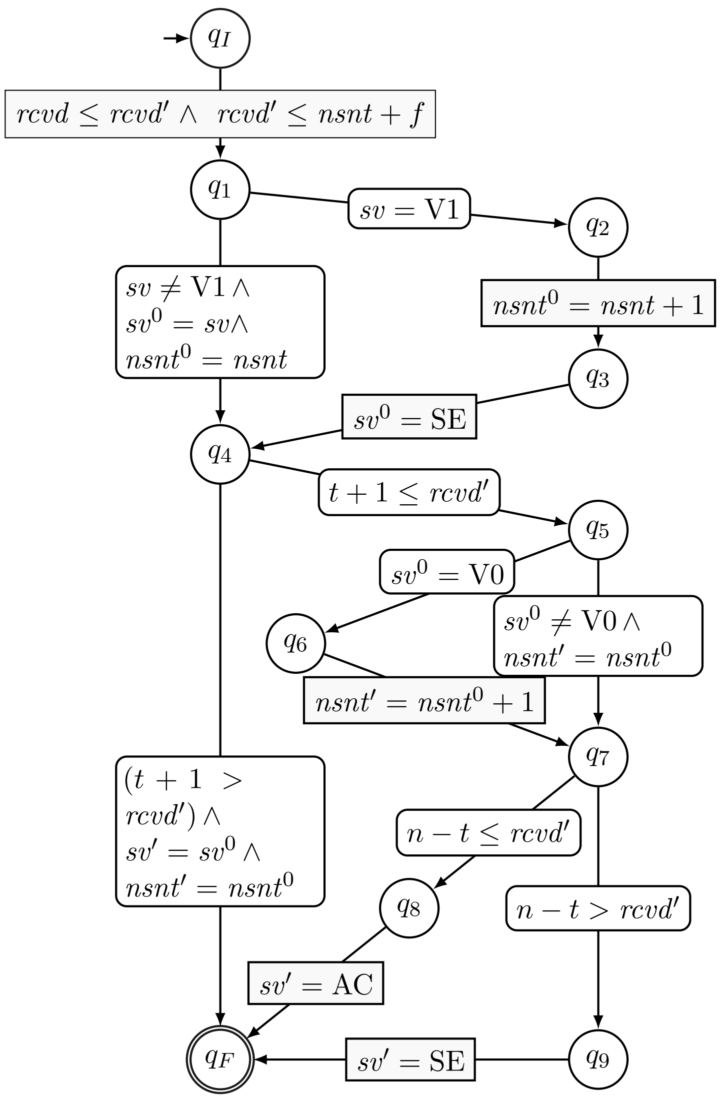


Figure 1.1: CFA for algorithm given in Listing 2.

1.4 Method of Approach

1.4.1 Formalizing the distributed algorithm

The first step is to tackle the formalization problem mentioned in Section 1.2.6. We use extended Control Flow Automata (CFA) which accommodates non-determinism and threshold guards to formalize threshold-guarded FTDAs.

For example, let us consider the CFA given in Figure 1.1 that formalizes the authenticated broadcast algorithm [89]. A detailed explanation of the algorithm, its formalization and modeling can be seen in Chapter 2. The CFA uses the shared integer variable *nsnt* which captures the number of messages sent by non-faulty processes, the local integer variable *rcvd* which stores the number of messages received by the process so far and the local status variable *sv* (ranges over a finite domain), which captures the local progress w.r.t. the FTDA.

We use the CFA in Figure 1.1 to represent one atomic *step* of the FTDA in [89]:Each edge is labeled with a guard. A path from q_I to q_F induces a conjunction of all the guards along it, and imposes constraints on the variables before the step, after the step and on the temporary variables. The variables are renamed after every assignment operation in order to prevent overwriting the value of the same variable. Thus, sv and sv' refer to the same variable, before the step and after the step respectively. sv^0 represents a temporary variable. If we fix the variables before the step, different valuations of the primed variables that satisfy the constraints capture non-determinism. Faulty processes can be modeled explicitly or implicitly in the CFA. In implicit modeling, the model does not include the faulty processes and their effect on the correct processes is modeled by letting them receive more messages than those sent in the system. Our example in Figure 1.1 uses implicit modeling of faulty processes.

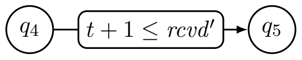
While the extended CFA model of the algorithm serves as a simple mathematical representation, we model the algorithms using an extended version of Promela (input language of Spin model checker) which is used as the input to the model checker Spin to practically verify the algorithm.

1.4.2 Parameterized model checking by abstraction

We give a brief introduction to this step in the following paragraphs. A detailed description can be found in Chapter 4.

From Section 1.4.1 we infer that there are two sources of unboundedness: the integer variables and the parametric number of processes. We deal with these two issues in two steps.

Step 1: PIA data abstraction: We observe that the CFA contains several transitions which are labeled with *threshold guards* that refer to (unbounded) variables and parameters. For instance, the CFA in Figure 1.1 contains the following transition, which is labeled with a threshold guard:



The CFA also contains a guard. Intuitively, the correctness of the FTDA is based on the fact that the values of the thresholds, e.g., $t + 1$ and $n - t$, are sufficiently far apart from each other. From the resilience condition $n > 3t \wedge f \le t$, it follows that $(n - t) - f \ge t + 1$. These properties are also used in the manual correctness proofs [90]. We observe that such FTDAs are designed by carefully choosing the thresholds and the resilience condition. Consequently, our abstraction must be sufficiently precise to preserve the resilience condition and the relationship between thresholds.

The second important observation is that the progress of a process depends on the satisfaction of the threshold guards. Thus, it is not necessary to keep track of the precise value of variables that are compared against thresholds, e.g., $rcvd'$. Rather, in our case study, it is sufficient to know whether the variable under consideration crosses certain threshold values or not. Thus, instead of tracking every single value taken by $rcvd'$, we may just check if $rcvd'$ lies in the interval $[0, t+1[$, or $[t+1, n-t[$, or $[n-t, \infty[$, in order to determine which of the threshold guards of the CFA are satisfied. Our *parametric interval abstraction* PIA exploits this idea. In addition, in Step 2 we will see that we also have to distinguish 0 from other values. Thus, PIA consists of mapping integers to a finite domain of four intervals $I_0 = [0, 1[$ and $I_1 = [1, t+1[$ and $I_2 = [t+1, n-t[$ and $I_3 = [n-t; \infty[$.

We replace the guards that refer to unbounded variables and parameters by their abstraction. For instance, the above transition with the guard $t + 1 \leq rcvd'$ means that $rcvd'$ lies in the intervals $[t+1, n-t[$ or $[n-t, \infty[$. As these correspond to the abstract intervals I_2 and I_3, respectively, we can replace the guard by:

$$\begin{tikzcd} q_4 \arrow[r, "rcvd' = I_2 \vee rcvd' = I_3"] & q_5 \end{tikzcd}$$

The abstraction of the guard $nsnt^0 = nsnt + 1$ can be expressed similarly, which will be demonstrated in Chapter 4. The expression $rcvd' \leq nsnt + f$, which is also used in a guard, is more complicated as it involves two variables and a parameter. Still, the basic abstraction idea is the same. The corresponding abstract expression has the form $(rcvd' = I_0 \wedge nsnt = I_0) \vee (rcvd' = I_0 \wedge nsnt = I_1) \vee \cdots \vee (rcvd' = I_3 \wedge nsnt = I_3)$.

These abstract guards are Boolean expressions over equalities between variables and abstract values. Therefore, it is sufficient to interpret the variables *nsnt* and *rcvd* over the finite domain. Hence, all variables range over finite domains, and we arrive at finite state processes in this way. Our system, however, is still parameterized, namely, in the number of processes. Thus our next task is to get rid of this parameter.

Step 2: PIA counter abstraction. The resultant system obtained by applying the previous step to the CFA shown in Figure 1.1 is still parameterized in the number of processes. We reduce this resultant system a finite state system as follows.

We use the global variable *nsnt* to keep track of the number of messages sent in the system. The global state is represented by the abstract shared variable *nsnt* and by one counter each for each local state. The counters keep a count of the number of processes in each local state. Second, as processes interact only via the *nsnt* variable, precisely counting processes in certain states may not be necessary; as *nsnt* already ranges over the abstract domain, it is natural to count processes in terms of the same abstract domain. Thus we use the same abstract domain to eliminate the unboundedness of the counter values.

The local state of a process is determined by the values of *sv* and *rcvd*. Thus, $\kappa[x, y] = I$ means that the number of processes with $sv = x$ and $rcvd = y$ lies in the abstract interval I. Then, in Figure 1.2, the state s_0 represents the initial states where the number of process with $sv = \texttt{V0}$ and $rcvd = 0$ lies between $t + 1$ and $n - t - 1$ and the number of process with $sv = \texttt{V1}$ and $rcvd = 0$ lies between 1 and t. We omit local states that have the counter value I_0 for better readability.

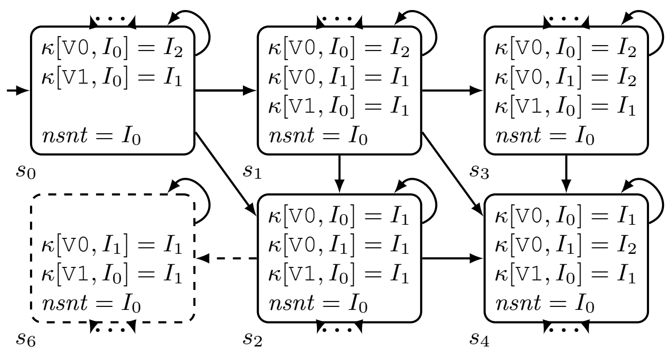

$\kappa[\mathtt{V0}, I_0] = I_2$
$\kappa[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_0
$\kappa[\mathtt{V0}, I_0] = I_2$
$\kappa[\mathtt{V0}, I_1] = I_1$
$\kappa[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_1
$\kappa[\mathtt{V0}, I_0] = I_2$
$\kappa[\mathtt{V0}, I_1] = I_2$
$\kappa[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_3
$\kappa[\mathtt{V0}, I_1] = I_1$
$\kappa[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_6
$\kappa[\mathtt{V0}, I_0] = I_1$
$\kappa[\mathtt{V0}, I_1] = I_1$
$\kappa[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_2
$\kappa[\mathtt{V0}, I_0] = I_1$
$\kappa[\mathtt{V0}, I_1] = I_2$
$\kappa[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_4

Figure 1.2: A small part of the transition system obtained by counter abstraction.

Figure 1.2 gives a small part of the transition system obtained from the counter abstraction starting from initial state s_0. Each transition corresponds to one process taking a step in the concrete system. For instance, in the transition (s_0, s_2) a process with local state $[\mathtt{V0}, I_0]$ changes its state to $[\mathtt{V0}, I_1]$. Therefore, the counter $\kappa[\mathtt{V0}, I_0]$ is decremented and the counter $\kappa[\mathtt{V0}, I_1]$ is incremented. However, as we interpret counters over the abstract domain, the operations of incrementing and decrementing a counter are actually non-deterministic. Consequently, the transition (s_0, s_1) captures the same concrete local step as (s_0, s_2). In (s_0, s_1), the non-deterministic decrement of the abstract counter $\kappa[\mathtt{V0}, I_0]$ did not change its value.

Abstraction refinement: Our abstraction steps result in a system which is an over-approximation of all systems with fixed parameters. For instance, the non-determinism in the counters may "increase" or "decrease" the number of processes in a system, although in all concrete system the number of processes is constant: Consider the transition (s_2, s_6) in Figure 1.2, and let x, y, z be the non-negative integers that are in s_2, abstracted to $\kappa[\mathtt{V0}, I_0]$, $\kappa[\mathtt{V0}, I_1]$, and $\kappa[\mathtt{V1}, I_0]$, respectively. Similarly y' and z' are abstracted to $\kappa[\mathtt{V0}, I_1]$ and $\kappa[\mathtt{V1}, I_0]$ in s_6. If the following inequalities do not have a solution under the resilience condition $(n > 3t, t \geq f)$, then there is no concrete system with a transition between two states that are abstracted to s_2 and s_6, respectively.

$$\begin{aligned} 1 \leq x < t+1,\ 1 \leq\ y\ < t+1,\ 1 \leq\ z\ < t+1,& \\ 1 \leq\ y' < t+1,\ 1 \leq\ z' < t+1,& \\ x + y + z = y' + z' = n - f.& \end{aligned}$$

We use an SMT solver to find out whether there is a solution for the above inequalities, and examine each transition of a counterexample returned by a model checker. If a transition is spurious, then we remove it from the abstract system. We force Spin to prune spurious executions, buy using a flag to indicate the spurious transitions in the current execution.

1.5 State of the Art

The conventional way of proving the correctness of FTDAs is by handwritten proofs [8, 71]. These handwritten proofs are usually difficult to understand and writing them requires expert knowledge of the algorithm. Moreover, proving correctness using handwritten proofs is very tedious and time consuming, and such proofs can be erroneous due to the inherent non-determinism of the algorithms [70].

Several distributed algorithms have been formally verified in the literature. Typically, these papers have addressed specific algorithms in fixed computational models. There are roughly two lines of research. On the one hand, the semi-manual proofs conducted with proof assistants which typically involve an enormous amount of manual work by the user, and on the other hand automatic verification, e.g., using model checking.

Among the work using proof assistants, Byzantine agreement in the synchronous case was considered in [70, 88]. In the context of the heard-of model with message corruption [11] Isabelle proofs are given in [20]. In [21] the authors used Isabelle/HOL to verify a well known consensus algorithm called Paxos, represented in the Heard-Of model. In [70] PVS is used to prove the correctness of the Oral Messages (OM) algorithm of Lamport, Shostak, and Pease. In this paper it is shown that a variant of this algorithm by Thambidurai and Park has serious flaws in spite of the proofs they provided in natural language. In [88] the PVS theorem prover is used to verify the 1-round Byzantine agreement algorithm OMH. Byzantine Paxos algorithm is verified using the TLAPS proof system in [69]. In all these cases, proof assistants are used to verify FTDAs and the verification process involves strong user interaction in the form of supplying lemmas to help the proof assistant verify the algorithms.

Our aim is to automatically verifying FTDAs using model checking techniques. Model checking of fault-tolerant distributed algorithms is usually limited to small instances, i.e., to systems consisting of only few processes (e.g., 4 to 10). However, distributed algorithms are typically designed for parameterized systems, i.e., for systems of arbitrary size. The model checking community has created interesting results toward closing this gap, although it still remains a big research challenge. Model checking has been used to verify FTDAs for different values of system sizes in [91] and [93]. That is, these papers do not tackle the verification of FTDAs for all possible values of the system size and faults.

We will now take a look at the state of the art of parametrized model checking techniques. In [24] a technique to verify a family of finite state-transition systems based on network grammars and abstraction is introduced. In [83] counter abstraction is used to verify liveness properties in parametrized systems. Here, the processes are abstracted away into three groups (0, 1 and many), based on the local states which they are in. In [9], again counter abstraction is used ot verify device drivers. The processes are assumed to be perfectly symmetric with each other and finite state. Environment abstraction is used in [28] to check Lamport's bakery algorithm and Szymanski's mutual exclusion algorithm. Vector Addition Systems with States (VASS) are used to model check concurrent systems with an arbitrary number of finite state process in [48].

In [41] it is shown how to model check ring-based message passing systems by reducing the ring with an arbitrary number of process to a fixed number of cut-off processes. Their method is sound and complete for bidirectional rings and the reduced system is a replica of the original system. These methods are different from the verification of concurrent code which uses mechanisms like shared memory and locks for large systems whose verification is non-trivial due to the size [10, 34, 61, 44].

In [92] German's cache coherence protocol has been verified using the compositional (CMP) method [73, 22] reinforced with invariants derived from message flows to verify cache coherence protocols. CMP method has been used to verify flash protocols in [74] and [22]. In [40] the authors consider parametrized reasoning of cache coherence protocols. Cache coherence is an area where parametrized model checking has been used to verify systems of arbitrary size. Since these protocols are usually described via message passing, they appear similar to asynchronous distributed algorithms. However, issues such as faulty components and liveness are not considered in the literature.

Most of the work on parameterized model checking considers only safety. For example, in [7], the safety of synchronous broadcasting algorithms that tolerate crash or send omission faults has been verified. Notable exceptions are [63, 83] where several notions of fairness are considered in the context of abstraction to verify liveness.

1.6 Structure of the Thesis and Contributions

This thesis presents work published in [59], [58], [57] and [49], except for Chapter 3 which consists of unpublished result. We will now outline the structure of the thesis along with chapter-wise contributions.

In Chapter 2 we deal with the formalization problem in verifying FTDAs. We introduce the formal framework for FTDAs and with the help of a use case algorithm [90], we show how to represent the pseudo code in our extended CFA. Then we show how to translate the algorithm represented in the extended CFA to Promela code, which is the input language of the Spin model checker. We explain the extensions we incorporated in Promela to accommodate several inherent features of FTDAs. We also show how to model different kinds of faults in our formal framework.

In Chapter 3 we discuss the undecidability of the parameterized model checking problem for liveness properties.

Chapter 4 introduces our Parameterized Interval Abstraction. We explain the abstract domain followed by the PIA data abstraction. Then the PIA counter abstraction and the associated formalism are explained with the help of a use case algorithm [90]. We give a general framework for sound refinement of our abstract model and provide a detailed discussion about the refinement techniques we use.

In Chapter 5, we present three different algorithms, explain how we formalize them using the extended CFA and translate them to Promela code. The algorithms differ in the number of message types and the kind of faults tolerated. The algorithms presented are Folklore Reliable Broadcast algorithm [19], Asynchronous Byzantine Agreement algorithm [16] and the Condition-Based Consensus algorithm [77]. We conduct experiments using our tool chain ByMC for the algorithms mentioned above. We present the results of these experiments for both non-parameterized (with fixed parameters) and parameterized versions of these algorithms.

Finally, we provide the summary of our work and discuss future work in Chapter 6.

CHAPTER 2

Formal Framework

In this chapter we present our approach towards solving the first problem to be dealt with in verifying fault-tolerant distributed algorithms: the formalization problem. As we have already discussed in Chapter 1, distributed algorithms are usually expressed using pseudo code with no formal backbone to it and their correctness is proved using reasoning in natural language. This poses a major barrier to the verification efforts of such algorithms due to the difficulty in understanding the multitude of intricate details of the underlying semantics of the algorithm, hidden behind the deceptively simple pseudo code. Thus, the first step towards our goal of automatically verifying FTDAs is of course formalizing them.

This chapter is organized as follows. In Section 2.1 we present the computational model for asynchronous distributed algorithms. We then develop a system model for fault-tolerant distributed algorithms with multiple parameters in Section 2.2 and express the specifications of parameterized systems using Linear Temporal Logic without the next time operator ($\mathsf{LTL}_{\text{-}\mathsf{X}}$). In Section 2.3 we introduce extended CFA which is used to to precisely express the semantics of the FTDA in the form of a transition system. Then we show how to transfer a threshold-based FTDA given in pseudo code to extended CFA, using Algorithm 2.1 in Section 2.4, for different fault assumptions. We use an extended version of Promela (input language of Spin model checker [55]), to encode the algorithm. The Promela code serves as the input to our next step: parameterized model checking by abstraction. It is also used to run experiments using our tool chain ByMC [1]. In Section 2.5 we give the encoding of Algorithm 2.1 for various fault models, in extended Promela. In Section 2.6 we present experiments which validates the adequacy of our formalization method. To this end we try different fixed parameter (non-parameterized) variations of Algorithm 2.1, with different combinations of parameter values and verify that each these variants gives the expected result.

2.1 Computational Model for Asynchronous Distributed Algorithms

The computational model we use for threshold-based FTDAs is the one presented in [46]. Let us first recall the standard assumptions for asynchronous distributed algorithms introduced in Section 1.2.2. A system consists of n processes out of which at most t may be faulty. When considering a fixed computation, we denote by f the actual number of faulty processes. The relation between n, t and f is fixed by the resilience condition. As these parameters do not change during a run, they can be encoded as constants. Correct processes follow the algorithm by taking steps that correspond to the algorithm description. Between every pair of processes, there is a bidirectional link over which messages are exchanged. A link contains two message buffers, each being the receive buffer of one of the incident processes.

A step of a correct process is *atomic* and consists of the following three parts:

- The process possibly receives a message. A process is not forced to receive a message in a step even if there is one in its buffer [46].
- Then, it performs a state transition depending on its current state and the received message.
- Finally, a process may send at most one message to each process, that is, it puts a message in the buffer of the other processes.

Moreover, if a message m is put into a process p's buffer, and p is correct, then m is eventually received, i.e., every message sent is eventually received. This property is called *reliable communication.* Thus, computations are asynchronous in a way that the steps can be arbitrarily interleaved, provided that each correct process takes an infinite number of steps. Since the processes are not forced to receive a message in every step, in order to satisfy the reliable communication property, each correct process should be able to take infinite number of steps. For instance, Algorithm 2 has runs that never accept and are infinite, since processes are not forced to receive messages when they take a step. Conceptually, the standard model requires that processes executing terminating algorithms loop forever in terminal states [71].

2.2 System Model with Multiple Parameters

In this section we develop all notions that are required to precisely state the parameterized model checking problem for a system with multiple parameters.

As we already have seen in Chapter 1, threshold-based FTDAs have multiple parameters. As running example, we use the parameters n, t and f (explained in Section 2.1). We define the processes parameterized over n, t and f in a way that we can modularly compose them into a parameterized system instance.

We define the local variables of the processes, the shared variables and the parameters referring to a single *domain* D that is totally ordered and has the operations of addition and subtraction. In the rest of this thesis we assume that D is the set of nonnegative integers $\mathbb{N}_0$.

Let us start with some notation. Let Y be a finite set of variables ranging over D. We denote the set of all $|Y|$-tuples of variable values, by $D^{|Y|}$. For instance, let $Y = \{x, y, z\}$. That is, the cardinality of Y, $|Y| = 3$. $D^{|Y|} = \{(x_1, y_1, z_1), (x_2, y_2, z_2), ...\}$, where x_i, y_i, z_i, are different valuations of the variables x, y and z, respectively, where $i = 1, 2, ..$. Each valuation is a natural number which belongs to D. Given a vector $\mathbf{s} \in D^{|Y|}$, we use the expression $\mathbf{s}.y$, to refer to the value of a variable $y \in Y$ in vector $\mathbf{s}$. That is, in the above example if $\mathbf{s}$ is (x_1, y_1, z_1), then $\mathbf{s}.y = y_1$ For two vectors $\mathbf{s}$ and $\mathbf{s}'$, by $\mathbf{s} =_X \mathbf{s}'$ we denote the fact that for all $x \in X$, $\mathbf{s}.x = \mathbf{s}'.x$ holds. Thus if $\mathbf{s}' = (x_2, y_2, z_2)$, then $\mathbf{s} =_Y \mathbf{s}'$ means $x_1 = x_2$, $y_1 = y_2$ and $z_1 = z_2$. That is, the value of each variable element of the set Y in $\mathbf{s}$ is the same as its value in $\mathbf{s}'$.

The set of variables V is $\{sv\} \cup \Lambda \cup \Gamma \cup \Pi$. The variable sv is the *status variable* that ranges over a finite set SV of *status values.* For simplicity, we assume that only one status variable is used. However, multiple finite domain status variables can be encoded into sv. The finite set Λ contains variables that range over the domain D. The variable sv and the variables from Λ are *local variables.* The finite set Γ contains the *shared variables* that range over D. The finite set Π is a set of *parameter variables* that range over D, and the *resilience condition RC* is a predicate over $D^{|\Pi|}$. For instance in our use case Algorithm 2.1, $\Pi = \{n, t, f\}$, and the resilience condition $RC(n, t, f)$ is $n > 3t \ \wedge \ f \leq t \ \wedge \ t > 0$. We denote the set of *admissible parameters* by $\mathbf{P}_{RC} = \{\mathbf{p} \in D^{|\Pi|} \mid \mathbf{p} \models RC\}$. That is, the set of admissible parameters is the set of all 3-tuples of parameter values (values of n, t and f), which satisfy the resilience condition.

We use this system model to verify FTDAs as follows: We take a process description that uses the parameters n and t and from this we construct a system instance parameterized in n, t, and f, which then describes all runs of an algorithm in which exactly f faults occur. The verification problem for a distributed algorithm in the *concrete case* with fixed n and t is the composition of model checking problems, where in each problem f takes values such that $f \leq t$. This modeling also allows us to set $f = t + 1$, which models runs in which more faults occur than expected, and search for counterexamples. For the *parameterized case*, we are required to verify the algorithm for all values of parameters that satisfy the resilience condition.

2.2.1 Process Skeleton and Instance

Here we explain the semantics of an individual process in terms of a transition system. A process operates on states from the set $S = SV \times D^{|\Lambda|} \times D^{|\Gamma|} \times D^{|\Pi|}$. Each process starts its computation in an initial state from a set $S^0 \subseteq S$. A relation $R \subseteq S \times S$ defines *transitions* from one state to another, with the restriction that the values of parameters remain unchanged, i.e., for all $(\mathbf{s}, \mathbf{t}) \in R$, $\mathbf{s} =_\Pi \mathbf{t}$. Then, a *parameterized process skeleton* is a tuple $\mathsf{Sk} = (S, S^0, R)$. Note that the set of states includes the set of all $|\Pi|$-tuples of parameter values which satisfy the resilience condition. But every transition relation is restricted in a way that the parameter values do not change when the process goes from one state to another.

We get a process instance by fixing the parameter values $\mathbf{p} \in D^{|\Pi|}$. That is, we choose one tuple from the set of $|\Pi|$-tuples $D^{|\Pi|}$. We can restrict the set of process states to $S|_{\mathbf{p}} = \{\mathbf{s} \in S \mid \mathbf{s} =_{\Pi} \mathbf{p}\}$ and the set of transitions to $R|_{\mathbf{p}} = R \cap (S|_{\mathbf{p}} \times S|_{\mathbf{p}})$. Thus, the set of the states of the processes are restricted to those where the parameter values are equal to the specific combination chosen from the set of $|\Pi|$-tuples $D^{|\Pi|}$. The transition relation is restricted to the transition between these states. Then, a *process instance* is a process skeleton $\mathsf{Sk}|_{\mathbf{p}} = (S|_{\mathbf{p}}, S^0|_{\mathbf{p}}, R|_{\mathbf{p}})$ where $\mathbf{p}$ is constant.

2.2.2 System Instance

For fixed admissible parameters $\mathbf{p}$, a distributed system is modeled as an asynchronous parallel composition of identical processes $\mathsf{Sk}|_{\mathbf{p}}$. We define the size of a system as the number of processes in the system, using a function $N\colon \mathbf{P}_{RC} \to \mathbb{N}_0$. The number of processes in the system model depends on our modeling choices. For instance, we may choose to model only the correct processes explicitly for a byzantine tolerant system. In such a scenario the system size $N(n,t,f) = n - f$.

Given $\mathbf{p} \in \mathbf{P}_{RC}$, and a process skeleton $\mathsf{Sk} = (S, S^0, R)$, a system instance is defined as an asynchronous parallel composition of $N(\mathbf{p})$ process instances, indexed by $i \in \{1, \ldots, N(\mathbf{p})\}$, with standard interleaving semantics [38]. Let AP be a set of atomic propositions. A *system instance* $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is a Kripke structure $(S_I, S_I^0, R_I, \mathrm{AP}, \lambda_I)$ where:

- $S_I = \{(\sigma[1], \ldots, \sigma[N(\mathbf{p})]) \in (S|_{\mathbf{p}})^{N(\mathbf{p})} \mid \forall i, j \in \{1, \ldots, N(\mathbf{p})\}, \sigma[i] =_{\Gamma \cup \Pi} \sigma[j]\}$ is the set of *(global) states*. Informally, a global state σ is a Cartesian product of the state $\sigma[i]$ of each process i, with identical values of parameters and shared variables for each process.
- $S_I^0 = (S^0)^{N(\mathbf{p})} \cap S_I$ is the set of *initial (global) states*, where $(S^0)^{N(\mathbf{p})}$ is the Cartesian product of initial states of individual processes.
- A transition (σ, σ') from a global state $\sigma \in S_I$ to a global state $\sigma' \in S_I$ belongs to R_I iff there is an index i, $1 \le i \le N(\mathbf{p})$, such that:

  (move) The i-th process *moves*: $(\sigma[i], \sigma'[i]) \in R|_{\mathbf{p}}$.

  (frame) The values of the local variables of the other processes are preserved: for every process index $j \neq i$, $1 \le j \le N(\mathbf{p})$, it holds that $\sigma[j] =_{\{sv\} \cup \Lambda} \sigma'[j]$.
- $\lambda_I : S_I \to 2^{\mathrm{AP}}$ is a state labeling function.

Remark 1. *The set of global states S_I and the transition relation R_I are preserved under every transposition $i \leftrightarrow j$ of process indices i and j in $\{1, \ldots, N(\mathbf{p})\}$. That is, every system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is* fully symmetric *by construction.*

Remark 2. *We call a pair of resilience condition and system size function (RC, N)* natural *if $\{N(\mathbf{p}) \mid RC(\mathbf{p})\}$ is infinite. From now on we consider only families of system* *instances with natural (RC, N), as this implies that there is no bound on the number of processes. Since $N(\mathbf{p})$ is a function, in principle it is possible that $N(\mathbf{p})$ takes a single value, which then do not reflect parameterized systems. In order to avoid such a scenario, it is important to define the natural pair of resilience condition and system size as explained above, so that all possible values of $N(\mathbf{p})$ are taken into consideration.*

Temporal logic

We write specifications of our parameterized systems in $\mathsf{LTL}_{\text{-}\mathsf{X}}$. We use the standard definitions of paths and $\mathsf{LTL}_{\text{-}\mathsf{X}}$ semantics [38]. A formula of $\mathsf{LTL}_{\text{-}\mathsf{X}}$ is defined inductively as:

- a literal p or $\neg p$, where $p \in \mathrm{AP}_{SV}$, or
- $\mathbf{F}\,\varphi$, $\mathbf{G}\,\varphi$, $\varphi\,\mathbf{U}\,\psi$, $\varphi \vee \psi$, and $\varphi \wedge \psi$, where φ and ψ are $\mathsf{LTL}_{\text{-}\mathsf{X}}$ formulas.

This contrasts the vast majority of work on parameterized model checking where *indexed* temporal logics are used [17, 29, 27, 42]. The reason for the use of indexed temporal logics is that they allow to express the progress of *individual* processes, e.g., in dining philosophers it is required that if a philosopher i is hungry, then i eventually eats. Intuitively, dining philosophers requires us to trace indexed processes along a computation, e.g., $\forall i.\ \mathbf{G}\,(\mathrm{hungry}_i \rightarrow (\mathbf{F}\,\mathrm{eating}_i))$.

In contrast, fault-tolerant distributed algorithms are typically used to achieve certain *global* properties such as, consensus (agreeing on a common value), or broadcast (ensuring that all processes deliver the same set of messages). To capture these kinds of properties, we have to trace only existentially or universally quantified properties, e.g., one of the broadcast specifications (relay) [90] states that if some correct process accepts a message, then all (correct) processes accept the message. That is, $(\mathbf{G}\,(\exists i.\ \mathrm{accept}_i)) \rightarrow (\mathbf{F}\,(\forall j.\ \mathrm{accept}_j))$.

We therefore use a temporal logic where the *quantification over processes is restricted to propositional formulas.*

2.2.3 Atomic Propositions

We will need two kinds of quantified propositional formulas. First, we introduce the set AP_{SV} that contains atomic propositions that capture comparison against some status value $Z \in SV$, i.e.,

$$[\forall i.\ sv_i = Z] \text{ and } [\exists i.\ sv_i = Z]\,.$$

This allows us to express specifications of distributed algorithms.

Second, in order to express comparison of variables ranging over D, we add a set of atomic propositions AP_D that capture the comparison of variables x, y, and a constant c, all of them ranging over D. AP_D consists of propositions of the form

$$[\exists i.\ x_i + c < y_i]\,.$$

We then define the set of atomic propositions AP to be the disjoint union of AP_{SV} and AP_D. The labeling function λ_I of a system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ maps its state σ to expressions p from AP as follows:

$$
\begin{aligned}
[\forall i.\ sv_i = Z] \in \lambda_I(\sigma) \text{ iff } & \bigwedge_{1 \le i \le N(\mathbf{p})} (\sigma[i].sv = Z) \\
[\exists i.\ sv_i = Z] \in \lambda_I(\sigma) \text{ iff } & \bigvee_{1 \le i \le N(\mathbf{p})} (\sigma[i].sv = Z) \\
[\exists i.\ x_i + c < y_i] \in \lambda_I(\sigma) \text{ iff } & \bigvee_{1 \le i \le N(\mathbf{p})} (\sigma[i].x + c < \sigma[i].y)
\end{aligned}
$$

Fairness. We are interested in verifying safety and liveness properties. The latter can be usually proven only in the presence of fairness constraints. As in [63, 83], we consider verification of safety and liveness in systems with *justice* fairness constraints. We define fair paths of a system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ using a set of justice constraints $J \subseteq \mathrm{AP}_D$. A path π of a system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is *J-fair* iff for every $p \in J$ there are infinitely many states σ in π with $p \in \lambda_I(\sigma)$. That is, p holds infinitely many times in the path π, for π to be *J-fair.* By $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \models_J \varphi$ we denote that the formula φ holds on all J-fair paths of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$.

Definition 3. *Parameterized Model Checking Problem: Given a system description containing*

- *a domain D,*
- *a parameterized process skeleton $\mathsf{Sk} = (S, S_0, R)$,*
- *a resilience condition RC (generating a set of admissible parameters $\mathbf{P}_{RC}$),*
- *a system size function N,*
- *justice requirements J,*

and an $\mathsf{LTL}_{\text{-}\mathsf{X}}$ formula φ, the parameterized model checking problem *(PMCP) is to verify $\forall \mathbf{p} \in \mathbf{P}_{RC}.\ \mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \models_J \varphi$.*

2.3 Extended CFA for Threshold-based FTDAs

In this section we explain our formalization approach using extended CFA in detail. CFA offers a simple way to express the semantics of an algorithm in a precise way. It also offers us a convenient model which we can use to explain our abstraction method. Thus, we formalize threshold-based FTDAs as a transition system, using an extended version of CFA which was briefly introduced in Section 1.4.1. We also gave an example CFA in Figure 1.1.

Processes that run distributed algorithms execute the same acyclic piece of code repeatedly. In the parlance of distributed algorithms, a single execution of this code is called a step, and steps of correct processes are considered to be atomic. Distributed algorithms can be classified based on what can happen during a step, which in turn depends on the actual code. For instance, in our case study, a step consists of a receive, a computation and a sending phase. We use the concept of CFA to formalize the steps taken by each process. The paths from the initial to the final location of the CFA describe *one step* of the distributed algorithm. We define an extended version of CFA to represent distributed algorithms that contain threshold guards.

The concept of CFA was introduced by Henzinger et al. [54], as a framework to describe the control flow of a program. We extend the original version of the CFA so that it can handle the features of a threshold-based FTDA. Formally, a *guarded control flow automaton* is an edge-labeled directed acyclic graph $\mathcal{A} = (Q, q_I, q_F, E)$ with a finite set Q of nodes called locations, an initial location $q_I \in Q$, and a final location $q_F \in Q$. A path from q_I to q_F is used to describe one step of a distributed algorithm. The edges have the form $E \subseteq Q \times \texttt{guard} \times Q$, where `guard` is defined as an expression of one of the following forms where $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, and $\Pi = \{p_1, \ldots, p_{|\Pi|}\}$:

- if $Z \in SV$, then $sv = Z$ and $sv \neq Z$ are *status guards*;
- if x is a variable in D and $\lhd \in \{\leq, >\}$, then
$$a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i \;\lhd\; x$$
is a *threshold guard*;
- if $y, z_1, \ldots, z_k$ are variables in D for $k \geq 1$, and $\lhd \in \{=, \neq, <, \leq, >, \geq\}$, and $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, then
$$y \;\lhd\; z_1 + \cdots + z_k + (a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i)$$
is a *comparison guard*;
- a conjunction $g_1 \wedge g_2$ of guards g_1 and g_2 is a guard.

Status guards are used to capture the basic control flow. Threshold guards capture the core primitive of the FTDAs we consider. Finally, comparison guards are used to model send and receive operations. Further, we model the faults such that the correct processes receive more or less messages than the actual number of messages that have been sent, based on the fault model.

Definition 4 (PMCP for CFA)**.** *We define the Parameterized Model Checking Problem for CFA $\mathcal{A}$ by specializing Definition 3 to the parameterized process skeleton $\mathsf{Sk}(\mathcal{A})$.*

In the following section we show how to represent an algorithm given in pseudo code using our extended CFA with the help of Algorithm 2.1. The soundness of the modeling approach requires involved arguments which is discussed in Section 2.4.

2.4 Transferring Pseudo Code to Extended CFA

To capture the step semantics of FTDAs, a step of the distributed algorithm is defined via a path from the initial location of the CFA to the final location. In this section we explain the process of transferring the pseudo code to our extended CFA with the help of an use case. For this purpose, let us recall the core logic of the reliable broadcast algorithm in Algorithm 2.1, which was already introduced in Chapter 1.

Algorithm 2.1 Core logic of the broadcasting algorithm from [90].

```
code for a correct process $i$

$v_i$ in { false , true }
$accept_i$ in { false , true } <- false

CODE

if $v_i$ and not sent ⟨echo⟩ before
then send ⟨echo⟩ to all;

if received ⟨echo⟩ from at least t+1 distinct processes and not sent
  ⟨echo⟩ before
then
  send ⟨echo⟩ to all;


if received ⟨echo⟩ from at least n−t distinct processes
then $accept_i$ <- true;
```

2.4.1 Algorithm 2.1 in detail

Reliable broadcast is an ongoing “system service” with the following informal specification: Each process i may invoke a primitive called broadcast by calling $bcast(i, m)$, where m is a unique message content. Processes may deliver a message by invoking $accept(i, m)$ for different process and message pairs (i, m). The goal is that all correct processes invoke $accept(i, m)$ for the same set of (i, m) pairs, under some additional constraints, which are, all messages broadcast by correct processes must be accepted by all correct processes, and $accept(i, m)$ may not be invoked, if i is correct and i has not invoked $bcast(i, m)$. Our case study is to verify that the algorithm from [90] implements these primitives on top of point-to-point channels, in the presence of Byzantine faults. In [90] the specifications were given in natural language as follows:

(U) Unforgeability. If correct process i does not broadcast (i, m), then no correct process ever accepts (i, m).

(C) Correctness. If correct process i broadcasts (i, m), then every correct process accepts (i, m).

(R) Relay If a correct process accepts (i, m), then every other correct process accepts (i, m).

In [90], the instances for different (i, m) pairs do not interfere. Therefore, we will not consider i and m. Rather, we distinguish the different kinds of invocations of $\mathit{bcast}(i, m)$ that may occur, e.g., the cases where the invoking process is faulty or correct. As we focus on the core functionality, we do not model the broadcaster explicitly. We observe that correct broadcasters send either to all or to no other correct processes. We model this by using the initial *status values* V1 and V0. V0 denotes that the process has not received the message from the broadcaster and V1 denotes that the process has received the message from the broadcaster.

The core logic in Algorithm 2.1 is typical pseudo code found in the distributed algorithms literature. The lines `8-18` describe one step of the algorithm. Receiving messages is implicit and performed before line `8`, and the possible sending of messages is deferred to the end, and is performed after line `18`.

We observe that the send operation of a process always sends to all. Moreover, lines `8-18` only consider messages of type ⟨echo⟩, while all other messages are ignored. Hence, a Byzantine faulty process has an impact on correct processes only if they send an ⟨echo⟩ when they should not, or vice versa.

Note that faulty processes may behave two-faced, that is, they may send messages only to a subset of the correct processes. Moreover, faulty processes may send multiple ⟨echo⟩ messages to a correct process. However, from the code we observe that multiple receptions of such messages do not influence the number of messages received by "distinct" processes due to non-masquerading as mentioned in Section 1.2.2. Finally, the condition "not sent ⟨echo⟩ before" guarantees that each correct process sends ⟨echo⟩ at most once.

2.4.2 Our modeling choices

The most immediate choice is that we consider the set of parameters Π to be $\{n, t, f\}$ and $RC(n, t, f) = n > 3t \wedge f \leq t \wedge t > 0$. In the pseudo code, the status of a process is only implicitly mentioned. We have to represent the following information in the status variable:

i the initial state,

ii whether a process has already sent ⟨echo⟩, and

iii whether a process has set accept to TRUE.

Observe that once a process has sent ⟨echo⟩, its value v_i does not interfere anymore with the further state transitions. Moreover, a process only sets accept to TRUE if it has sent a message (or is about to do so in the current step). Hence, we define the set SV to be $\{\mathrm{V0}, \mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$, where $SV_0 = \{\mathrm{V0}, \mathrm{V1}\}$. V0 corresponds to the case where initially $v_i =$ FALSE, and V1 to the case where initially $v_i =$ TRUE. Further, SE means that a process has sent an ⟨echo⟩ message but has not set accept to TRUE yet, and AC means that the process has set accept to TRUE. Having fixed the status values, we can formalize the specifications, *unforgeability*, *correctness*, and *relay* mentioned in Section 2.4.1 as follows:

$$\mathbf{G}\ ([\forall i.\ sv_i \neq \mathrm{V1}] \rightarrow \mathbf{G}\ [\forall j.\ sv_j \neq \mathrm{AC}]) \tag{U}$$

$$\mathbf{G}\ ([\forall i.\ sv_i = \mathrm{V1}] \rightarrow \mathbf{F}\ [\exists j.\ sv_j = AC]) \tag{C}$$

$$\mathbf{G}\ ([\exists i.\ sv_i = \mathrm{AC}] \rightarrow \mathbf{F}\ [\forall j.\ sv_j = AC]) \tag{R}$$

Note carefully that (U) is a safety specification while (C) and (R) are liveness specifications.

As the asynchrony of steps is already handled by our parallel composition described in Section 2.2.2, what remains is to describe the semantics of sending and receiving messages in our system model using control flow automata.

Let us first focus on messages sent to and received from the correct processes. Since each correct process sends at most one message, and multiple messages from faulty processes have no influence, it would be sufficient to represent each buffer by a single variable that represents whether a message of a certain kind has been put into the buffer. As we have only ⟨echo⟩ messages sent by correct processes, it is sufficient to model one variable per buffer. Moreover, if we only consider the buffers between correct processes, due to the "send to all" it is sufficient to capture all messages between correct processes in a single variable. To model this, we introduce the shared variable *nsnt*.

The reception of messages can then be modeled by a local variable *rcvd* whose update depends on the messages sent. In particular, upon a receive, the variable *rcvd* can be increased to any value less than or equal to *nsnt*.

Now, it remains to model faults. As our system model is symmetric by construction, all processes must be identical to each other. This allows at least two possibilities to model faults:

- We capture whether a process is correct or faulty using a flag in the status, and require that in each run $f \leq t$ processes are faulty. Then we would have to derive a CFA sub-automaton for faulty processes, and would need additional variables to capture sent messages by faulty processes.
- We consider the system to consist of correct processes only, let $N(n,t,f) = n - f$, and model only the influence of faults, via the messages correct processes may receive. This can be done by allowing each correct process to receive at most f messages more than sent by correct ones, that is that *rcvd* can be increased to any value less than or equal to $nsnt + f$.

Implementing the first option would require more variables: the additional flag to distinguish correct from faulty processes, and the additional variables to capture messages by faulty processes. These variables would increase the state space, and thus make this option non-practical. Moreover, we would have to capture the number of faults f and the corresponding resilience condition. Therefore, we have implemented the latter approach.

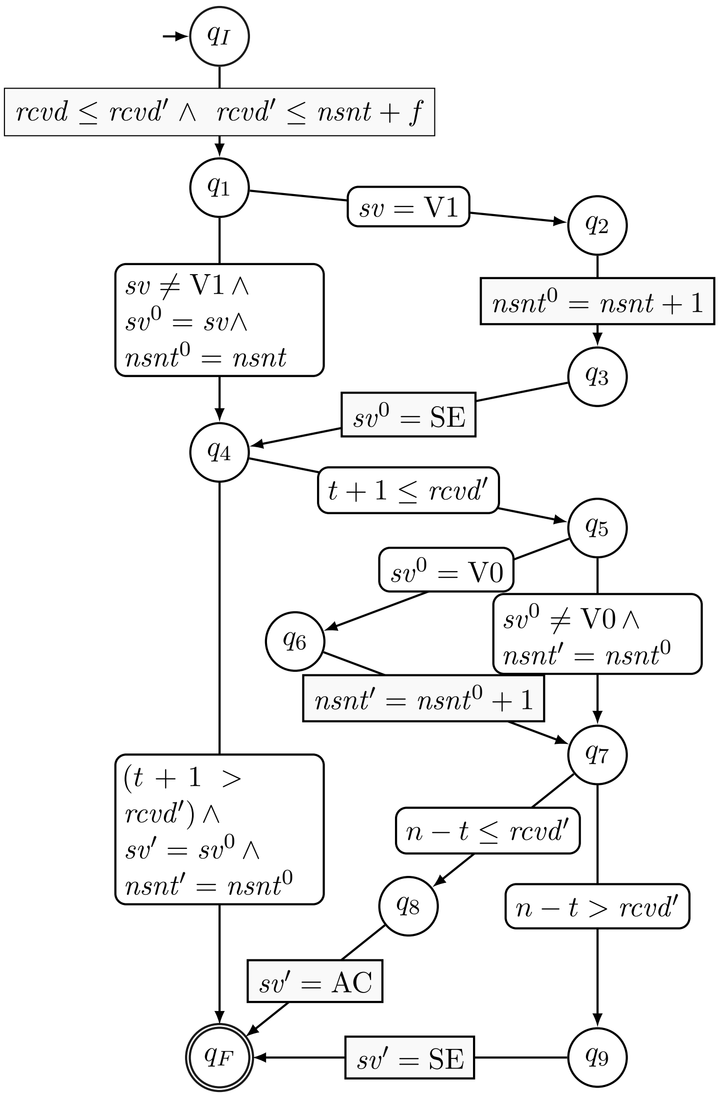


Figure 2.1: CFA of our use case Algorithm 2.1.

Based on this discussion we directly obtain the CFA given in Figure 2.1 that describes the steps of Algorithm 2.1. Note that its structure follows the pseudo code description of Algorithm 2.1.

Fairness: Relevant liveness properties can typically only be guaranteed if the underlying system ensures some fairness guarantees (refer to Section 2.2.3). In asynchronous distributed systems one assumes communication fairness. That is, every message that has been sent is eventually received. The statement $\exists i.\ rcvd_i < nsnt$ describes a global state where messages are still in transit. It follows that a formula ψ defined by

$$\mathsf{G}\,\mathsf{F}\,\neg\,[\exists i.\ rcvd_i < nsnt] \qquad \text{(RelComm)}$$

states that the system periodically delivers all messages sent by (correct) processes. Thus, the formula $\neg\psi$ defined by

$$\mathsf{F\,G}\ [\exists i.\ rcvd_i < nsnt] \tag{UnrelComm}$$

states that the system violates communication fairness. We only require a liveness specification φ to be satisfied if the system is communication fair. In other words, φ is satisfied *or* the communication is unfair, that is, $\varphi \vee \neg\psi$. Our approach is to verify $\varphi \vee \neg\psi$.

Along all paths where communication is fair, the value of $rcvd_i$ has to reach at least the value of $nsnt$. Since $rcvd_i$ can only increase when i takes a step, i is forced to take steps as long as it has not enough received messages to take any other actions yet. That is, receiving is a non-blocking operation. Thus, by this modeling communication fairness implies some form of computation fairness.

2.4.3 Obtaining a skeleton from a CFA

One step of a process skeleton is defined by a path from q_I to q_F in a CFA. In order to obtain the process skeleton induced by the CFA, we are required to capture these steps as a transition relation. In particular, we need to compute a formula that represents this relation. To this end, we use the standard technique introduced by Cytron et al. [31] to construct CFA that are in the form of single static assignment (SSA). Informally, a variable x is used to represent the value before a step and the variable x' to represent value after the step. That is, we create new variables by renaming the existing ones, and use an assignment operation to avoid overwriting. For example, the increment operation $x = x + 1$ is written as, $x' = x + 1$, to avoid the variable x from being overwritten. Given SV, Λ, Γ, Π, RC, and a CFA $\mathcal{A}$, we define the process skeleton $\mathsf{Sk}(\mathcal{A}) = (S, S^0, R)$ induced by $\mathcal{A}$ as follows:

The set of variables used by the CFA is $W \supseteq \Pi \cup \Lambda \cup \Gamma \cup \{sv\} \cup \{x' \mid x \in \Lambda \cup \Gamma \cup \{sv\}\}$.

A path p from q_I to q_F of CFA induces a conjunction of all the guards along it. We call a mapping v from W to the values from the respective domains a *valuation.* We may write $v \models p$ to denote that the valuation v satisfies the guards of the path p. We are now in the position to define the mapping between a CFA $\mathcal{A}$ and the transition relation of a process skeleton $\mathsf{Sk}(\mathcal{A})$: If there is a path p and a valuation v with $v \models p$, then v defines a single transition (s, t) of a process skeleton $\mathsf{Sk}(\mathcal{A})$, if for each variable $x \in \Lambda \cup \Gamma \cup \{sv\}$ it holds that $s.x = v(x)$ and $t.x = v(x')$ and for each parameter variable $z \in \Pi$, $s.z = t.z = v(z)$.

Finally, to specify S^0, all variables of the skeleton that range over D are initialized to 0, and sv ranging over SV takes an initial value from a fixed subset of SV.

2.4.4 Modeling other fault scenarios

Fault scenarios other than Byzantine faults can be modeled by changing the system size, using conditions similar to (RelComm), and slightly changing the CFA. More precisely,

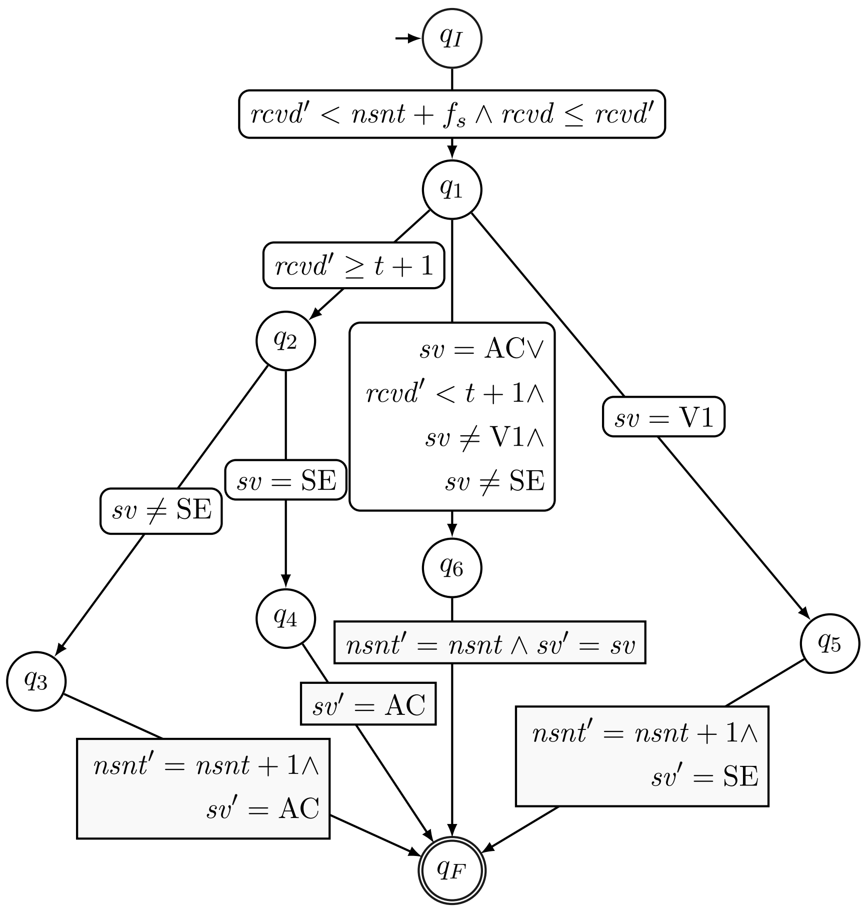


Figure 2.2: CFA of Broadcast Algorithm in [89] with symmetric faults

by changing the guard on the edge leaving q_I that corresponds to receiving messages, we can change the fault model.

Symmetric faults [8]

A symmetric faulty process may send a wrong message, but it sends the message to either all the processes or to none of them. Symmetric faults are modeled implicitly, similar to Byzantine faults. That is, the effects of the faulty process on the correct processes are modeled instead of modeling the faulty processes explicitly. Thus, the system size is $n - f_s$, where f_s is the actual number of faulty processes in the system in a specific run. The effect of the faulty processes on correct processes is modeled by letting the correct processes receive f_s messages more that the number of messages sent in the system. The CFA of Algorithm 2.1 for symmetric faults is shown in Figure 2.2. The fairness condition used to verify liveness in case of symmetric faults is $\mathbf{G}\,\mathbf{F} \neg [\exists i.\ rcvd_i < nsnt + f_s]$.

Omission faults [52]

Figure 2.3 shows the CFA of Algorithm 2.1 for omission faults. Processes which exhibit omission faults are not as malicious as the Byzantine faulty processes. They do not send conflicting messages to different processes. Neither do they send messages which do not comply with the algorithm. But they may fail to send messages to some or all processes. That is, their behavior is more restrictive when compared to Byzantine faulty processes.

We model the omission faulty processes explicitly by setting $N(n, t, f) = n$. Thus, the processes may receive all the messages being sent in the system, but not more. That is, *rcvd* can be increased to any value less than or equal to *nsnt*, which is reflected by the guard on the edge leaving q_I. While verifying liveness, we use the condition $\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_i + f < nsnt]$ to express fairness in communication and thus, model the effect of faulty processes on the number of messages received (f processes may fail to send messages).

Clean crash faults [96]

These faults are more restrictive than omission faults. With clean crash faults, a faulty process either sends a message to all processes or it sends it to none. Moreover, the messages sent have to comply with the algorithm. That is, they also follow the algorithm in a way that they do not send malicious messages. As in the case of omission faults, we model the faulty process explicitly. Thus the system size is n.

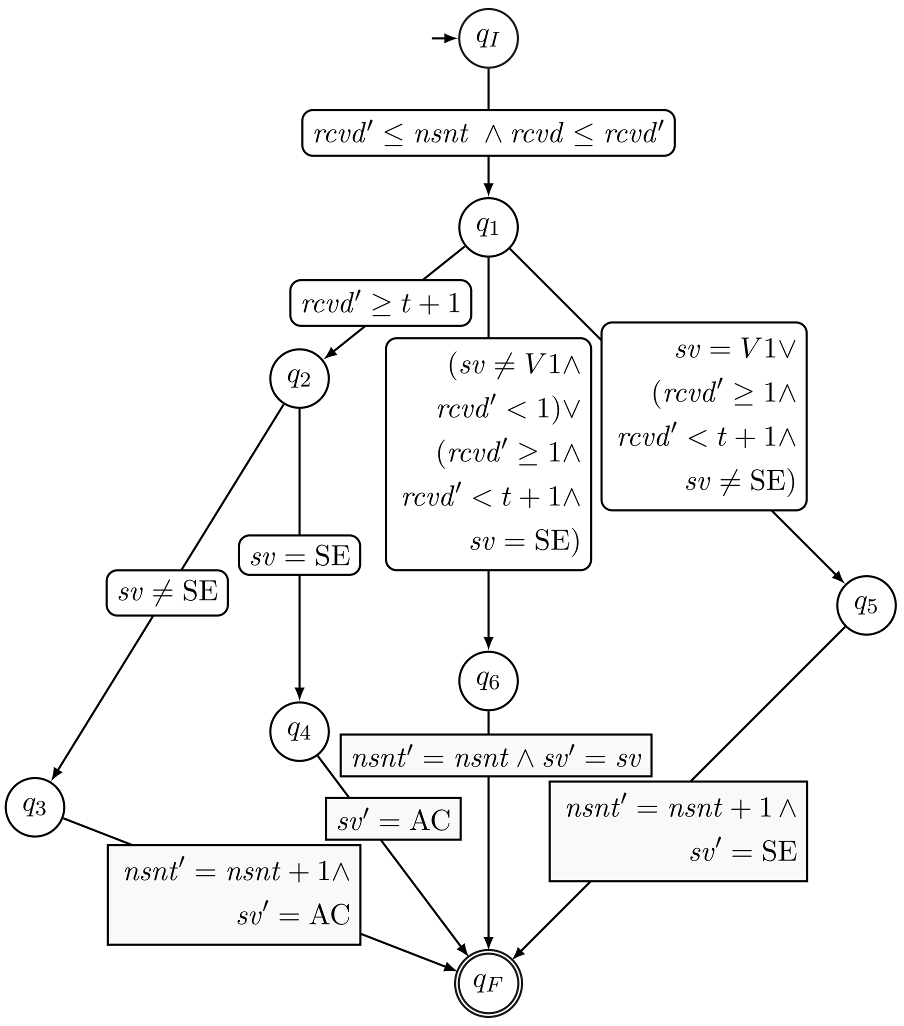
There are two ways to model the effect of faults on the correct processes:

Modeling faults by communication restrictions: Here we distinguish two kinds of faulty processes: those that crash before sending the message and those that crash after sending a message. If a process crashes before they send a message, the correct processes do not receive any messages from it. If it crashes after sending a message, the correct processes should eventually receive the message sent by the faulty process. Let f_c be the number of processes which crash, f_{nc} be the number of processes which crash before sending the message and t, the maximum number of faulty processes allowed in the system. That is, $f_{nc} \leq f_c \leq t \leq n$.

Thus, *rcvd* can be incremented up to $nsnt - f_{nc}$, since f_{nc} processes fail to send messages. Figure 2.4 shows the CFA of our use case algorithm with clean crash fault tolerance, for the first fault model. The communication fairness in this model of clean crashes is represented by the condition $\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_i < nsnt - f_{nc}]$

That is, for the communication to be fair, every correct process must eventually receive all the messages sent and f_{nc} may fail to send messages. Note that this includes the messages sent by the faulty processes before crashing. Figure 2.4 shows the CFA of Algorithm 2.1 with clean crash faults for this model.

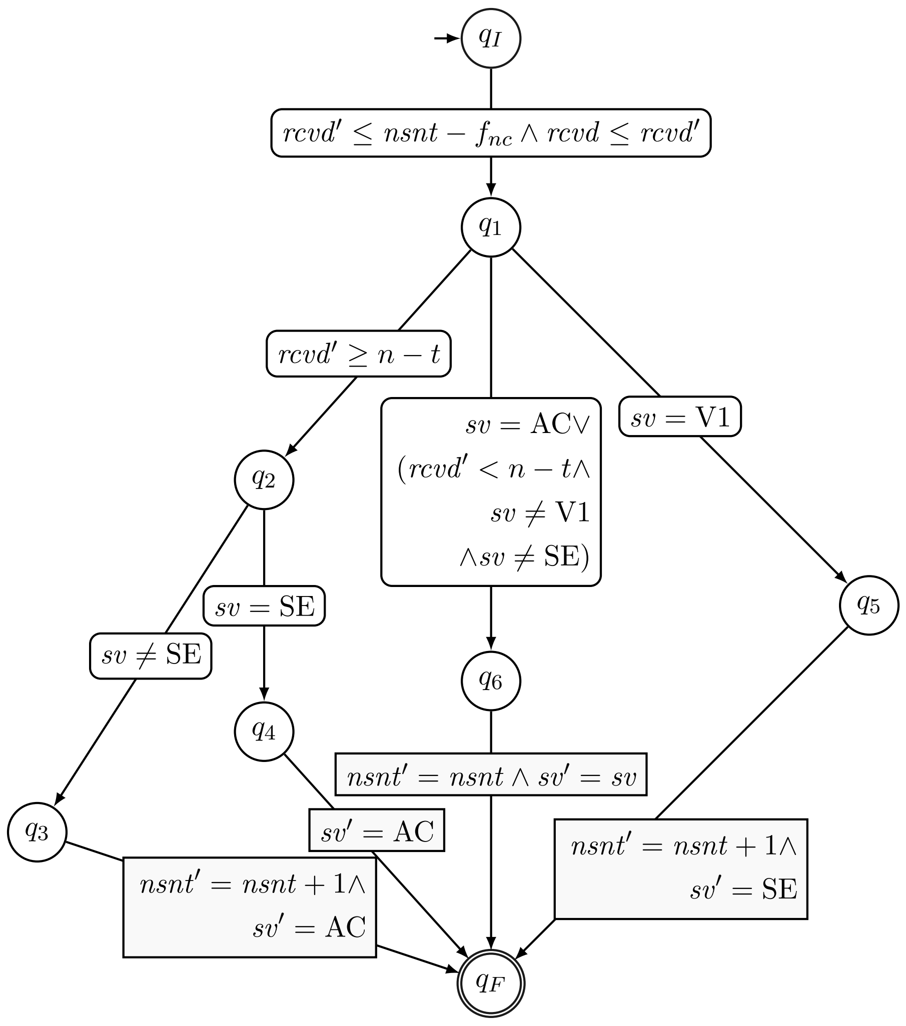
Modeling faults by adding local state: In this modeling method we have a distinct status value CR, which is used to distinguish a process which has crashed. We use a global variable *nfaulty*, which denotes the number of processes that has crashed. We let



q_I
$rcvd' \leq nsnt \ \wedge rcvd \leq rcvd'$
q_1
$rcvd' \geq t+1$
q_2
$(sv \neq V1 \wedge$ $rcvd' < 1) \vee$ $(rcvd' \geq 1 \wedge$ $rcvd' < t+1 \wedge$ $sv = \mathrm{SE})$
$sv = V1 \vee$ $(rcvd' \geq 1 \wedge$ $rcvd' < t+1 \wedge$ $sv \neq \mathrm{SE})$
$sv = \mathrm{SE}$
$sv \neq \mathrm{SE}$
q_5
q_6
q_4
$nsnt' = nsnt \wedge sv' = sv$
q_3
$sv' = \mathrm{AC}$
$nsnt' = nsnt + 1 \wedge$ $sv' = \mathrm{SE}$
$nsnt' = nsnt + 1 \wedge$ $sv' = \mathrm{AC}$
q_F

Figure 2.3: CFA of Broadcast Algorithm in [89] with omission faults

processes crash if the number of crashed process is less than f_c, which is the number of actual crashes in a run. When a process crashes, its status value is changed to CR and it increments the global variable *nfaulty*. We do not need to discriminate between the messages sent by a faulty process vs. those sent by correct process, since if a clean crash faulty process sends messages it sends to all processes and the messages sent are correct. Thus, *rcvd* can be incremented up to *nsnt*, because *nsnt* represents the message sent by the correct process and the faulty processes. The communication fairness in this case is represented by the condition $\mathbf{G}\,\mathbf{F} \neg [\exists i.\ rcvd_i < nsnt]$ That is, for the communication to be fair every correct process must eventually receive all the messages sent in the system. Figure 2.5 shows the CFA of our use case algorithm with clean crash fault tolerance, for this fault model.



q_I
$rcvd' \leq nsnt - f_{nc} \wedge rcvd \leq rcvd'$
q_1
$rcvd' \geq n - t$
$sv = \text{AC} \vee (rcvd' < n - t \wedge sv \neq \text{V1} \wedge sv \neq \text{SE})$
$sv = \text{V1}$
q_2
$sv = \text{SE}$
$sv \neq \text{SE}$
q_5
q_6
q_4
$nsnt' = nsnt \wedge sv' = sv$
q_3
$sv' = \text{AC}$
$nsnt' = nsnt + 1 \wedge sv' = \text{SE}$
$nsnt' = nsnt + 1 \wedge sv' = \text{AC}$
q_F

Figure 2.4: CFA of Broadcast Algorithm in [89] with clean crashes - Model 1

Non-clean crash faults [46]

With non-clean crash, a faulty process is allowed to send messages to all, none or a subset of processes, although they are not allowed to send malicious messages. Here our approach is to use two global variables, *nsntf* and *nfaulty*, to model the effect of faulty process on correct process. The variable *nfaulty* is used exactly as in model 2 of clean crash faults. *nsntf* is used to count the number of messages sent by the faulty processes, since they might send the messages to a subset of processes. It can be noticed that this is very similar to omission faults, the difference being the faulty process in case of a non-clean crash does not do anything after it crashes. With omission faults, the faulty process continues following the algorithm, but they might just miss sending some messages to a subset of processes.

In this case, *rcvd* can be incremented up to $nsnt + nsntf$, since *nsnt* models just

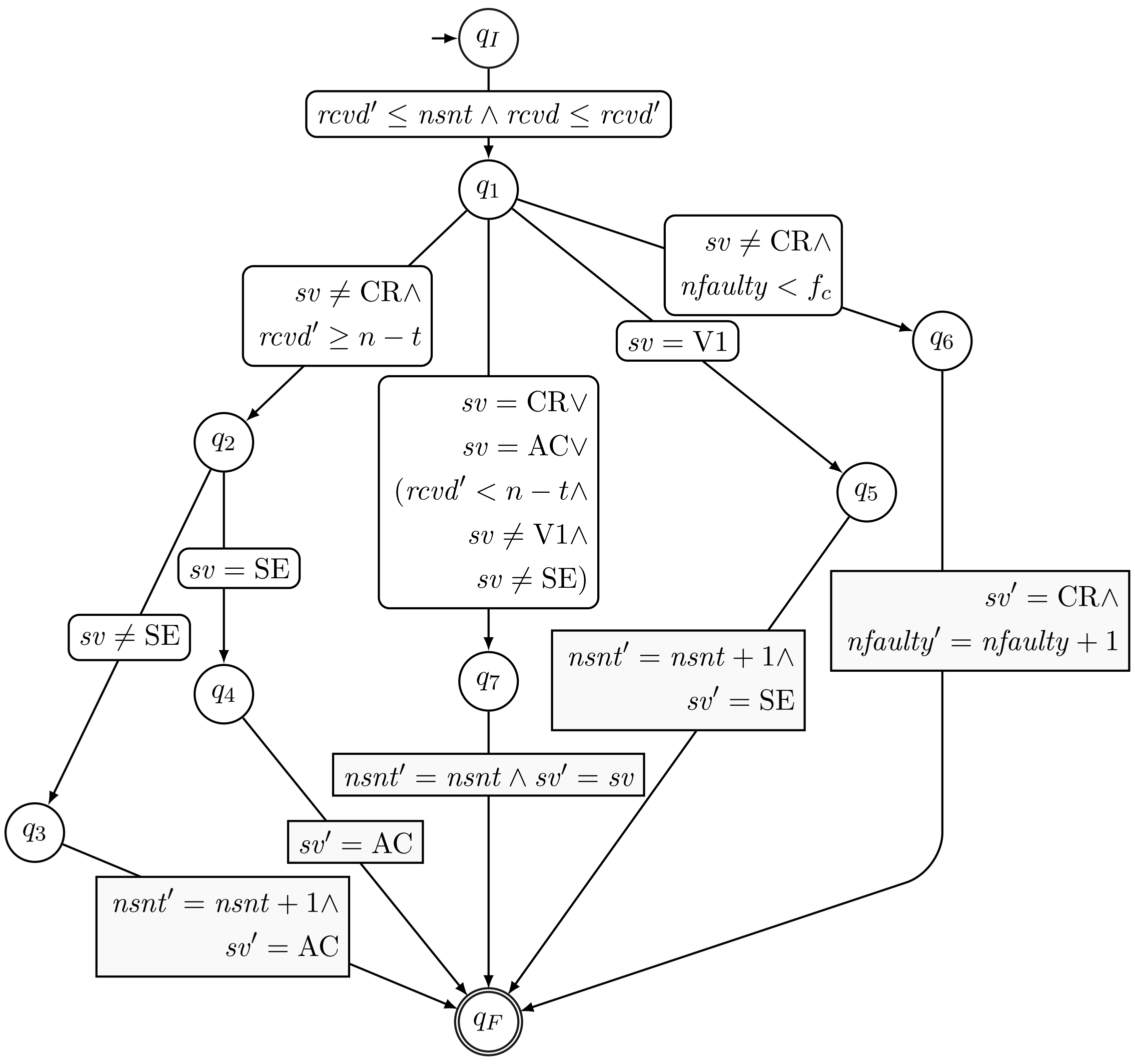

q_I
$rcvd' \leq nsnt \wedge rcvd \leq rcvd'$
q_1
$sv \neq \mathrm{CR} \wedge$ $rcvd' \geq n - t$
$sv = \mathrm{CR} \vee$ $sv = \mathrm{AC} \vee$ $(rcvd' < n - t \wedge$ $sv \neq \mathrm{V1} \wedge$ $sv \neq \mathrm{SE})$
$sv = \mathrm{V1}$
$sv \neq \mathrm{CR} \wedge$ $nfaulty < f_c$
q_2
q_5
q_6
$sv = \mathrm{SE}$
$sv \neq \mathrm{SE}$
q_4
q_7
q_3
$nsnt' = nsnt + 1 \wedge$ $sv' = \mathrm{SE}$
$sv' = \mathrm{CR} \wedge$ $nfaulty' = nfaulty + 1$
$nsnt' = nsnt \wedge sv' = sv$
$sv' = \mathrm{AC}$
$nsnt' = nsnt + 1 \wedge$ $sv' = \mathrm{AC}$
q_F

Figure 2.5: CFA of Broadcast Algorithm in [89] with clean crashes - Model 2

the messages sent by correct processes. The communication fairness is represented by the condition $\mathbf{G}\,\mathbf{F} \neg [\exists i.\ rcvd_i < nsnt]$ That is, for the communication to be fair every correct process must eventually receive all the messages sent by the correct processes. Processes may, but not necessarily, receive upto $nsnt + nsntf$ messages. Thus we model the behavior of the faulty processes where they send messages to subsets of processes or to none. We model the faulty processes such that, if they are in a state where they have to send a message (location q_3 and q_5 in Figure 2.6), they can either crash or send the message to every process. They may also crash in other locations without sending any messages.

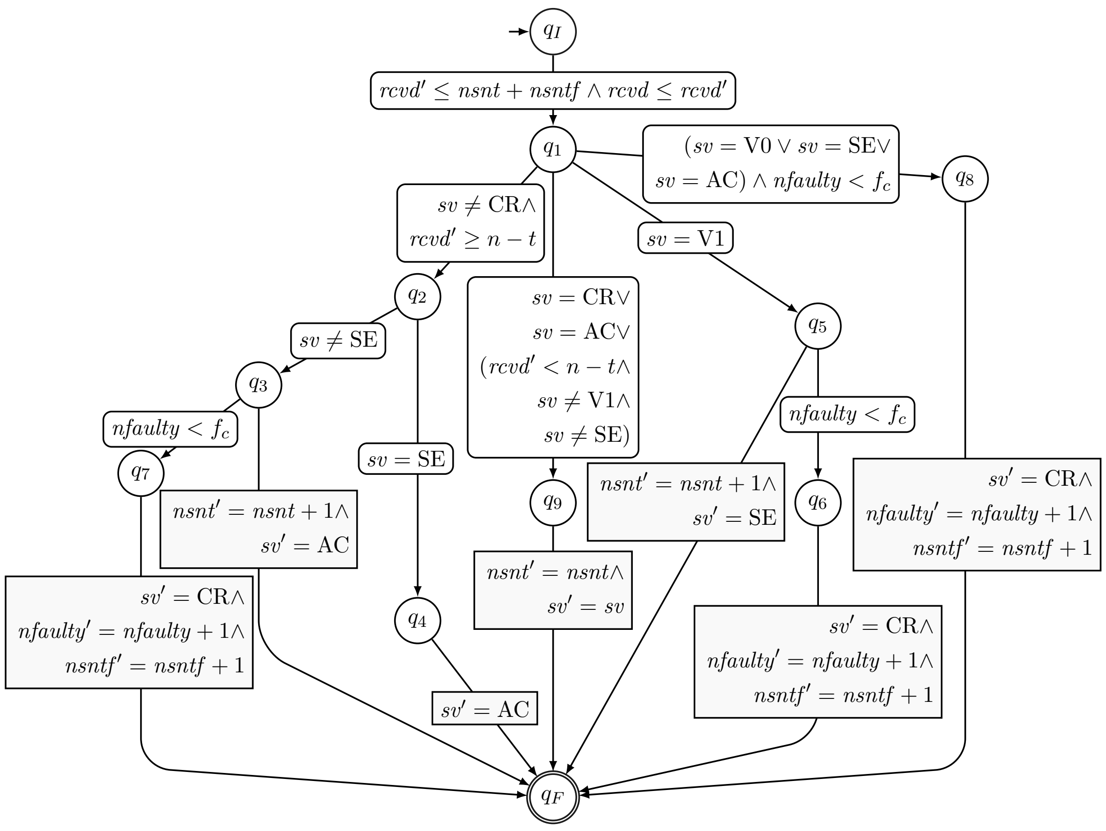


Figure 2.6: CFA of Broadcast Algorithm in [89] with non-clean crashes

2.5 Manual Translation of the pseudo code to Promela

Now we translate the pseudo code to PROMELA, which gives a parallel formalism in addition to the CFA representation. The PROMELA code for the algorithm closely follows the CFA representation. This step is necessary because it provides a formal representation of the algorithm which can be used by the SPIN model checker for practically verifying it.

As an example we translate Algorithm 2.1 into PROMELA code. In Algorithm 2.1 we consider *Byzantine* faults. We have also seen the CFA formalization of Algorithm 2.1 for other faults like omission, clean crash and symmetric faults in Section 2.4.4. Now we will explain how to formalize Algorithm 2.1 for each fault type mentioned above, in PROMELA code.

First of all, we present the extensions we added to PROMELA. Then, we explain our encoding of message passing for threshold-guarded fault-tolerant distributed algorithms. and encode the control flow of Algorithm 2.1. The rationale of the modeling decisions are that, the resulting PROMELA model

i captures the assumptions of distributed algorithms adequately, and ii allows efficient verification either using explicit state enumeration or by abstraction as discussed in Chapter 4.

2.5.1 Parametric extensions to Promela

We extend PROMELA such that we can express our CFA that uses unbounded and symbolic variables to express parameters. Let us see the different extensions we have incorporated in PROMELA in the examples given below:

The keyword *symbolic* is used to declare parameters. In our case these are n, t and f as explained in Section 1.2.2. The syntax is as follows:

```
symbolic int n, t, f;
```

To impose resilience conditions on the parameters we use the keyword *assume*. This keyword is ignored in explicit state model checking.

```
assume (n > 3*t);
```

Next, we show how to declare atomic propositions. We use the key word *atomic* for declaring atomic propositions that are unfolded into conjunctions over all processes (similarly for `some`).

Examples are:

```
/* there exists a process with sv = V0 */
atomic p = some(Proc:sv == V0);
/* sv = V1 for all processes */
atomic q = all(Proc:sv == V1);
atomic r = (nsnt > 1);
```

We can also use boolean combinations of atomic propositions as shown in the example below:

```
atomic p = some(Proc:sv == V0) ||
           all(Proc:sv == V1) && (nsnt > 1);
```

We can use the keyword *active* to create a number of processes, such that the number for processes is an expression over parameters as follows:

```
active [n-f] proctype  ProcName() {... }
```

Note that in standard PROMELA the number of processes created has to be a constant.

There is one more extension which we have developed, to be used with NuSMV and symbolic model checkers. We give a brief introduction to it, though it is not used for work done in this thesis:

```
x = 1;
havoc (x);
assume (x > 5);
y = x;
```

The example above has the *havoc()* function followed by *assume()*. This is used for the SSA representation [31] of the variable x in the code. That is, *havoc*(x) makes sure that x is not rewritten, by creating fresh variables. So the above code in effect does what is shown below:

```
$x^0 = 1$;
$x^1 > 5$;
$y^0 = x^1$;
```

2.5.2 Efficient encoding of message passing

In threshold-guarded distributed algorithms, the processes

i count how many messages of the same type they have received from *distinct* processes, and change their states depending on this number,

ii always send to *all* processes (including the process itself), and

iii send messages only for a fixed number of message types (only messages of type ⟨echo⟩ are sent in Algorithm 2.1).

Fault-free communication: From the discussion in Section 2.1 we observe that buffers are required to be unbounded, and thus sending is non-blocking. Further, receiving is non-blocking even if no message has been sent to the process. If we assume that for each message type, each correct process sends at most one message in each run (as in Algorithm 2.1), non-blocking send can be natively encoded in PROMELA using message channels. In principle, non-blocking receive also can be implemented in PROMELA, but it is not a basic construct.

Given the standard assumptions on the computational model, it is tempting to model communication in PROMELA using point-to-point message passing over FIFO channels. Now we discuss how to model such algorithms more efficiently than the straightforward implementation with PROMELA channels. Our approach captures both message passing and the effect of faults on the correct process. However, for the sake of comprehensibility let us first consider a fault-free system ($f = 0$). Later in the section we model different fault scenarios.

In the following examples of PROMELA code, we show a straightforward way to implement "received ⟨echo⟩ from at least x distinct processes" and "send ⟨echo⟩ to all" using PROMELA channels. We declare an array `p2p` of n^2 channels, one per pair of processes, and an array `rx` to keep track of the ⟨echo⟩ messages received from distinct processes (at most one ⟨echo⟩ message from a process j is received by a process i).

```
mtype = { ECHO }; /* one message type */
chan p2p[NxN] = [1] of { mtype }; /* channels of capacity 1 */
bit  rx[NxN]; /* a bit map to implement "distinct" */
active[N] proctype STBcastChan() {
  int i, nrcvd = 0; /* nr. of echoes */
```

Then, the receive code iterates over n channels. A process can choose to receive or not to receive an ⟨echo⟩ message from a non-empty channel and the empty channels are skipped. If a message is received, the corresponding channel is marked in the array `rx`:

```
    i = 0; do
      :: (i < N) && nempty(p2p[i * N + _pid]) ->
        p2p[i * N + _pid]?ECHO; /* retrieve a message */
        if
          :: !rx[i * N + _pid] ->
            rx[i * N + _pid] = 1; /* mark the channel */
            nrcvd++; break; /* receive at most one message */
          :: rx[i * N + _pid];    /* ignore duplicates */
        fi; i++;
      :: (i < N) ->
        i++;    /* channel is empty or postpone reception */
      :: i == N -> break;
    od
```

Finally, the sending code also iterates over n channels and sends on each:

```
    for (i : 1 .. N) { p2p[_pid * N + i]!ECHO; }
```

Recall that threshold-guarded algorithms have specific constraints:

i messages from all processes are processed uniformly,

ii messages carry only a message type without a process identifier, and

iii each process sends a message to all processes in no particular order.

This suggests a simpler modeling solution. Instead of using message passing directly, we keep only the numbers of sent and received messages in integer variables:

```
  int nsnt; /* one shared variable per a message type */
  active[N] proctype STBcast() {
    int nrcvd = 0, next_nrcvd = 0; /* nr. of echoes */
    ...
  step: atomic {
      if /* receive one more echo */
        :: (next_nrcvd < nsnt) ->
          next_nrcvd = nrcvd + 1;
        :: next_nrcvd = nrcvd; /* or nothing */
      fi;
      ...
      nsnt++; /* send echo to all */
    }
```

As one process step is executed atomically, concurrent reads and updates of *nsnt* are not a concern to us. Note that the presented code is based on the assumption that each correct process sends at most one message. We show how to enforce this assumption when discussing the control flow of our implementation of Algorithm 2.1 in Section 2.5.3.

Recall that in asynchronous distributed systems one assumes communication fairness, that is, every message sent is eventually received as given in RelComm. Hence we will add such fairness requirements to our specifications.

Faulty processes: Figure 2.7 shows how to model the different types of faults discussed above using channels. The implementations are direct consequences of the fault description given in Section 2.4.4.

```
active[F] proctype Byz() {
step: atomic {
  i = 0; do
   :: i < N -> sendTo(i); i++;
   :: i < N -> i++; /* some */
   :: i == N -> break;
  od
 }; goto step;
}

active[F] proctype Omit() {
step: atomic {
 /* receive as a correct */
 /* compute as a correct */
 if :: correctCodeSendsAll ->
   i = 0; do
    :: i < N -> sendTo(i); i++;
    :: i < N -> i++; /* omit */
    :: i == N -> break;
   od
   :: skip;
 fi
 if :: not(correctCodeSendsAll)
     ->
   goto crash;
 fi
 }; goto step;
}

active[F] proctype Symm() {
step: atomic {
  if
   :: /* send all */
     for (i : 1 .. N)
     { sendTo(i); }
   :: skip; /* or none */
  fi
 }; goto step;
}
```

```
active[F] proctype NonClean() {
step: atomic {
 /* receive as a correct */
 /* compute as a correct */
 if :: correctCodeSendsAll ->
   i = 0; do
       :: i < N -> sendTo(i); i++;
       :: i == N -> break;
      od
    :: correctCodeSendsAll ->
      i = 0; do
       :: i < N -> sendTo(i); i++;
       :: i < N -> i++; /* omit */
       :: i == N -> break;
       ::goto crash;
      od
 fi
 if :: !correctCodeSendsAll->
   goto crash;
 }; goto step;
}

active[F] proctype Clean() {
step: atomic {
 /* receive as a correct */
 /* compute as a correct */
 /* send as a correct one */
 };
 if
    :: goto step;
    :: goto crash;
 fi;
crash:
}
```

Figure 2.7: Modeling faulty processes explicitly: Byzantine (Byz), symmetric (Symm), omission (Omit), non-clean (NonClean) and clean crashes (Clean)

Figure 2.8 shows how can the impact of faults on correct processes be implemented in the shared memory implementation of message passing. Note that in contrast to Figure 2.7, the processes in Figure 2.8 are *not* the faulty ones, but correct ones whose variable `next_nrcvd` is subject to non-deterministic updates that correspond to the impact of faulty process. For instance, in the Byzantine case, in addition to the messages sent by correct processes, a process can receive up to f messages more. This is expressed by the condition `(next_nrcvd < nsnt + F)`.

For Byzantine and symmetric faults we only model correct processes explicitly. Thus, we specify that there are `N-F` processes in the system. Moreover, we can use Property (RelComm) to model reliable communication. However, we model omission and crash (clean and non-clean) faults explicitly. Thus we have `N` processes in these two cases. The impact of faulty processes is modeled by relaxed fairness requirements as explained in Section 2.4.4. Similar adaptations can be made to model, e.g., corrupted communication (e.g., due to faulty links) [87], or hybrid fault models [12] that contain different fault scenarios.

Figure 2.9 compares the number of states and memory consumption when modeling message passing using both solutions. We ran SPIN to perform exhaustive state enumeration on the encoding of Algorithm 2.1 (discussed in the next section). As can be seen, the model with explicit channels and faulty processes ran out of memory on *six* processes, whereas the shared memory model did so only with *nine* processes. Moreover, the latter scales better in the presence of faults, while the former degrades with faults. This leads us to use the shared memory encoding based on *nsnt* variables.

2.5.3 Encoding the control flow

Recall Algorithm 2.1, which is written in typical pseudo code found in the distributed algorithms literature, the details of which are given in Section 2.4.1.

We encoded the Algorithm 2.1 in Listing 3 using custom PROMELA extensions (see Section 2.5.1) to express notions of fault-tolerant distributed algorithms. The extensions are required to express a parameterized model checking problem and are used by our tool that implements the abstraction methods introduced in Chapter 4. When the parameters are fixed, these extensions are just syntactic sugar.

In the encoding in Listing 3, the whole step is captured within an atomic block (lines 20–42). As usual for fault-tolerant algorithms, this block has three logical parts: the receive part (lines 21–24), the computation part (lines 25–32), and the sending part (lines 33–38). We have already discussed the encoding of message passing above. Now we discuss the control flow of the algorithm.

Control state of the algorithm: Apart from receiving and sending messages, Algorithm 2.1 refers to several facts about the current control state of a process: "sent $\langle echo \rangle$ before", "if v_i", and "$accept_i \leftarrow$ TRUE". We capture all possible control states in a finite set SV. For instance, for Algorithm 2.1 we have the set of status values $SV = \{\mathrm{V0}, \mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$, where:

```
/* N > 3T ∧ T ≥ F ≥ 0 */
active[N-F] proctype ByzI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt + F)
    -> next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}

/* N > 2T ∧ T ≥ F ≥ 0 */
active[N] proctype OmitI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt) ->
    next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}

/* N > 2T ∧ T ≥ F_p ≥ F_s ≥ 0 */
active[N-Fp] proctype SymmI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt + Fs)
    -> next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

```
/* N ≥ T ∧ T ≥ F_c ≥ F_nc ≥ 0 */
active[N] proctype CleanI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt - Fnc)
     -> next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}

/* N ≥ T ∧ T ≥ F_c ≥ 0 */
active[N] proctype Clean2I() {
step: atomic {
  if
   :: (next_nrcvd < nsnt)
     -> next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}

/* N ≥ T ∧ T ≥ F_c ≥ 0 */
active[N] proctype NonCleanI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt+nsntf)
     -> next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

Figure 2.8: Modeling the effect of faults on correct processes: Byzantine (ByzI), symmetric (SymmI), omission (OmitI), clean crashes (CleanI and Clean2I) and non-clean crashes (NonCleanI).

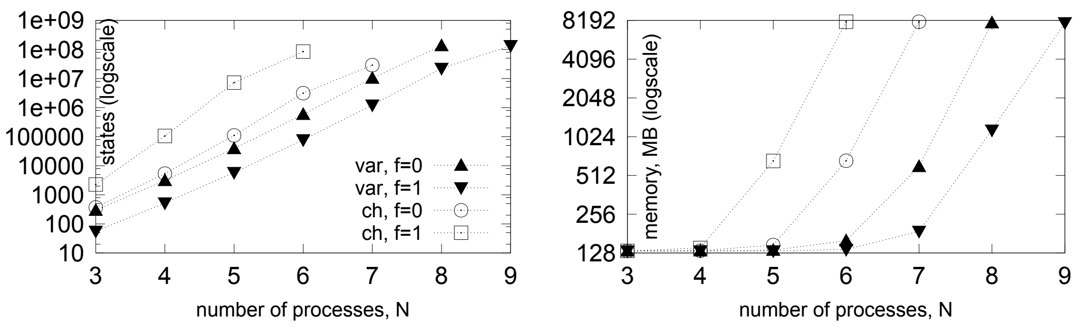


Figure 2.9: Visited states (left) and memory usage (right) when modeling message passing with channels (ch) or shared variables (var). The faults are in effect only when $f > 0$. Ran with `SAFETY`, `COLLAPSE`, `COMP`, and 8GB of memory.

- the status value V0 corresponds to $v_i =$ FALSE, $\text{accept}_i =$ FALSE and ⟨echo⟩ not sent.
- the status value V1 corresponds to $v_i =$ TRUE, $\text{accept}_i =$ FALSE and ⟨echo⟩ not sent.
- the status value SE corresponds to the case where $\text{accept}_i =$ FALSE and ⟨echo⟩ has been sent. Observe that once a process has sent ⟨echo⟩, its value of v_i does not interfere with the subsequent control flow anymore.
- the status value AC corresponds to the case where $\text{accept}_i =$ TRUE and ⟨echo⟩ has been sent. A process sets accept to TRUE only if it has sent a message (or is about to do so in the current step).

Thus, the control state is captured within a single *status variable sv* over SV with the set $SV_0 = \{\text{V0}, \text{V1}\}$ of initial control states.

Verification goal: Recall the resilience condition on the parameters for Algorithm 2.1 n, t, and f, namely, $n > 3t \land f \le t \land t > 0$. As these parameters do not change during a run, they can be encoded as constants in PROMELA. The verification problem for a distributed algorithm with fixed n and t is then the composition of model checking problems that differ in the actual value of f (satisfying $f \le t$).

2.6 Experiments

In this section we demonstrate the results of experiments on model checking of Algorithm 2.1 for fixed parameters using our ByMC tool chain. A brief description about the usage of the tool is given in Appendix A. The tool and the benchmarks are available at [1].

```
symbolic int N, T, F;  /* parameters  */
/* the resilience condition */
assume(N > 3 * T && T >= 1 && 0 <= F && F <= T);
int nsnt;  /* number of echoes sent by correct processes */
/* quantified atomic propositions */
atomic prec_unforg = all(STBcast:sv == V0);
atomic prec_corr = all(STBcast:sv == V1);
atomic prec_init = all(STBcaststep);
atomic ex_acc = some(STBcast:sv == AC);
atomic all_acc = all(STBcast:sv == AC);
atomic in_transit = some(STBcast:nrcvd < nsnt);

active[N - F] proctype STBcast() {
  byte sv, next_sv;          /* status of the algorithm */
  int nrcvd = 0, next_nrcvd = 0; /* nr. of echoes received */
  if  /* initialize */
    :: sv = V0; /* v_i = FALSE */
    :: sv = V1; /* v_i = TRUE  */
  fi;
step: atomic { /* an indivisible step */
    if /* receive one more echo (up to nsnt + F) */
      :: (next_nrcvd < nsnt + F) -> next_nrcvd = nrcvd + 1;
      :: next_nrcvd = nrcvd; /* or nothing */
    fi;
    if /* compute */
      :: (next_nrcvd >= N - T) ->
        next_sv = AC; /* accept_i = TRUE */
      :: (next_nrcvd < N - T && sv == V1
          || next_nrcvd >= T + 1) ->
        next_sv = SE; /* remember that <echo> is sent */
      :: else -> next_sv = sv; /* keep the status */
    fi;
    if /* send */
      :: (sv == V0 || sv == V1)
          && (next_sv == SE || next_sv == AC) ->
        nsnt++; /* send <echo>  */
      :: else;  /* send nothing */
    fi;
    /* update local variables and reset scratch variables */
    sv = next_sv; nrcvd = next_nrcvd;
    next_sv = 0; next_nrcvd = 0;
    } goto step;
}
/* LTL-X formulas */
ltl fairness { []<>(!in_transit) } /* added to other formulas */
ltl relay { [](ex_acc -> <>all_acc) }
ltl corr { []((prec_init && prec_corr) -> <>(ex_acc)) }
ltl unforg { []((prec_init && prec_unforg) -> []!ex_acc) }
```
Listing 3: Encoding of Algorithm 2.1 in parametric PROMELA.

#	parameter values	spec	valid	Time	Mem.	Stored	Transitions	Depth
				Byz				
B1	N=7,T=2,F=2	(U)	✓	3.13 sec.	74 MB	$193 \cdot 10^3$	$1 \cdot 10^6$	229
B2	N=7,T=2,F=2	(C)	✓	3.43 sec.	75 MB	$207 \cdot 10^3$	$2 \cdot 10^6$	229
B3	N=7,T=2,F=2	(R)	✓	6.3 sec.	77 MB	$290 \cdot 10^3$	$3 \cdot 10^6$	229
B4	N=7,T=3,F=2	(U)	✓	4.38 sec.	77 MB	$265 \cdot 10^3$	$2 \cdot 10^6$	233
B5	N=7,T=3,F=2	(C)	✓	4.5 sec.	77 MB	$271 \cdot 10^3$	$2 \cdot 10^6$	233
B6	N=7,T=3,F=2	(R)	✗	0.02 sec.	68 MB	$1 \cdot 10^3$	$13 \cdot 10^3$	210
				omit				
O1	N=5,To=2,Fo=2	(U)	✓	1.43 sec.	69 MB	$51 \cdot 10^3$	$878 \cdot 10^3$	175
O2	N=5,To=2,Fo=2	(C)	✓	1.64 sec.	69 MB	$60 \cdot 10^3$	$1 \cdot 10^6$	183
O3	N=5,To=2,Fo=2	(R)	✓	3.69 sec.	71 MB	$92 \cdot 10^3$	$2 \cdot 10^6$	183
O4	N=5,To=2,Fo=3	(U)	✓	1.39 sec.	69 MB	$51 \cdot 10^3$	$878 \cdot 10^3$	175
O5	N=5,To=2,Fo=3	(C)	✗	1.63 sec.	69 MB	$53 \cdot 10^3$	$1 \cdot 10^6$	183
O6	N=5,To=2,Fo=3	(R)	✗	0.01 sec.	68 MB	17	135	53
				symm				
S1	N=5,T=1,Fp=1,Fs=0	(U)	✓	0.04 sec.	68 MB	$3 \cdot 10^3$	$23 \cdot 10^3$	121
S2	N=5,T=1,Fp=1,Fs=0	(C)	✓	0.03 sec.	68 MB	$3 \cdot 10^3$	$24 \cdot 10^3$	121
S3	N=5,T=1,Fp=1,Fs=0	(R)	✓	0.08 sec.	68 MB	$5 \cdot 10^3$	$53 \cdot 10^3$	121
S4	N=5,T=3,Fp=3,Fs=1	(U)	✓	0.01 sec.	68 MB	66	267	62
S5	N=5,T=3,Fp=3,Fs=1	(C)	✗	0.01 sec.	68 MB	62	221	66
S6	N=5,T=3,Fp=3,Fs=1	(R)	✓	0.01 sec.	68 MB	62	235	62
				clean				
C1	N=3,Tc=2,Fc=2,Fnc=0	(U)	✓	0.01 sec.	68 MB	668	$7 \cdot 10^3$	77
C2	N=3,Tc=2,Fc=2,Fnc=0	(C)	✓	0.01 sec.	68 MB	892	$8 \cdot 10^3$	81
C3	N=3,Tc=2,Fc=2,Fnc=0	(R)	✓	0.02 sec.	68 MB	$1 \cdot 10^3$	$17 \cdot 10^3$	81

Table 2.1: Summary of experiments related to [90]

Listing 3 provides the central parts of the code of our case study. We implemented Algorithm 2.1 for different fault models as given below:

Byz: tolerates t Byzantine faults if $n > 3t$,

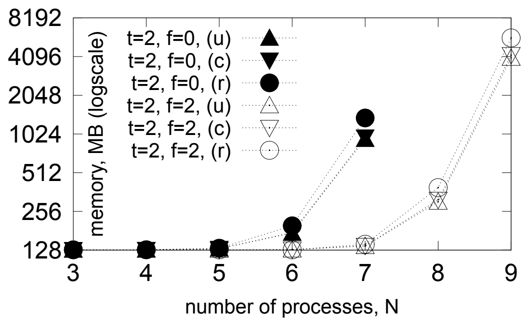
symm: tolerates t symmetric (identical Byzantine [8]) faults if $n > 2t$,

omit: tolerates t send omission faults if $n > 2t$,

clean: tolerates t clean crash faults for $n > t$.

The given resilience conditions on n and t are the ones we expected from the literature, and their tightness was confirmed by our experiments.

The major goal of the experiments was to check the adequacy of our formalization. To this end, we first considered the four well-understood variants of [90], for each of which we systematically changed the parameter values. We verify each variant with Spin. By doing so, we verify that under our modeling the different combination of parameters



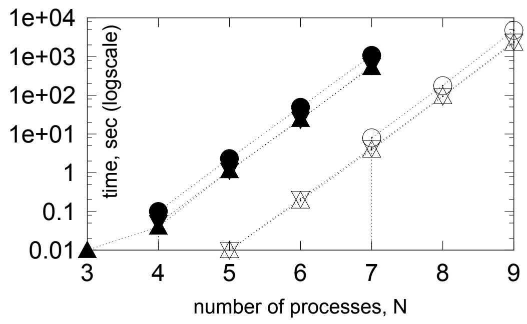


Figure 2.10: Spin memory usage (left) and running time (right) for Byz.

lead to the expected result. Table 2.1 and Figure 2.10 summarize the results of our experiments for broadcasting algorithms in the spirit of [90]. Lines B1 – B3, O1 – O3, S1 – S3, and C1 – C3 capture the cases for which the resilience condition is satisfied. In Lines B4 – B6, the algorithm's parameters are chosen to achieve a goal that is known to be impossible [80], i.e., to tolerate 3 faulty processes out of 7 processes. This violates the $n > 3t$ requirement. Our experiment shows that even if only 2 faults occur in this setting, the relay specification (R) is violated. In lines O4 – O6, the algorithm is designed properly, i.e., 2 out of 5 processes may fail ($n > 2t$ in the case of omission faults). Our experiments show that this algorithm fails in the presence of 3 faulty processes, i.e., (C) and (R) are violated.

We present experiments with other algorithms which covers different threshold conditions as well as fault models, with different combination of parameter values, in Chapter 5. Our experiments run out of memory for systems as big as those with $n = 11$. This shows the need for parameterized verification of such algorithms. Before moving on to the parameterized verification, in the next chapter, we will show that the parameterized model checking problem for FTDAs is an undecidable.

2.7 Related Work

Most of the distributed algorithms are described using pseudo code and natural language is used to specify the computational model, as can be seen in the seminal papers like [45, 33, 37, 19]. This makes the algorithms difficult to understand and reason about for non-experts and causes misunderstanding of the important subtleties of the algorithm.

There have been two major undertakings of formalization that gained acceptance within the distributed algorithms community. Both were initiated by researchers with a background in distributed algorithms and with a precise understanding of what needs to be expressed. These approaches are on the one hand, I/O Automata by Lynch and several collaborators [72, 62, 76], and on the other hand, TLA by Lamport and others [67, 60, 68]. IOA and TLA are general frameworks that are based on labeled transition systems and a variant of linear temporal logic, respectively. I/O Automata are used to model individual components of the distributed system, like each process and the communication medium. An automaton is capable of input and output actions. TLA is used for specifying and reasoning about concurrent systems and is a variation of temporal logic by Pnueli [82]. TLA+ is the specification language based on TLA. This approach is very general and can be used to specify a wide range of systems. In this approach both the algorithm and the properties are specified in the same logic.

Both frameworks were originally developed at a time when automated verification was out of reach, and they were mostly intended to be used as formal foundations for handwritten proofs. Today, the tool support for IOA is still in preliminary stages [2]. For TLA [3], the TLC model checker is a simple explicit state model checker, while the current version of the TLA+ Proof System can only check safety proofs.

In all these approaches, specifying the semantics for fault-tolerant distributed algorithms is a research challenge, and we believe that this research requires an interdisciplinary effort between researchers in distributed algorithms and model checking. In this chapter we presented our first results towards this direction.

CHAPTER 3

Undecidability of PMCP

In this chapter we prove the undecidability of the PMCP to motivate the development of our new abstraction method which facilitates the verification of threshold-based FTDAs. We show that the verification problem we have at hand is hard and thus it is impossible to have a general solution for it.

We discussed the modeling details of different kinds of faults in Chapter 2. In some cases, e.g., Byzantine faults, we model faults by the influence they have on values of variables in the domain D. As we do not restrict the set of local and global variables, the result also applies if these sets are non-empty. Moreover, in this kind of modeling, the atomic propositions $[\exists i.\ sv_i = Z]$ range over correct processes only. Hence, the undecidability result also holds for FTDAs. If we choose to model faults differently, i.e., by changing the transition relation of a process, then the decidability depends on the way the transition relation is modified.

In the previous chapter we discussed the different kind of guards used in our CFA, of which the status guards capture the control flow of the algorithm. We call a CFA with only status guards, a *non-communicating* CFA. We prove that the PMCP for CFA as given in Definition 4 is undecidable even if the CFA is non-communicating. As in [48, 42], our approach is to reduce the non-halting problem of *2-counter machines* to our problem.

3.1 2-Counter Machines

Counter machines or Minsky machines [75] consist of at least one unbounded counter encoded in unary representation. It allows three instructions on each counter: increment, decrement, and test for zero. The increment instruction increments the value of the corresponding counter, the decrement instruction decrements the corresponding counter value, and the test for zero instruction compares the value of a counter with zero. The counters can take only positive integer values. Thus, a counter is decremented only if it has a value greater than zero. A 2-counter machine (2CM) as the name implies, is a counter machine with two counters. With the help of some examples we will show that 2CMs are indeed capable of powerful operations on the counters.

Let us take the example given below:

```
$l_1$: if $C_2 = 0$ then goto $l_2$
   else $C_2--$; goto $l_3$
$l_3$: $C_1++$; goto $l_1$ ;

$l_2$: <halt>
```

Assume that initially $C_1 = a$ and $C_2 = b$. Then at location l_2 we have $C_1 = a + b$ and $C_2 = 0$. Thus, the code above adds the initial value of the counters and stores it in counter C_1. The operation destroys the value in counter C_2.

The example below performs a comparison operation between the two counters:

```
$l_1$: if $C_1 = 0$ then goto $l_2$
    else $C_1--$;  goto $l_3$

$l_2$: if $C_2 = 0$ then goto $l_4$
    else  goto  $l_5$

$l_3$: if $C_2 = 0$ then goto $l_5$
    else $C_2--$; goto $l_1$

$l_4$: <equal>
$l_5$: <not equal>
```

The above code checks if the two counters C_1 and C_2 are equal. Note that the operation destroys both the counter values.

As the final example, let us consider the implementation of an assignment operation $C_1 := C_2$ without destroying the counter C_2:

```
$l_1$: if $C_1 = 0$ then  goto $l_2$
        else $C_1--$; goto $l_1$

$l_2$: if $C_3 = 0$ then  goto $l_3$
        else $C_3--$; goto $l_2$

$l_3$: if $C_2 = 0$ then goto $l_4$
         else $C_2--$; $C_1++$; $C_3++$;  goto $l_3$

$l_4$: if $C_3 = 0$ then  goto $l_5$
         else $C_3--$; $C_2++$; goto $l_4$

$l_5$: <halt>
```

This example uses a temporary counter C_3 to restore the value of counter C_2 after the assignment operation.

3.1.1 Halting and non-halting problem for counter machines

Given a counter machine with m counters C_1, C_2, ... C_m such that, $C_1 = b_1$, $C_2 = b_2$, ... $C_m = b_m$, where b_1, b_2, ... b_m can be any non-negative integers, we define the *halting (P1) and non-halting (P2) problems* of counter machines as follows:

P1: For all b_1, b_2, ... b_m whether the counter machine halts.

P2: Whether there exist b_1, b_2, ... b_m such that the counter machine never halts.

Then we have the following theorems:

Theorem 5. *[75] The halting and non-halting problems for counter machines are undecidable for $m \geq 2$.*

Theorem 6. *[75] The halting and non-halting problems for counter machines are undecidable for $C_1 = 0$, $C_2 = 0$, ... $C_m = 0$.*

From Theorem 5 we know that the halting and the non-halting problems for 2CMs are undecidable.

3.2 Undecidability of Liveness Properties

In this section, we show that the non-halting problem of a 2-counter machine (2CM) is reducible to the parameterized model checking problem. We using a parameter n and a CFA $\mathcal{A}$ to construct a system instance $\mathsf{Inst}(n+1, \mathsf{Sk}(\mathcal{A}))$, an $\mathsf{LTL}_{\text{-}\mathsf{X}}$ formula φ_{nonhalt} that uses $\mathbf{G}$, $\mathbf{F}$, and atomic propositions $[\exists i.\ sv_i = Z]$. We then show that $\mathsf{Inst}(n+1, \mathsf{Sk}(\mathcal{A}))$ simulates at least n steps of the 2CM, and the general parameterized model checking problem is therefore undecidable. Note that $\mathbf{G}$, $\mathbf{F}$, and $[\exists i.\ sv_i = Z]$ are required to express the liveness property (R) of our use case Algorithm 2.1. Let us recall the specifications of Algorithm 2.1 presented in Chapter 2 below:

$$[\forall i.\ sv_i \neq \mathrm{V1}] \rightarrow \mathbf{G}\, [\forall j.\ sv_j \neq \mathrm{AC}] \tag{U}$$

$$[\forall i.\ sv_i = \mathrm{V1}] \rightarrow \mathbf{F}\, [\exists j.\ sv_j = \mathrm{AC}] \tag{C}$$

$$\mathbf{G}\, (\neg\, [\exists i.\ sv_i = \mathrm{AC}]) \vee \mathbf{F}\, [\forall j.\ sv_j = \mathrm{AC}] \tag{R}$$

Thus, we prove the following theorem:

Theorem 7. *Let $\mathcal{M}$ be a 2-counter machine, and (RC, N) be a natural pair of resilience condition and system size function. Then a non-communicating CFA $\mathcal{A}(\mathcal{M})$ and an $\mathsf{LTL}_{\text{-}\mathsf{X}}$ property $\varphi_{\text{nonhalt}}(\mathcal{M})$ can be efficiently constructed such that the following two statements are equivalent:*

- *$\mathcal{M}$ does not halt.*
- *$\forall \mathbf{p} \in \mathbf{P}_{RC}$, $\mathit{Inst}(\mathbf{p}, \mathit{Sk}(\mathcal{A})) \models_{\emptyset} \varphi_{\text{nonhalt}}(\mathcal{M})$.*

Corollary 8. *PMCP for CFAs is undecidable even if CFAs contain only status guards.*

The outline of the proof is similar to the undecidability proofs in [48, 42]: one of the processes plays the role of a *control process* that simulates the program of a two counter machine (2CM), and the other processes are *data processes* that each store at most one digit of one of the two counters encoded in unary representation. The control processes increments or decrements a counter by a handshake with a data process in a way that only one data process synchronizes with the control process at a time. In system instances that contain n data processes, this is sufficient to simulate n steps of a 2CM. If the parameterized model checking problem under consideration is defined for a natural (RC, N), then for arbitrarily many steps there is some system instance that simulates at least that many steps of the 2CM. Undecidability of the parameterized model checking problem then follows from the undecidability of the non-halting problem for 2CMs.

The CFA $\mathcal{A}$ contains the functionalities of a *control process* that simulates the program of the 2CM, as well as of *data processes* that each store at most one digit of one of the two counters B and C encoded in unary representation. For simplicity of presentation, we say that if a data process does not store a digit for B or C, then it stores a digit for a counter D. Counter D thus serves as a capacity (initially set to n), from which B and C can borrow (and return) digits, that is, initially the status variables of all data processes correspond to D. Our CFA $\mathcal{A}$ uses only the status variable sv, while the sets of local and global variables can be empty. We consider paths where exactly one process plays the role of the control process, and the remaining n processes are data processes. This can be encoded using G, F, and $[\exists i.\ sv_i = Z]$.

Esparza [43] has shown that test for zero statements of a 2CM can be simulated with a temporal logic specification using an atomic proposition "test for zero". Given the proof strategies from [48, 42], the only technical difficulty that remains is to ensure a handshake between non-communicating CFAs. We do so by enforcing a *handshake* using sequences of status values the CFAs go through, and an $\mathsf{LTL}_{\text{-}\mathsf{X}}$ specification which acts as a scheduler that ensures a specific interleaving between the updates of the status variables of the two CFAs. Intuitively, whenever the control process has to increase or decrease the value of a counter, this is done by a handshake of the control with a data process; up to this point, our proof follows ideas from [48, 42, 43]. In contrast to these papers, however, our system model does not provide primitives for such a handshake, which leads to the central contribution for our proof: we "move" this handshake into the specification without using the "next time" operator which is not present in $\mathsf{LTL}_{\text{-}\mathsf{X}}$. In addition, as in [43], also the test for zero is moved into the specification using our propositions $\neg[\exists i.\ sv_i = Z]$. We start with some preliminary definitions.

A *2-counter machine* (2CM) $\mathcal{M}$ is a list of $m+1$ statements over two counters B and C. A statement at location v uses a counter $\mathcal{C}(v) \in \{B, C\}$ and has one of the following forms (note, that the machine halts at location m):

$$v : \textbf{inc } \mathcal{C}(v);\ \textbf{goto } w \tag{3.1}$$
$$v : \textbf{if } \mathcal{C}(v) = 0 \textbf{ then goto } w' \tag{3.2}$$
$$\textbf{else dec } \mathcal{C}(v);\ \textbf{goto } w'' \tag{3.3}$$
$$m : \textbf{halt} \tag{3.4}$$

The control flow of the machine $\mathcal{M}$ is defined by the labeled graph $(\mathcal{V}, \mathcal{E}, \mathcal{C})$, where $\mathcal{V} = \{v \mid 0 \le v \le m\}$ is the set of locations, $\mathcal{E} = \mathcal{E}_+ \cup \mathcal{E}_0 \cup \mathcal{E}_- \cup \{m, m\}$ is the set of edges, and $\mathcal{C} : \mathcal{V} \to \{B, C\}$ is the labeling function which maps a location to the counter used in this location. The sets $\mathcal{E}_+$, $\mathcal{E}_0$, $\mathcal{E}_-$ are defined as follows:

- $\mathcal{E}_+ = \{(v, w) \mid$ statement at v goes to w as in (3.1)$\}$;
- $\mathcal{E}_0 = \{(v, w') \mid$ statement at v goes to w' as in (3.2)$\}$;
- $\mathcal{E}_- = \{(v, w'') \mid$ statement at v goes to w'' as in (3.3)$\}$.

Handshake: In what follows, we model a handshake between the control process and a data process in order to implement an increment as in (3.1) or decrement as in (3.3). The handshake is guaranteed by a combination of steps both in the control and the data processes and by a constraint formulated in $\mathsf{LTL}_{\mathsf{-X}}$ as follows.

We define the set SV_C of status values of the control process and the set SV_D be the set of status values of a data process:

$$SV_C = \bigcup_{v,w \in \mathcal{V}} \{(v, v, \mathit{IdlC}), (v, w, \mathit{SynC}), (v, w, \mathit{AckC}), (v, w, \mathit{WaitC})\}$$
$$SV_D = \bigcup_{x,y \in \{B,C,D\}} \{(x, x, \mathit{IdlD}), (x, y, \mathit{SynD}), (x, y, \mathit{AckD})$$

For each $(f, t, h) \in SV_C \cup SV_D$, f is the state before a handshake, t is the scheduled state after a handshake, and h is the status of the handshake.

Consider an edge $(v, w) \in \mathcal{E}_+$ and $x = D$ and $y = \mathcal{C}(v)$, that is, in location v the counter $\mathcal{C}(v)$ is incremented, and then the control goes to location w. Incrementing the counter is done by a handshake during which the control process goes from v to w, while a data process goes from D (the capacity) to $\mathcal{C}(v)$.

To do so, we construct two CFAs $J(v, w)$ and $I(x, y)$ shown in Figure 3.1: $J(v, w)$ goes from location v of $\mathcal{M}$ to location w in three steps $\mathit{SynC} \to \mathit{AckC} \to \mathit{WaitC} \to \mathit{IdlC}$, whereas $I(x, y)$ transfers one digit from counter x to counter y in steps $\mathit{SynD} \to \mathit{AckD} \to \mathit{IdlD}$. To actually enforce the handshake synchronization, we add the following formulas

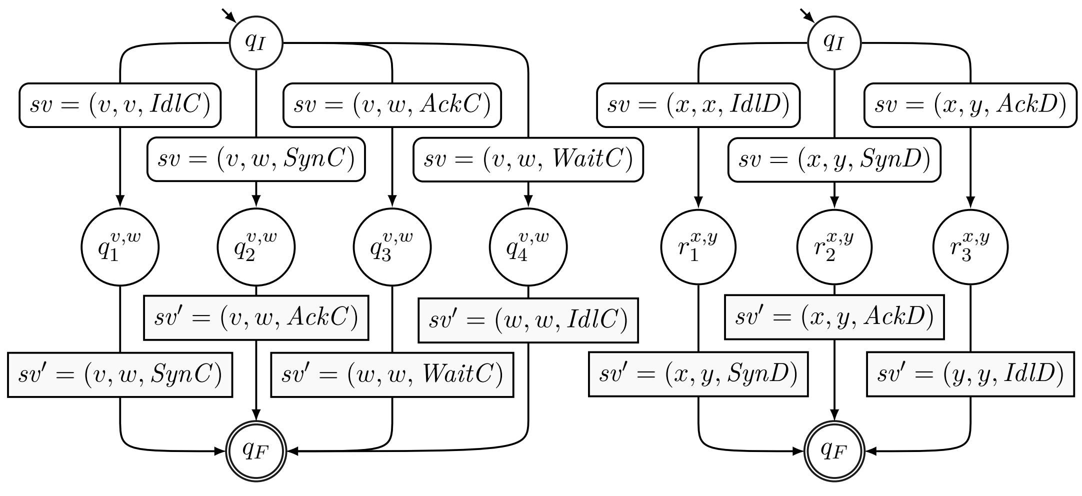


Figure 3.1: CFA $J(v, w)$ for $(v, w) \in \mathcal{E}_+$ and $I(x, y)$ for **inc** y (and **dec** x).

that must hold in every state of a system instance:

$$\begin{aligned}
[\exists k.sv_k = (x, y, SynD)] &\rightarrow ([\exists k.sv_k = (v, w, SynC)] \vee \\
&\qquad [\exists k.sv_k = (v, w, AckC)]) && (3.5)\\
[\exists k.sv_k = (x, y, AckD)] &\rightarrow \neg[\exists k.sv_k = (x, y, SynD)] && (3.6)\\
[\exists k.sv_k = (x, y, AckD)] &\rightarrow ([\exists k.sv_k = (v, w, AckC)] \vee \\
&\qquad [\exists k.sv_k = (v, w, WaitC)]) && (3.7)\\
[\exists k.sv_k = (w, w, IdlC] &\rightarrow (\neg[\exists k.sv_k = (x, y, SynD)] \wedge \\
&\qquad \neg[\exists k.sv_k = (x, y, AckD)]) && (3.8)\\
[\exists k.sv_k = (y, y, IdlD)] &\rightarrow (\neg[\exists k.sv_k = (v, w, AckC)] \vee \\
&\qquad [\exists k.sv_k = (x, y, SynD)] \vee \\
&\qquad [\exists k.sv_k = (x, y, AckD)]) && (3.9)
\end{aligned}$$

Let the handshaking formula $HS(v, w, x, y)$ be the conjunction of the formulas (3.5)-(3.9). Then $HS(v, w, x, y)$ must hold in all states of a system instance.

In what follows, we will consider the union of CFAs, where union is defined naturally as the union of the sets of nodes and the union of the sets of edges (note that the CFAs are joint at the initial node q_I, and the final node q_F).

We are now ready to prove the central result: Let M be a system of K processes $\mathsf{Inst}(K, \mathsf{Sk}(J(v, w) \cup I(x, y)))$ and σ_1 be a global state of M such that

$$\begin{aligned}
&\sigma_1[1].sv = (v, v, IdlC) \text{ and}\\
&\sigma_1[k].sv = (x, x, IdlD) \text{ for all } 2 \leq k \leq K.
\end{aligned}$$

The constraints (3.5)-(3.9) impose a synchronization behavior:

Proposition 9. *Let π be an infinite path $\{\sigma_i\}_{i\geq 1}$ of M starting with σ_1. If $\pi \models$ $\boldsymbol{G}\,HS(v,w,x,y)$, then there exists an index ℓ such that $2 \leq \ell \leq K$ and the following prefix of states holds true for all executions:*

	step	$\sigma_i[1]$	$\sigma_i[\ell]$	$\sigma_i[k],\ k \neq \ell$
σ_1	*1*	(v, v, IdlC)	(x, x, IdlD)	(x, x, IdlD)
σ_2	ℓ	(v, w, SynC)	(x, x, IdlD)	(x, x, IdlD)
σ_3	*1*	(v, w, SynC)	(x, y, SynD)	(x, x, IdlD)
σ_4	ℓ	(v, w, AckC)	(x, y, SynD)	(x, x, IdlD)
σ_5	*1*	(v, w, AckC)	(x, y, AckD)	(x, x, IdlD)
σ_6	ℓ	$(v, w,\ \mathit{WaitC})$	(y, y, AckD)	(x, x, IdlD)
σ_7	*1*	$(v, w,\ \mathit{WaitC})$	(y, y, IdlD)	(x, x, IdlD)
σ_8		(w, w, IdlC)	(y, y, IdlD)	(x, x, IdlD)

Proof. Now, we show that the prefix $\sigma_1, \ldots \sigma_8$ is as given above for some process ℓ such that $2 \leq \ell \leq K$. We show that other possible executions contradict the proposition's hypothesis. Recall that a single step of a process corresponds to a path of its CFA from q_I to q_F. We start with σ_1 and show that the only possible step that can be taken from σ_1 and each succeeding state in the prefix is as shown in the Proposition 9:

State σ_1. The infinite path starts with σ_1 by the hypothesis of the proposition.

State σ_2. By (3.5), data processes are blocked until the control process initiates the handshake with *SynC*. Thus, only process 1 can take a step in σ_1.

State σ_3. By (3.9), the control process is blocked in σ_2. Hence, there is some data process ℓ which takes the step.

State σ_4. In σ_3 process ℓ is blocked by (3.7). Further, suppose by contradiction that a data process $k \neq \ell$ takes a step resulting in $\sigma'_4[k] = (x, y, \mathit{SynD})$. Due to (3.6), processes ℓ and k are blocked in σ'_4. It follows that in the path starting with σ'_4, all data processes can move from (x, x, IdlD) to (x, y, SynD). As we have finitely many processes, eventually all data processes will stop at (x, y, SynD). Let σ'_x be the state where all the data process have moved to (x, y, SynD). Then, once all the data process have moved to (x, y, SynD), process 1 can take a step and move to (v, w, AckC) and then to $(v, w,\ \mathit{WaitC})$ as shown below:

	step	$\sigma_i[1]$	$\sigma_i[\ell]$	$\sigma_i[k],\ k \neq \ell$
σ'_x	1	(v, w, SynC)	(x, y, SynD)	(x, x, SynD)
σ'_{x+1}	1	(v, w, AckC)	(y, y, SynD)	(x, x, SynD)
σ'_{x+2}		$(v, w,\ \mathit{WaitC})$	(y, y, SynD)	(x, x, SynD)

In σ'_x and σ'_{x+1} all the data processes are blocked by (3.6). In σ'_{x+2}, process 1 is blocked by (3.8) and all the data processes are blocked by (3.6). Thus, no process can take a step in σ'_{x+2}. This contradicts the assumption that π is infinite. Thus, only process 1 can take a step in σ_3.

State σ_5. In σ_4, every data process $k \neq \ell$ is blocked by the same argument as in state σ_3 (they all eventually group and are blocked in (x, y, SynD)). Process 1 can move to

$\sigma'_5[k] = (x, y, \mathit{WaitC})$, but after that it is blocked due to (3.8), which contradicts the assumption that π is infinite. Thus, only process ℓ can take a step in σ_4.

State σ_6. In σ_5, every data process $k \neq \ell$ is blocked due to (3.6) and process ℓ is blocked due to (3.9). Thus, only process 1 can take a step in σ_5.

State σ_7. In σ_6, every data process $k \neq \ell$ is blocked due to (3.5) and process 1 is blocked due to (3.8). Thus, only process ℓ can take a step in σ_6.

State σ_8. In σ_7, all the data processes are blocked due to (3.5). Thus, only process 1 can take a step. □

Now we can prove Theorem 7 as follows:

Proof. Using $J(v,w)$ and $I(x,y)$ we can simulate (3.1) for $(v,w) \in \mathcal{E}_+$ by instantiating $J(v,w)$ and $I(D, \mathcal{C}(v))$. Moreover, we can simulate (3.3) for $(v,w'') \in \mathcal{E}_-$ by instantiating $J(v,w'')$ and $I(\mathcal{C}(v), D)$. Finally, we can simulate (3.2) for $(v,w') \in \mathcal{E}_0$ (that is, the test for zero) by instantiating $J(v,w')$ and adding one more temporal constraint:

$$[\exists k. sv_k = (v, w', \mathit{SynC})] \rightarrow \neg[\exists k. sv_k = (\mathcal{C}(v), \mathcal{C}(v), \mathit{IdlD})] \tag{EQ0}$$

Now we can construct the CFA $\mathcal{A}(\mathcal{M})$ that simulates $\mathcal{M}$. This CFA is a union of CFAs constructed for edges of $\mathcal{E}$. If

$$Q_J = \bigcup_{(v,w)\in\mathcal{E}} J(v,w),$$

then

$$\mathcal{A}(\mathcal{M}) = Q_J \cup \bigcup_{(v,w)\in\mathcal{E}_+} I(D, \mathcal{C}(v)) \cup \bigcup_{(v,w)\in\mathcal{E}_-} I(\mathcal{C}(v), D).$$

Further we define a specification which ensures that there is always exactly one control process as

$$\bigwedge_{q\in Q_J} \bigwedge_{q'\in Q_J\setminus\{q\}} \neg[\exists k.\ sv_k = q] \vee \neg[\exists k.\ sv_k = q'] \tag{CP}$$

Now, we can specify the non-halting property φ_{nonhalt} as follows:

$$\mathbf{G}\Big(\neg[\exists k. sv_k = (m, m, \mathit{IdlC})]\Big) \vee \mathbf{F}\,\neg\Big(\mathit{CP} \wedge \bigwedge_{(v,w)\in\mathcal{E}_0} \mathit{EQ0} \wedge \bigwedge_{(v,w)\in\mathcal{E}_+\cup\mathcal{E}_-} \mathit{HS}(v, w, \mathcal{C}(v), \mathcal{C}(v))\Big)$$

We can specify two initial process states: One is where the process stays in $sv_C = (\ell_0, \ell_0, \mathit{IdlC})$ and another one is where the process $sv_D = (D, D, \mathit{IdlD})$. Then $SV_0 = \{sv_C, sv_D\}$. This concludes the proof. □

3.3 Related Work

The decidability of model checking of distributed systems under different link semantics is studied in [5, 18]. Although several approaches have been made to identify decidable classes for parameterized verification [42, 39, 97], no decidable formalism has been found which covers a reasonably large class of interesting problems. The diversity of problem domains for parameterized verification and the difficulty of the problem gave rise to many approaches including regular model checking [4] and abstraction [83, 29] — the method discussed the following chapter. The challenge in abstraction is to find an abstraction $h(\mathbf{K})$ such that $h(\mathbf{K}) \models \varphi$ implies $K_i \models \varphi$ for *all* i.

Notwithstanding this undecidability results, the rest of the thesis is concerned with abstraction techniques for threshold-based FTDAs. In this context, Corollary 8 shows that the model checking problem that we obtain after the first abstraction step (mentioned in the introduction) is still undecidable. As discussed in the introduction, abstraction always has to be accompanied by a case study along with practical experiments.

CHAPTER 4

Abstraction Scheme

After dealing with the formalization problem in Chapter 2, in this chapter we deal with the second challenge faced in verifying FTDAs: the verification problem. Here we present our Parameterized Interval Abstraction (PIA) method which was sketched in Section 1.4.2 and demonstrate how it is used to reduce a parameterized model checking problem to a finite-state model checking problem. As explained in Section 1.3, our verification problem involves a family of concrete systems with unbounded states. We use two levels of abstraction to abstract away this family of concrete systems to an abstract finite state system: Parameterized Interval Data Abstraction (PIA data abstraction) and Parameterized Interval Counter Abstraction (PIA counter abstraction).

Parameterized model checking requires us to ensure that the specifications to be checked are satisfied by the family of concrete systems with all possible combinations of parameter values. Hence, our abstract system must contain all the concrete system behaviors. Consequently, we use existential abstraction which ensures that if there exists a concrete run in a concrete system, it is mapped to a run in the abstract system. Thus, if a specification is violated in a concrete system, it is also violated in the abstract system. That is, in the other direction, if a specification holds in the abstract system, then it holds in *all* concrete systems. Such an abstraction method is said to be *sound*.

Usually abstractions introduce new behavior that is not present in the original system. Thus, a finite-state model checker might find a spurious run, that is, a run that does not exist in the concrete system, while checking the abstract system. In order to discard such runs, abstraction refinement techniques [25] have to be developed.

An overview of our abstraction scheme is given in Figure 4.1. In the sections that follow, we will explain in detail both the abstraction steps involved, the abstraction refinement step and also show why our abstraction method is sound.

Throughout this chapter we use the Algorithm 2.1 as our running example. Its encoding in parametric PROMELA is given in Listing 3. Our final goal is to obtain a PROMELA program that we can verify in SPIN model checker. We will begin with our case study, introduce the formalism and show how to abstract the parametric PROMELA code

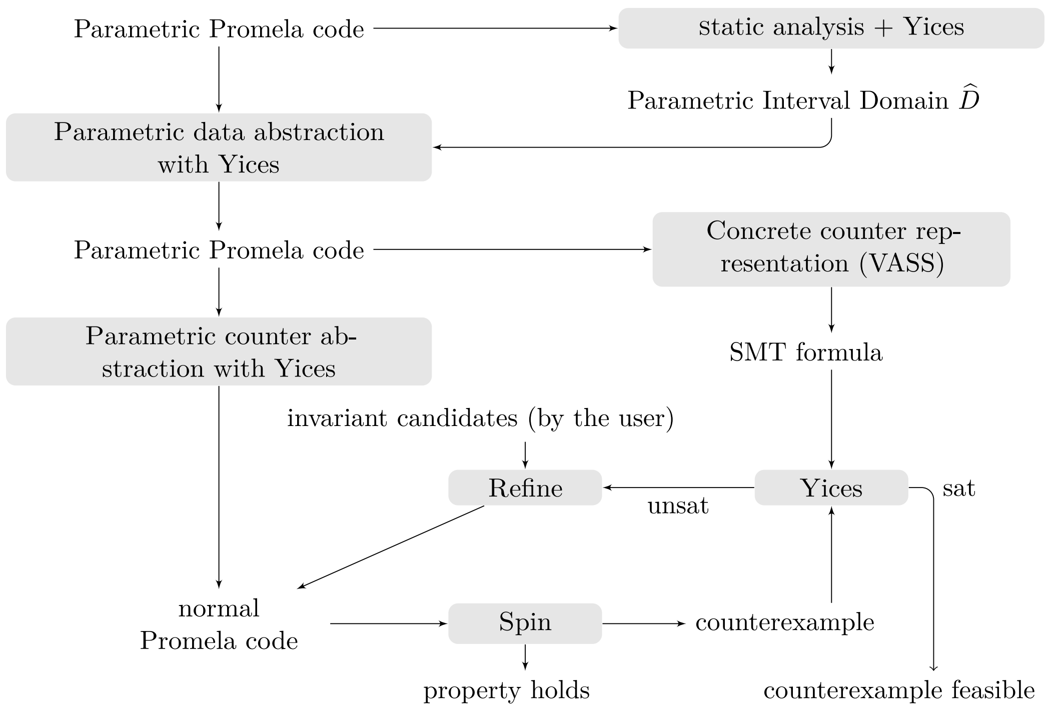

Parametric Promela code
static analysis + Yices
Parametric Interval Domain $\widehat{D}$
Parametric data abstraction with Yices
Parametric Promela code
Concrete counter representation (VASS)
Parametric counter abstraction with Yices
SMT formula
invariant candidates (by the user)
Refine
unsat
Yices
sat
normal Promela code
Spin
counterexample
property holds
counterexample feasible

Figure 4.1: The abstraction scheme

given in Listing 3 to a program in standard PROMELA [55]. We also present experiments with SPIN where we verify different parameterized algorithms in order to validate our abstraction method.

4.1 Abstract Domain for Parametric Intervals (PIA)

In order to gain an intuition about our abstract domain, let us begin with the code in Listing 3. The process prototype `STBcast` refers to two kinds of variables, each of them having a special role:

- *Bounded variables.* These are local variables that range over a finite domain, the size of which is independent of the parameters. In our example, `sv` and `next_sv` are variables of this kind.
- *Unbounded variables.* These variables range over an unbounded domain. They may be local or shared. In our example, the variables `nrcvd`, `next_nrcvd`, and `nsnt` are unbounded. These variables might become bounded, if we fix the parameters, as in our example with `nsnt` $\leq n - f$. However, we need a finite representation independent of the parameters, that is, the bounds on the variable values must be independent of the parameter values.

We can partition the variables into the sets B (bounded) and U (unbounded) by performing value analysis on the process body. Intuitively, one can imagine that the analysis iteratively computes the set B of variables that are assigned their values using two kinds of statements:

- An assignment that copies a constant expression to a variable;
- An assignment that copies the value of another variable, which already belongs to B.

The variables outside of B, e.g., those that are incremented in the code, belong to U. This can be done by a simple implementation of abstract interpretation [30].

The data abstraction deals with unbounded variables by turning the operations over unbounded domains into operations over finite domains. Threshold-based fault-tolerant distributed algorithms give us a natural source of abstract values: the threshold expressions. In our example, the variable `next_nrcvd` is compared against thresholds $t+1$ and $n-t$. Thus, it appears natural to forget about concrete values of `next_nrcvd`. Let us first try to replace the expressions that involve `next_nrcvd` with the expressions over the two predicates: `p1_next_nrcvd` $\equiv x < t+1$ and `p2_next_nrcvd` $\equiv x < n-t$. Then, the following code is an abstraction of the computation block in lines (25)–(32) of Listing 3:

```
if /* compute */
  :: (!p2_next_nrcvd) -> next_sv = AC;
  :: (!p2_next_nrcvd && (sv == V1 || !p1_next_nrcvd)) ->
    next_sv = SE;
  :: else -> next_sv = sv;
fi;
```

Listing 4: Predicate abstraction of the computation block

However, our modeling involves operations like comparison of two variables, incrementing the value of a variable etc. as can be seen in the code line 22 of Listing 3. Since such notions are not naturally expressed in terms of predicate abstraction, we introduce parametric interval abstraction PIA. In this abstraction the abstract domain represents intervals. The boundaries of these intervals are expressions to which the unbounded variables are compared to, e.g., $t+1$ and $n-t$. We then use an SMT solver (YICES) to abstract the expressions involving unbounded variables, e.g., comparisons.

Thus, instead of using several predicates, we replace the concrete domain of every variable $x \in U$ in Listing 3, with the abstract domain $\{I_0, I_{t+1}, I_{n-t}\}$. For reasons that are motivated by the counter abstraction, we have to distinguish value 0 from a positive value. This will be introduced later in Section 4.3. So we extend the domain with the threshold "1", that is, $\widehat{D} = \{I_0, I_1, I_{t+1}, I_{n-t}\}$.

Semantics of the abstract domain: We introduce an abstract version of x, denoted by $\hat{x}$. The values of $\hat{x}$ (from $\widehat{D}$) relate to the concrete values of x as follows:

$$\hat{x} = \begin{cases} I_0 & \text{iff } x \in [0;1[\\ I_1 & \text{iff } x \in [1;t+1[\\ I_{t+1} & \text{iff } x \in [t+1;n-t[\\ I_{n-t} & \text{iff } x \in [n-t;\infty[\end{cases} \tag{4.1}$$

Now we translate the computation block in lines (25)–(32) of Listing 3 as follows:

```
if /* compute */
 :: next_nrcvd == $I_{n-t}$ -> next_sv = AC;
 :: (next_nrcvd == $I_0$ || next_nrcvd == $I_1$ || next_nrcvd== $I_{t+1}$)
   && (sv == V1 || (next_nrcvd == $I_{t+1}$ || next_nrcvd == $I_{n-t}$))
   -> next_sv = SE;
 :: else -> next_sv = sv;
fi;
```

Listing 5: Parametric interval abstraction of the computation block

The abstraction of the receive block (lines 21–24 of Listing 3) involves the assignment `next_nrcvd = nrcvd + 1`. This becomes a *non-deterministic choice* of the abstract value of `next_nrcvd`, since it depends on the abstract value of `nrcvd`. Intuitively, `next_nrcvd` could be in the same interval as `nrcvd` or in the interval above. We provide the abstraction of lines 21–24 below. Later in this chapter, we will discuss how this abstraction can be computed using an SMT solver.

```
  if /* receive */
    :: (/* abstraction of (next_nrcvd < nsnt + F) */) ->
    if :: nrcvd == $I_0$ -> next_nrcvd = $I_1$;
        :: nrcvd == $I_1$ -> next_nrcvd = $I_1$;
        :: nrcvd == $I_1$ -> next_nrcvd = $I_{t+1}$;
        :: nrcvd == $I_{t+1}$ -> next_nrcvd = $I_{t+1}$;
        :: nrcvd == $I_{t+1}$ -> next_nrcvd = $I_{n-t}$;
        :: nrcvd == $I_{n-t}$ -> next_nrcvd = $I_{n-t}$;
    fi;
    :: next_nrcvd = nrcvd;
  fi;
```

Listing 6: Parametric interval abstraction of the receive block

There are several interesting consequences of transforming the receive block as above. First, due to our resilience condition (which ensures that intervals do not overlap) for every value of `nrcvd` there are at most two values that can be assigned to `next_nrcvd`. For instance, if `nrcvd` equals I_{t+1}, then `next_nrcvd` becomes either I_{t+1}, or I_{n-t}. Second, due to non-determinism, the assignment is not anymore guaranteed to reach any value, e.g., `next_nrcvd` might be always assigned value I_1.

Formalization: The input to our abstraction method is the infinite parameterized family $\mathcal{F} = \{\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \mid \mathbf{p} \in \mathbf{P}_{RC}\}$ of Kripke structures specified via a CFA $\mathcal{A}$. The family $\mathcal{F}$ has two principal sources of unboundedness: unbounded variables in the process skeleton $\mathsf{Sk}(\mathcal{A})$, and the unbounded number of processes $N(\mathbf{p})$. We deal with these two aspects separately, using two abstraction steps, namely the *PIA data abstraction* and the *PIA counter abstraction*. In both abstraction steps we use the parametric interval abstraction PIA.

Given a CFA $\mathcal{A}$, let $\mathcal{G}_\mathcal{A}$ be the set of all linear combinations $a_0 + \sum_{1 \le i \le |\Pi|} a_i \cdot p_i$ in the left-hand sides of $\mathcal{A}$'s threshold guards (see Section 2.3). Every expression ε of $\mathcal{G}_\mathcal{A}$ defines a function $f_\varepsilon \colon \mathbf{P}_{RC} \to D$. Let $\mathcal{T} = \{0,1\} \cup \{f_\varepsilon \mid \varepsilon \in \mathcal{G}_\mathcal{A}\}$ be a finite *threshold set*, and $\mu + 1$ its cardinality. For convenience, we name elements of $\mathcal{T}$ as $\vartheta_0, \vartheta_1, \dots, \vartheta_\mu$ with ϑ_0 corresponding to the constant 0, and ϑ_1 corresponding to the constant 1. For example, the CFA in Fig. 2.1 has the threshold set $\{\vartheta_0, \vartheta_1, \vartheta_2, \vartheta_3\}$, where $\vartheta_2(n,t,f) = t+1$ and $\vartheta_3(n,t,f) = n - t$. Then, we define the domain of parametric intervals as:

$$\widehat{D} = \{I_j \mid 0 \le j \le \mu\}$$

Intuitively, as in [83], I_0 reflects an existential quantifier. However, in our case, an abstract domain which distinguishes between 0, 1, and *more* [83] is too coarse to track whether variables have surpassed certain thresholds.

Our abstraction rests on an implicit property of many FTDAs, namely, that the resilience condition RC induces an order on the thresholds used in the algorithm (e.g., $t + 1 < n - t$). We may thus restrict the threshold sets we consider by the following definition:

Definition 10. *The finite set $\mathcal{T}$ is uniformly ordered if for all $\mathbf{p} \in \mathbf{P}_{RC}$, and all $\vartheta_j(\mathbf{p})$ and $\vartheta_k(\mathbf{p})$ in $\mathcal{T}$ with $0 \le j < k \le \mu$, it holds that $\vartheta_j(\mathbf{p}) < \vartheta_k(\mathbf{p})$.*

In cases where only a partial order is induced by RC, one can simply enumerate all finitely many total orders. As parameters, and thus thresholds, are kept unchanged in a run, one can verify an algorithm for each threshold order separately, and then combine the results.

Definition 10 allows us to properly define the *parameterized abstraction function* $\alpha_\mathbf{p} \colon D \to \widehat{D}$ and the *parameterized concretization function* $\gamma_\mathbf{p} \colon \widehat{D} \to 2^D$.

$$\alpha_\mathbf{p}(x) = \begin{cases} I_j & \text{if } x \in [\vartheta_j(\mathbf{p}), \vartheta_{j+1}(\mathbf{p})[\text{ for some } 0 \le j < \mu \\ I_\mu & \text{otherwise.} \end{cases}$$

$$\gamma_\mathbf{p}(I_j) = \begin{cases} [\vartheta_j(\mathbf{p}), \vartheta_{j+1}(\mathbf{p})[& \text{if } j < \mu \\ [\vartheta_\mu(\mathbf{p}), \infty[& \text{otherwise.} \end{cases}$$

From $\vartheta_0(\mathbf{p}) = 0$ and $\vartheta_1(\mathbf{p}) = 1$, it immediately follows that for all $\mathbf{p} \in \mathbf{P}_{RC}$, we have $\alpha_\mathbf{p}(0) = I_0$, $\alpha_\mathbf{p}(1) = I_1$, and $\gamma_\mathbf{p}(I_0) = \{0\}$. Moreover, from the definitions of α, γ, and Definition 10 one immediately obtains:

Proposition 11. *For all $\mathbf{p}$ in $\mathbf{P}_{RC}$, and for all a in D, it holds that $a \in \gamma_\mathbf{p}(\alpha_\mathbf{p}(a))$.*

Definition 12. *We define comparison between parametric intervals I_k and I_ℓ as $I_k \leq I_\ell$ iff $k \leq \ell$.*

The PIA domain has similarities to predicate abstraction since the interval borders are naturally expressed as predicates, and computations over PIA are directly reduced to SMT solvers. However, notions such as the order of Definition 12 are not naturally expressed in terms of predicate abstraction.

4.2 PIA data abstraction

Our parameterized data abstraction is based on two abstraction ideas. First, the variables used in a process skeleton are unbounded and we have to map these unbounded variables to a finite domain. If we fix parameters $\mathbf{p} \in \mathbf{P}_{RC}$, then an interval abstraction [30] is a natural solution to the problem of unboundedness. Second, we want to produce a single process skeleton that does not depend on parameters $\mathbf{p} \in \mathbf{P}_{RC}$ and captures the behavior of *all* process instances. This can be done by using ideas from existential abstraction [26, 32, 63] and sound abstraction of fairness constraints [63]. Our contribution consists of *combining these two ideas* to arrive at parametric interval data abstraction.

Our abstraction maps values of unbounded variables to parametric intervals I_j, whose boundaries are symbolic expressions over parameters. However, for every instance, the boundaries are *constant* because the parameters are fixed.

We now discuss an existential abstraction of a formula Φ that is either a threshold or a comparison guard (we consider other guards later). To this end, we introduce notation for sets of vectors satisfying Φ. Formula Φ has two kinds of free variables: parameter variables from Π and data variables from $\Lambda \cup \Gamma$. Let $\mathbf{x}^p$ be a vector of parameter variables $(x^p_1, \dots, x^p_{|\Pi|})$ and $\mathbf{x}^v$ be a vector of variables $(x^v_1, \dots, x^v_k)$ over D^k. Given a k-dimensional vector $\mathbf{d}$ of values from D, by

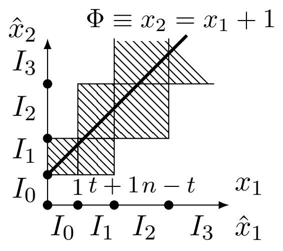
$$\mathbf{x}^p = \mathbf{p}, \mathbf{x}^v = \mathbf{d} \models \Phi$$

we denote that Φ is satisfied on concrete values $x^v_1 = d_1, \dots, x^v_k = d_k$ and parameter values $\mathbf{p}$. Then, we define:

$$\begin{aligned} ||\Phi||_\exists = \{\hat{\mathbf{d}} \in \widehat{D}^k \mid \exists \mathbf{p} \in \mathbf{P}_{RC}\, \exists \mathbf{d} = (d_1, \dots, d_k) \in D^k. \\ \hat{\mathbf{d}} = (\alpha_{\mathbf{p}}(d_1), \dots, \alpha_{\mathbf{p}}(d_k)) \ \wedge \ \mathbf{x}^p = \mathbf{p}, \mathbf{x}^v = \mathbf{d} \models \Phi\} \end{aligned}$$

Hence, the set $||\Phi||_\exists$ contains all vectors of abstract values that correspond to some concrete values satisfying Φ. Parameters do not appear anymore due to existential quantification. A PIA *existential abstraction* of Φ is defined to be a formula $\hat{\Phi}$ over a vector of variables $\hat{\mathbf{x}} = (\hat{x}_1, \dots, \hat{x}_k)$ over $\widehat{D}^k$ such that $\{\hat{\mathbf{d}} \in \widehat{D}^k \mid \hat{\mathbf{x}} = \hat{\mathbf{d}} \models \hat{\Phi}\} \supseteq ||\Phi||_\exists$.

A pictorial representation of PIA data abstraction is given in Figure 4.2. The shaded area approximates the line $x_2 = x_1 + 1$ along the boundaries of our parametric intervals. Each shaded rectangle corresponds to one conjunctive clause in the formula to the right.



$$\begin{aligned}
\hat{\Phi} \;\equiv\; & \hat{x}_1 = I_0 \wedge \hat{x}_2 = I_1 \\
\vee\; & \hat{x}_1 = I_1 \wedge \hat{x}_2 = I_1 \\
\vee\; & \hat{x}_1 = I_1 \wedge \hat{x}_2 = I_2 \\
\vee\; & \hat{x}_1 = I_2 \wedge \hat{x}_2 = I_2 \\
\vee\; & \hat{x}_1 = I_2 \wedge \hat{x}_2 = I_3 \\
\vee\; & \hat{x}_1 = I_3 \wedge \hat{x}_2 = I_3
\end{aligned}$$

Figure 4.2: Existential abstraction of $x_2 = x_1 + 1$

Thus, given $\Phi \equiv x_2 = x_1 + 1$, the shaded rectangles correspond to $||\Phi||_\exists$, from which we immediately construct the existential abstraction $\hat{\Phi}$.

The central property of our abstract domain is that it allows to abstract comparisons against thresholds (i.e., threshold guards) in a precise way. That is, we can abstract formulas of the form $\vartheta_j(\mathbf{p}) \le x_1$ by $I_j \le \hat{x}_1$ and $\vartheta_j(\mathbf{p}) > x_1$ by $I_j > \hat{x}_1$. In fact, this abstraction is precise in the following sense.

Proposition 13. *For all $\mathbf{p} \in \mathbf{P}_{RC}$ and all $a \in D$:*
$\vartheta_j(\mathbf{p}) \le a$ iff $I_j \le \alpha_\mathbf{p}(a)$, and $\vartheta_j(\mathbf{p}) > a$ iff $I_j > \alpha_\mathbf{p}(a)$.

Proof. Fix an arbitrary $\mathbf{p} \in \mathbf{P}_{RC}$.
Case $a \ge \vartheta_j(\mathbf{p})$. ($\Rightarrow$) Fix an arbitrary $a \in D$ satisfying $a \ge \vartheta_j(\mathbf{p})$. Let k be a maximum number such that $a \ge \vartheta_k(\mathbf{p})$. Then $\alpha_\mathbf{p}(a) = I_k$. By Definition of $\alpha_\mathbf{p}$ we have $k \ge j$ and thus, by Definition 12, $I_k \ge I_j$. It immediately gives $\alpha_\mathbf{p}(a) \ge I_j$.

($\Leftarrow$) Let $a \in D$ be a value satisfying $\alpha_\mathbf{p}(a) \ge I_j$. There is k such that $\alpha_\mathbf{p}(a) = I_k$ and $a \ge \vartheta_k(\mathbf{p})$. From $\alpha_\mathbf{p}(a) \ge I_j$ it follows that $I_k \ge I_j$ and, by Definition 12, $k \ge j$. Then by Definition 10 we have $\vartheta_k(\mathbf{p}) \ge \vartheta_j(\mathbf{p})$ and by transitivity $a \ge \vartheta_j(\mathbf{p})$.
Case $a < \vartheta_j(\mathbf{p})$. ($\Rightarrow$) Fix an arbitrary $a \in D$ satisfying $a < \vartheta_j(\mathbf{p})$. Let k be a maximum number such that $a \ge \vartheta_k(\mathbf{p})$. Then $\alpha_\mathbf{p}(a) = I_k$.

Consider the case when $k \ge j$. By definition 12 it implies $I_k \ge I_j$. It immediately gives $\alpha_\mathbf{p}(a) \ge I_j$, which contradicts the assumption $a < \vartheta_j(\mathbf{p})$. Thus, the only case is $k < j$.

By Definition 12, $k < j$ implies $I_k \le I_j$. As we excluded the case $k = j$ we have $I_k \le I_j$, $I_k \ne I_j$ or, equivalently, $\alpha_\mathbf{p}(a) = I_k < I_j$.

($\Leftarrow$) Let $a \in D$ be a value satisfying $\alpha_\mathbf{p}(a) < I_j$ or, equivalently, $\alpha_\mathbf{p}(a) \le I_j$ and $\alpha_\mathbf{p}(a) \ne I_j$. There exists k such that $\alpha_\mathbf{p}(a) = I_k$ and either *(a)* $a < \vartheta_{k+1}(\mathbf{p})$ or *(b)* $k = \mu$. From the assumption we have $I_k \le I_j$ and $I_k \ne I_j$. From this we conclude: *(c)* $k \ne \mu$ excluding *(b)*; *(d)* $I_{k+1} \le I_j$. From *(d)* by Definition 12, $k + 1 \le j$. This implies by Definition 10, $\vartheta_{k+1}(\mathbf{p}) \le \vartheta_j(\mathbf{p})$. From this and *(a)* we conclude that $a < \vartheta_j(\mathbf{p})$. □

For comparison guards we use the general form, well-known from the literature, from the following proposition.

Proposition 14. *If Φ is a formula over variables $x_1, \dots, x_k$ over D, then $\bigvee_{(\hat{d}_1,\dots,\hat{d}_k)\in||\Phi||_\exists} \hat{x}_1 = \hat{d}_1 \wedge \dots \wedge \hat{x}_k = \hat{d}_k$ is a PIA existential abstraction.*

Proof. Consider an arbitrary $\mathbf{d} \in ||\Phi||_\exists$. As $\mathbf{d} \in ||\Phi||_\exists$, it satisfies the conjunct $\hat{x}_1 = \hat{d}_1 \wedge \dots \wedge \hat{x}_k = \hat{d}_k$ and thus satisfies the disjunction $\hat{\Phi}$, i.e. $\mathbf{x} = \mathbf{d} \models \hat{\Phi}_\exists$. As $\mathbf{d}$ is chosen arbitrarily, we conclude that $||\Phi||_\exists \subseteq \{\hat{\mathbf{x}} \in \widehat{D}^k \mid \hat{\mathbf{x}} \models \hat{\Phi}_\exists\}$. □

4.2.1 Computing the existential abstraction

If the domain $\widehat{D}$ is small (as it is in our case), then all vectors of abstract values in $\widehat{D}^k$ can be enumerated and can be checked to find out which belong to our abstraction $||\Phi||_\exists$, using an SMT solver. As example consider the PIA domain $\{I_0, I_1, I_2, I_3\}$ for the CFA from Fig. 2.1. Figure 4.2 illustrates $||\Phi||_\exists$ of $x_2 = x_1 + 1$ and the use of the formula from Proposition 14.

So far, we have seen the abstraction examples and the formal machinery in the form of existential abstraction $||\Phi||_\exists$. Now we show how to compute the abstractions using an SMT solver. The input language we use is Yices [36], but this choice is not essential. Any other solver that supports linear arithmetics over integers, e.g., Z3 [78], should be sufficient for our purpose. We start with declaring the parameters and the resilience condition:

```
(define n :: int)
(define t :: int)
(define f :: int)
(assert (and (> n 3) (>= f 0)
             (>= t 1) (<= f t) (> n (* 3 t))))
```

Listing 7: The parameters and the resilience condition in Yices

Assume that we want to compute the existential abstraction of an expression similar to one found in line 22 in Listing 3, that is,

$$\Phi_1 \equiv a < b + f.$$

According to the definition of $||\Phi_1||_\exists$, we have to enumerate all abstract values of a and b, and check whether there exists a valuation of the parameters n, t, and f and a concretization $\gamma_{n,t,f}$ of the abstract values that satisfies Φ_1. In the case of Φ_1 this boils down to finding all the abstract pairs $(\hat{a}, \hat{b}) \in \widehat{D} \times \widehat{D}$ satisfying the formula:

$$\exists a, b : \alpha_{n,t,f}(a) = \hat{a} \wedge \alpha_{n,t,f}(b) = \hat{b} \wedge a < b + f \tag{4.2}$$

Given $\hat{a}$ and $\hat{b}$, Formula (4.2) can be encoded as a satisfiability problem in linear integer arithmetics. For instance, if $\hat{a} = I_1$ and $\hat{b} = I_0$, then we encode Formula (4.2) as follows:

```
(push)                  ;; store the context for the future use
(define a :: int)
(define b :: int)
(assert (and (>= a 1) (< a (+ t 1)))) ;; α_{n,t,f}(a) = I_1
```

```
10 (assert (and (>= b 0) (< b 1)))        ;; $\alpha_{n,t,f}(b) = I_0$
11 (assert (< a (+ b f)))                 ;; $\Phi_1$
12 (check)                                ;; is satisfiable?
13 (pop)                  ;; restore the previously saved context
```

Listing 8: Are there a and b with $a < b + f$, $\alpha_{n,t,f}(a) = I_1$, and $\alpha_{n,t,f}(b) = I_0$?

When we execute lines (1)–(13) of Listing 8 in Yices, we receive `sat` on the output. That is, Formula 4.2 is valid for the values $\hat{a} = I_1$ and $\hat{b} = I_0$ and $(I_1, I_0) \in ||a < b + f||_\exists$. To see concrete values of a, b, n, t, and f satisfying lines (1)–(13), we issue the following command:

```
14 (set-evidence! true)
15 ;; copy lines (1) - (13) here
```

Yices provides us with the following model:

```
(= n 7)
(= f 2)
(= t 2)
(= a 1)
(= b 0)
```

By enumerating all values from $\widehat{D} \times \widehat{D}$, we obtain the following abstraction of $a < b + f$ (this is an abstraction of line (22) in Listing 3):

```
   a == $I_{n-t}$ && b == $I_{n-t}$ || a == $I_{t+1}$ && b == $I_{n-t}$
|| a == $I_1$ && b == $I_{n-t}$ || a == $I_0$ && b == $I_{n-t}$
|| a == $I_{n-t}$ && b == $I_{t+1}$ || a == $I_{t+1}$ && b == $I_{t+1}$
|| a == $I_1$ && b == $I_{t+1}$ || a == $I_0$ && b == $I_{t+1}$
|| a == $I_{t+1}$ && b == $I_1$ || a == $I_1$ && b == $I_1$
|| a == $I_0$ && b == $I_1$ || a == $I_1$ && b == $I_0$ || a == $I_0$ && b== $I_0$
```

Listing 9: Parametric interval abstraction of $a < b + f$

By applying the same principle to all expressions in Listing 3, we abstract the process code. As the abstract code is too verbose, we do not give it here. It can be obtained by running the tool on our benchmarks [1], as described in Appendix A.

4.2.2 Simulation proof for data abstraction

Transforming CFA: We now describe a general method to abstract `guard` formulas, and thus construct an abstract process skeleton. To this end, by α_E we denote a mapping from a concrete formula Φ to some existential abstraction of Φ (not necessarily constructed as above). By fixing α_E, we can define an abstraction of a `guard` of a CFA:

$$abs(g) = \begin{cases} \alpha_E(g) & \text{if } g \text{ is a threshold guard} \\ \alpha_E(g) & \text{if } g \text{ is a comparison guard} \\ g & \text{if } g \text{ is a status guard} \\ abs(g_1) \wedge abs(g_2) & \text{otherwise, i.e., } g \text{ is } g_1 \wedge g_2 \end{cases}$$

By abusing the notation, for a CFA $\mathcal{A}$ by $abs(\mathcal{A})$ we denote the CFA that is obtained from $\mathcal{A}$ by replacing every guard g with $abs(g)$. Note that $abs(\mathcal{A})$ contains only guards over sv and over abstract variables over $\widehat{D}$.

Definition 15. *We define a mapping $h^{dat}_{\mathbf{p}}$ from valuations v of a CFA $\mathcal{A}$ to valuations $\hat{v}$ of CFA $abst(\mathcal{A})$ as follows: for each variable x over D, $\hat{v}.x = \alpha_{\mathbf{p}}(v.x)$, and for each variable y over SV, $\hat{v}.y = v.y$.*

The following theorem follows immediately from the definition of existential abstraction and $abst(\mathcal{A})$:

Theorem 16. *For all guards g, all $\mathbf{p}$ in $\mathbf{P}_{RC}$, and for all valuations v with $v =_{\Pi} \mathbf{p}$, if $v \models g$, then $h^{dat}_{\mathbf{p}}(v) \models abst(g)$.*

For model checking purposes we have to reason about the Kripke structures that are built using the skeletons obtained from CFAs. By $\mathsf{Sk}_{abs}(\mathcal{A})$ we denote the process skeleton that is induced by CFA $abst(\mathcal{A})$. Analogously to $h^{dat}_{\mathbf{p}}$, we define the parameterized abstraction mapping $\bar{h}^{dat}_{\mathbf{p}}$ that maps global states from $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ to global states from $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$. After that, we obtain Theorem 18 from Theorem 16 and the construction of system instances.

Soundness: It can be shown that for all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all CFA $\mathcal{A}$, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ is simulated by $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$, with respect to AP_{SV}. Moreover, the abstraction of a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ is a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$.

Definition 17. *Let σ be a state of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$, and $\hat{\sigma}$ be a state of the abstract instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$. Then, $\hat{\sigma} = \bar{h}^{dat}_{\mathbf{p}}(\sigma)$ if for each variable $y \in \Lambda \cup \Gamma \cup \Pi$, $\hat{\sigma}.y = \alpha_{\mathbf{p}}(\sigma.y)$, and $\hat{\sigma}.sv = \sigma.sv$.*

Theorem 18. *For all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all CFA $\mathcal{A}$, if system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) = (S_I, S^0_I, R_I, AP, \lambda_I)$ and system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A})) = (S_{\hat{I}}, S^0_{\hat{I}}, R_{\hat{I}}, AP, \lambda_{\hat{I}})$, then: if $(\sigma, \sigma') \in R_I$, then $(\bar{h}^{dat}_{\mathbf{p}}(\sigma), \bar{h}^{dat}_{\mathbf{p}}(\sigma')) \in R_{\hat{I}}$.*

Proof. Let R and R_{abs} be the transition relations of $\mathsf{Sk}(\mathcal{A})$ and $\mathsf{Sk}_{abs}(\mathcal{A})$ respectively. From $(\sigma, \sigma') \in R_I$ and the definition of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ it follows that there is a process index $i : 1 \leq i \leq N(\mathbf{p})$ such that $(\sigma[i], \sigma'[i]) \in R$ and other processes do not change their local states.

Let v be a valuation of $\mathcal{A}$. By the definition of $\mathsf{Sk}(\mathcal{A})$ from $(\sigma[i], \sigma'[i]) \in R$ we have that CFA $\mathcal{A}$ has a path $q_1, g_1, q_2, \dots, q_k$ such that $q_1 = q_I$, $q_k = q_F$ and for every guard g_j it holds that $v \models g_j$. Moreover, for any $x \in \Pi \cup \Lambda \cup \Gamma \cup \{sv\}$ it holds $v(x) = \sigma[i].x$ and $v(x') = \sigma'[i].x$.

We choose the same path in $abst(\mathcal{A})$ and construct the valuation $h^{dat}_{\mathbf{p}}(v)$. From Theorem 16 we have that for every guard g_j it holds that $h^{dat}_{\mathbf{p}}(v) \models abst(g_j)$. Hence, the path $q_1, g_1, q_2, \dots, q_k$ is a path of CFA $abst(\mathcal{A})$ as well.

By the definitions of $h^{dat}_{\mathbf{p}}$ and $\bar{h}^{dat}_{\mathbf{p}}$ we have that for every $x \in \Pi \cup \Lambda \cup \Gamma \cup \{sv\}$ it holds $h^{dat}_{\mathbf{p}}(v)(x) = \bar{h}^{dat}_{\mathbf{p}}(\sigma)[i].x$ and $h^{dat}_{\mathbf{p}}(v)(x') = \bar{h}^{dat}_{\mathbf{p}}(\sigma')[i].x$. By the definition of $\mathsf{Sk}_{abs}(\mathcal{A})$ it immediately follows that $(\bar{h}^{dat}_{\mathbf{p}}(\sigma), \bar{h}^{dat}_{\mathbf{p}}(\sigma')) \in R_{abs}$.

Finally, $(\bar{h}^{dat}_{\mathbf{p}}(\sigma), \bar{h}^{dat}_{\mathbf{p}}(\sigma')) \in R_{abs}$ implies that $(\bar{h}^{dat}_{\mathbf{p}}(\sigma), \bar{h}^{dat}_{\mathbf{p}}(\sigma')) \in R_{\hat{I}}$. □

Theorem 18 is the first step to prove simulation. In order to actually do so, we now define the labeling function $\lambda_{\hat{I}}$. For propositions from AP_{SV}, $\lambda_{\hat{I}}(\hat{\sigma})$ is defined in the same way as λ_I. Similar to [63], for propositions from AP_D, which are used in justice constraints, we define $\lambda_{\hat{I}}(\hat{\sigma})$ as:

$$\exists i.\ x_i + c < y_i \in \lambda_{\hat{I}}(\hat{\sigma}) \text{ iff } \bigvee_{1 \le i \le N(\mathbf{p})} \hat{\sigma}[i] \models \alpha_E(\{x + c(\mathbf{p}) < y\})$$

$$\forall i.\ x_i + c \ge y_i \in \lambda_{\hat{I}}(\hat{\sigma}) \text{ iff } \bigwedge_{1 \le i \le N(\mathbf{p})} \hat{\sigma}[i] \models \alpha_E(\{x + c(\mathbf{p}) \ge y\})$$

From Theorem 18, the definition of $\bar{h}^{dat}_{\mathbf{p}}$ with respect to the variable sv, and the definition of $\lambda_{\hat{I}}$, one immediately obtains the following theorems. Theorem 20 ensures that justice constraints J in the abstract system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$ are a sound abstraction of justice constraints J in $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$.

Theorem 19. *For all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all CFA $\mathcal{A}$, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \preceq \mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$ holds with respect to AP_{SV}.*

Theorem 20. *Let $\pi = \{\sigma_i\}_{i \ge 1}$ be a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$. Then $\hat{\pi} = \{\bar{h}^{dat}_{\mathbf{p}}(\sigma_i)\}_{i \ge 1}$ is a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$.*

Proof. By inductively applying Theorem 18 to π we conclude that $\hat{\pi}$ is indeed a path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$.

Fix an arbitrary justice constraint $q \in J \subseteq \mathrm{AP}_D$; infinitely many states on π are labelled with q. Fix a state σ on π with $q \in \lambda_I$. We show that $q \in \lambda_{\hat{I}}(\bar{h}^{dat}_{\mathbf{p}}(\sigma))$. Consider two cases:

Case 1. Proposition q has a form $[\exists i.\ \Phi(i)]$, where Φ has free variables of two types: a vector of parameters $\mathbf{x}^p = x^p_1, \dots, x^p_{|\Pi|}$ from Π and a vector of variables $\mathbf{x}^v = x^v_1, \dots, x^v_k$. There is a process index $i : 1 \le i \le N(\mathbf{p})$ such that $\sigma[i] \models \Phi(i)$. Hence, $\mathbf{x}^p = \mathbf{p}, x^v_1 = \sigma[i].x_1, \dots, x^v_k = \sigma[i].x_k \models \Phi(i)$. From the definition of the existential abstraction it follows that $(\alpha_{\mathbf{p}}(\sigma[i].x_1), \dots, \alpha_{\mathbf{p}}(\sigma[i].x_k)) \in ||\Phi(i)||_E$. Thus, $\hat{x}^v_1 = \alpha_{\mathbf{p}}(\sigma[i].x_1), \dots, \hat{x}^v_k = \alpha_{\mathbf{p}}(\sigma[i].x_k) \models \alpha_E(\Phi(i))$. As for every $x_j : 1 \le j \le k$ the value $\bar{h}^{dat}_{\mathbf{p}}(\sigma)[i].x_j$ is exactly $\hat{x}^v_j$, we arrive at $\bar{h}^{dat}_{\mathbf{p}}(\sigma)[i] \models \alpha_E(\Phi(i))$. Then by the construction of $\lambda_{\hat{I}}$ it holds that $[\exists i.\ \Phi(i)] \in \lambda_{\hat{I}}(\bar{h}^{dat}_{\mathbf{p}}(\sigma))$.

Case 2. Proposition q has a form $[\forall i.\ \Phi(i)]$, where Φ has free variables of two types: a vector of parameters $\mathbf{x}^p = x^p_1, \dots, x^p_{|\Pi|}$ from Π and a vector of variables $\mathbf{x}^v = x^v_1, \dots, x^v_k$. Then for every process index $i : 1 \le i \le N(\mathbf{p})$ it holds $\sigma[i] \models \Phi(i)$. By fixing an arbitrary $i : 1 \le i \le N(\mathbf{p})$ and repeating exactly the same argument as in the Case 1, we show that

$\bar{h}^{dat}_{\mathbf{p}}(\sigma)[i] \models \alpha_E(\Phi(i))$. We conclude that $\bigwedge_{1 \le i \le N(\mathbf{p})} \bar{h}^{dat}_{\mathbf{p}}(\sigma)[i] \models \alpha_E(\Phi(i))$, as i is chosen arbitrarily. By the construction of $\lambda_{\hat{I}}$ it holds that $[\forall i.\ \Phi(i)] \in \lambda_{\hat{I}}(\bar{h}^{dat}_{\mathbf{p}}(s))$.

From Case 1 and Case 2 we conclude that $q \in \lambda_{\hat{I}}(\bar{h}^{dat}_{\mathbf{p}}(\sigma))$. As we chose σ to be an arbitrary state on π labelled with q and we know that there are infinitely many such states on π, we have shown that there are infinitely many states $\bar{h}^{dat}_{\mathbf{p}}(\sigma)$ on $\hat{\pi}$ labelled with q. Finally, as q was chosen to be an arbitrary justice constraint from J, we conclude that every justice constraint $q \in J$ appears infinitely often on $\hat{\pi}$.

This proves that $\hat{\pi}$ is a fair path. $\square$

Abstraction of specifications: As we have seen in Section 2.4.1, we use only specifications that compare status variable *sv* against a value from SV. For instance, the unforgeability property **U** (U) refers to atomic proposition $[\forall i.\ sv_i \neq \mathrm{V1}]$. Interval data abstraction neither affects the domain of *sv*, nor does it change expressions over *sv*. Thus, we do not have to change the specifications when applying the data abstraction.

However, the specifications are verified under justice constraints, e.g., the reliable communication constraint RelComm (RelComm): $\mathbf{G}\,\mathbf{F} \neg [\exists i.\ \mathit{rcvd}_i < \mathit{nsnt}]$. Our goal is to preserve *fair* or in other words, *just* runs with the abstraction. That is, if each state of a just run is abstracted, then the resulting sequence of abstract states is a just run of the abstract system. Intuitively, when we verify a property that holds on all *abstract* just runs, then we conclude that the property also holds on all *concrete* just runs. In fact, we apply existential abstraction to the formulas that capture just states, e.g., we transform the expression $\neg [\exists i.\ \mathit{rcvd}_i < \mathit{nsnt}]$ using existential abstraction $||\neg [\exists i.\ \mathit{rcvd}_i < \mathit{nsnt}]\,||_\exists$.

Let ψ be a propositional formula that describes just states, and $[\![\psi]\!]_{\mathbf{p}}$ be the set of states that satisfy ψ in the concrete system with the parameter values $\mathbf{p} \in \mathbf{P}_{RC}$. Then, by the definition of existential abstraction, for all $\mathbf{p} \in \mathbf{P}_{RC}$, it holds that $[\![\psi]\!]_{\mathbf{p}}$ is contained in the concretization of $||\psi||_\exists$. This property ensures justice preservation.

Remark on the precision: It can be argued that domain $\widehat{D}$ is too imprecise and it might be helpful to add more elements to $\widehat{D}$. By Proposition 13, however, the domain gives us a *precise* abstraction of the comparisons against the thresholds. Thus, we do not lose precision when abstracting the expressions like $\mathit{next\_nrcvd} < t + 1$ and $\mathit{next\_nrcvd} \geq n - t$, and we cannot benefit from enriching the abstract domain $\widehat{D}$ with expressions different from the thresholds.

4.3 PIA counter abstraction

In the previous section we abstracted a parameterized process into a finite-state process. In this section we turn a system parameterized in the number of finite-state processes into a system which consists of a single process and finitely many states. First, we fix parameters $\mathbf{p}$ and show how to convert a system of $N(\mathbf{p})$ processes into a single process system by using a counting argument.

Counter representation: The structure of the PROMELA program after applying the data abstraction from Section 4.2 looks as follows:

```
int nsnt: 2 = 0;          /* 0↦I_0, 1↦I_1, 2↦I_{t+1}, 3↦I_{n−t} */
active[n - f] proctype Proc() {
  int pc: 2 = 0;          /* 0↦V0, 1↦V1, 2↦SE, 3↦AC */
  int nrcvd: 2 = 0;       /* 0↦I_0, 1↦I_1, 2↦I_{t+1}, 3↦I_{n−t} */
  int next_pc: 2 = 0, next_nrcvd: 2 = 0;
  if  :: pc = 0; /* V0 */
      :: pc = 1; /* V1 */
  fi;
loop: atomic {
    /* receive */
    /* compute */
    /* send */ }
  goto loop;
}
```

Listing 10: Process structure after data abstraction

Note that a system consists of $N(\mathbf{p})$ *identical* processes. We may thus change the representation of a global state. Instead of storing which process is in which local state, we just count the number of processes in each local state. We have seen in the previous section that after the PIA data abstraction, processes have a fixed number of states. Hence, we can use a fixed number of counters. To this end, we introduce a global array of counters k that keeps the number of processes in every potential local state. By L we denote the set of local states and by L_0 the set of initial local states. In order to map the local states to array indices, we define a bijection: $h\colon L \to \{0, \dots, |L| - 1\}$.

In our example, we have 16 potential local states, i.e., $L_{ST} = \{(pc, nrcvd) \mid pc \in \{V0, V1, SE, AC\}, nrcvd \in \widehat{D}\}$. In our PROMELA encoding, the elements of $\widehat{D}$ and SV are represented as integers; we represent this encoding by the function $val\colon \widehat{D} \cup SV \to \{0, 1, 2, 3\}$ so that no two elements of $\widehat{D}$ are mapped to the same number and no two elements of SV are mapped to the same number. We allocate 16 elements for k and define the mapping $h_{ST} : L_{ST} \to \{0, \dots, |L_{ST}| - 1\}$ as $h_{ST}((pc, nrcvd)) = 4 \cdot val(pc) + val(nrcvd)$. Then $k[h_{ST}(\ell)]$ stores how many processes are in local state ℓ. Thus, a global state is given by the array k, and the global variable *nsnt*.

Now we define the transition relation. As we have to capture interleaving semantics, intuitively, if a process is in local state ℓ and goes to a different local state ℓ', then $k[h_{ST}(\ell)]$ must be decreased by 1 and $k[h_{ST}(\ell')]$ must be increased by 1. To do so in our encoding, we first *select* a state ℓ to move away from, perform a step as above, that is, calculate the successor state ℓ', and finally update the counters. Thus, the template of the counter representation looks as follows:

```
int k[16]; /* number of processes in every local state */
int nsnt: 2 = 0;
active[1] proctype CtrAbs() {
  int pc: 2 = 0, nrcvd: 2 = 0;
  int next_pc: 2 = 0, next_nrcvd: 2 = 0;
  /* init */
loop: /* select */
    /* receive */
    /* compute */
    /* send */
    /* update counters */
  goto loop;
}
```
Listing 11: Process structure of counter representation

The blocks *receive*, *compute*, and *send* stay the same, as they were in Section 4.2. The new blocks have the following semantics: In *init*, an initial combination of counters is chosen such that $\sum_{\ell \in L_0} k[\ell] = N(\mathbf{p})$ and $\sum_{\ell \in L \setminus L_0} k[\ell] = 0$. In *select*, a local state ℓ with $k[\ell] \neq 0$ is non-deterministically chosen; In *update counters*, the counters of ℓ and a successor of ℓ are decremented and incremented respectively.

We now consider the blocks in detail and start with *init*. Each of $n - f$ processes starts in one of the two initial states: $(V0, I_0)$ with $h_{ST}(V0, I_0) = 0$ and $(V1, I_0)$ with $h_{ST}(V1, I_0) = 3$. Thus, the initial block non-deterministically chooses the values for the counters $k[0]$ and $k[3]$, so that $k[0] + k[3] = n - f$ and all the other indices are set to zero. The following code fragment encodes this non-deterministic choice. Observe that the number of choices needed is $n - f + 1$, so the length of this code must depend on the choices of these parameters. We will get rid of this requirement in the counter abstraction below.

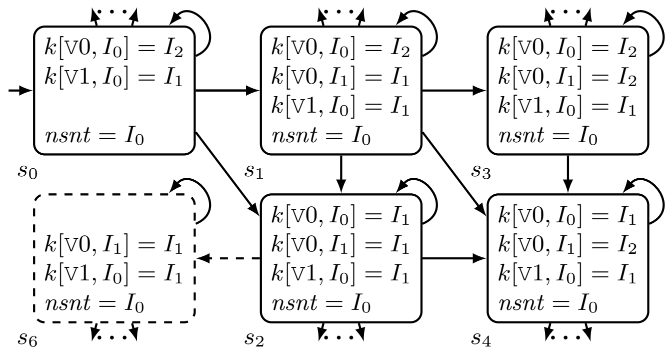
```
if /* 0 ↦ (pc = V0, nrcvd = I_0);  3 ↦ (pc = V1, nrcvd = I_0) */
  :: k[0] = n - f; k[3] = 0;
  :: k[0] = n - f - 1; k[3] = 1;
  ...
  :: k[0] = 0; k[3] = n - f;
fi;
```

In the *select* block we pick non-deterministically a non-zero counter $k[\ell]$ and set *pc* and *nrcvd* so that $h_{ST}(pc, nrcvd) = \ell$. Again, here is a small fragment of the code:

```
if
  :: k[0] != 0  -> pc = 0 /* V0 */; nrcvd = 0 /* I_0 */;
  :: k[1] != 0  -> pc = 0 /* V0 */; nrcvd = 1 /* I_1 */;
  ...
  :: k[15] != 0 -> pc = 3 /* AC */; nrcvd = 3 /* I_{n-t} */;
fi;
```

Finally, as the *compute* block assigns new values to *next_pc* and *next_nrcvd*, which correspond to the successor state of $(pc, nrcvd)$, we update the counters to reflect the fact that one process moved from state $(pc, nrcvd)$ to state $(next\_pc, next\_nrcvd)$:

```
if
  :: pc != next_pc || nrcvd != next_nrcvd ->
     k[4 * pc + nrcvd]--; k[4 * next_pc + next_nrcvd]++;
  :: else; /* do not update the counters */
fi;
```



$k[\text{V0}, I_0] = I_2$
$k[\text{V1}, I_0] = I_1$
$nsnt = I_0$
s_0
$k[\text{V0}, I_0] = I_2$
$k[\text{V0}, I_1] = I_1$
$k[\text{V1}, I_0] = I_1$
$nsnt = I_0$
s_1
$k[\text{V0}, I_0] = I_2$
$k[\text{V0}, I_1] = I_2$
$k[\text{V1}, I_0] = I_1$
$nsnt = I_0$
s_3
$k[\text{V0}, I_1] = I_1$
$k[\text{V1}, I_0] = I_1$
$nsnt = I_0$
s_6
$k[\text{V0}, I_0] = I_1$
$k[\text{V0}, I_1] = I_1$
$k[\text{V1}, I_0] = I_1$
$nsnt = I_0$
s_2
$k[\text{V0}, I_0] = I_1$
$k[\text{V0}, I_1] = I_2$
$k[\text{V1}, I_0] = I_1$
$nsnt = I_0$
s_4

Figure 4.3: A small part of the transition system obtained by counter abstraction.

This representation might look inefficient in comparison to the one with explicit processes; e.g., Spin cannot use partial order reduction on this representation. However, this representation is only an intermediate step.

Specifications: In the original system (before introducing the counter representation) it is obvious how global states are linked with atomic propositions of the form $[\exists i.\ \Phi(i)]$ and $[\forall i.\ \Phi(i)]$; a process i must satisfy $\Phi(i)$ or all processes must do so, respectively. In the counter representation we do not have processes in the system anymore, and we have to understand which states to label with our atomic propositions.

In the counter representation, we exploit the fact that our properties are all quantified, which naturally translates to statements about counters. Let $[\![\Phi]\!]$ be the set of local states that satisfy Φ. In our example we are interested in the local states that satisfy $sv = \text{AC}$, as it appears in our specifications. There are several such states (not all reachable) depending on the different values of *nsnt*. Then, a global state satisfies $[\exists i.\ \Phi(i)]$ if $\bigvee_{\ell \in [\![\Phi]\!]} k[\ell] \neq 0$. Similarly, a global state satisfies $[\forall i.\ \Phi(i)]$ if $\bigwedge_{\ell \notin [\![\Phi]\!]} k[\ell] = 0$.

As we are dealing with counters, instead of using disjunctions and conjunctions, we could also use sums to evaluate quantifiers: the universal quantifier could also be expressed as $\sum_{\ell \in [\![\Phi]\!]} k[\ell] = N(\mathbf{p})$. However, in the following counter abstraction this formalization has drawbacks, due to the non-determinism of the operations on the abstract domain, while the abstraction of 0 is precise.

Counter abstraction: The counter representation encodes a system of $n - f$ processes as a single process system. When, n, t, and f are fixed, the elements of array k are bounded by n. However, in the parameterized case the elements of k are unbounded. To circumvent this problem, we apply the PIA abstraction from Section 4.2 to the elements of k.

In the counter abstraction, the elements of k range over the abstract domain $\widehat{D}$. Similar to Section 4.2, we have to compute the abstract operations over k. These are the operations in the *init* block and in the *update* block.

To transform the *init* block, we first compute the existential abstraction of $\sum_{\ell \in L_0} k[\ell] = N(\mathbf{p})$. In our example, we compute the set $||k[0]+k[3] = n-f||_\exists$ and non-deterministically choose an element from this set. Again, we can do it with YICES. We give the initialization block after the abstraction (note that the number of choices is fixed and determined by the size of the abstract domain):

```
if /* 0 ↦ (pc = V0, nrcvd = I_0);  3 ↦ (pc = V1, nrcvd = I_0)  */
  :: k[0] = 3 /* I_{n-t} */; k[3] = 0 /* I_0 */;
  :: k[0] = 3 /* I_{n-t} */; k[3] = 1 /* I_1 */;
  ...
  :: k[0] = 0 /* I_0 */;     k[3] = 3 /* I_{n-t} */;
fi;
```

Listing 12: Initialization of the counters

In the *update* block we have to compute the abstraction of `k[4 * pc + nrcvd]-` and `k[4 * next_pc + next_nrcvd]++`. We have already seen how to do this with the data abstraction. The update block looks as follows after the abstraction:

```
if
  :: pc != next_pc || nrcvd != next_nrcvd ->
    if /* decrement the counter of the previous state */
      :: (k[((pc * 4) + nrcvd)] == 3) ->
        k[((pc * 4) + nrcvd)] = 3;
      :: (k[((pc * 4) + nrcvd)] == 3) ->
        k[((pc * 4) + nrcvd)] = 2;
      ...
      :: (k[((pc * 4) + nrcvd)] == 1) ->
        k[((pc * 4) + nrcvd)] = 0;
    fi;
    if /* increment the counter of the next state */
      :: (k[((next_pc * 4) + next_nrcvd)] == 3) ->
        k[((next_pc * 4) + next_nrcvd)] = 3;
      :: (k[((next_pc * 4) + next_nrcvd)] == 2) ->
        k[((next_pc * 4) + next_nrcvd)] = 3;
      ...
      :: (k[((next_pc * 4) + next_nrcvd)] == 0) ->
        k[((next_pc * 4) + next_nrcvd)] = 1;
    fi;
  :: else; /* do not update the counters */
fi;
```

Listing 13: Abstract increment and decrement of the counters

In contrast to the counter representation, the increment and decrement of the counters in the array k are now non-deterministic. For instance, the counter `k[((pc * 4) + nrcvd)]` can change its value from I_{n-t} to I_{t+1} or stay unchanged. Similarly, the value of `k[((next_pc * 4) + next_nrcvd)]` can change from I_1 to I_{t+1} or stay unchanged.

This non-determinism adds the following behaviors to the abstract systems:

- both counters could stay unchanged, which leads to stuttering.
- the value of `k[(pc * 4 + nrcvd)]` decreases, while at the same time the value of `k[(next_pc * 4 + next_nrcvd)]` stays unchanged. Thus we lose processes.
- `k[(next_pc * 4 + next_nrcvd)]` increases and `k[(pc * 4 + nrcvd)]` stays unchanged. That is, processes are added.

Some of these behaviors lead to spurious counterexamples we deal with in Section 4.4. Figure 4.3 shows a small part of the transition system obtained from the counter abstraction. We omit local states that have the counter value I_0 to facilitate reading. The state s_0 represents the initial states with $t+1$ to $n-t-1$ processes having $sv = \texttt{V0}$ and 1 to t processes having $sv = \texttt{V1}$. Each transition corresponds to one process taking a step in the concrete system. For instance, in the transition (s_0, s_2) a process with local state $[\texttt{V0}, I_0]$ changes its state to $[\texttt{V0}, I_1]$. Therefore, the counter $\kappa[\texttt{V0}, I_0]$ is decremented and the counter $\kappa[\texttt{V0}, I_1]$ is incremented.

Specifications: Similar to the counter representations, quantifiers can be encoded as expressions on the counters. Instead of comparing to 0, we compare to the abstract zero I_0: A global state satisfies $[\exists i.\ \Phi(i)]$ if $\bigvee_{\ell \in [\![\Phi]\!]} k[\ell] \neq I_0$. Similarly, a global state satisfies $[\forall i.\ \Phi(i)]$ if $\bigwedge_{\ell \notin [\![\Phi]\!]} k[\ell] = I_0$.

Formalization: Our counter abstraction is inspired by [83], which maps a system instance composed of *identical finite state* process skeletons to a single finite state system. We use the PIA domain $\widehat{D}$ along with abstractions $\alpha_E(\{x' = x+1\})$ and $\alpha_E(\{x' = x-1\})$ for the counters.

Stage 1: Vector Addition System with States (VASS) Let $L = \{\ell \in SV \times \bar{D}^{|\Lambda|} \mid \exists s \in S.\ \ell =_{\{sv\}\cup\Lambda} s\}$ be the set of *local states* of a process skeleton. As the domain $\bar{D}$ and the set of local variables Λ are finite, L is finite. We write the elements of L as $\ell_1, \dots, \ell_{|L|}$. We define the counting function $K\colon S_I \times L \to D$ such that $K[\sigma, \ell]$ is the number of processes i whose local state is ℓ in global state $\sigma \in S_I$, i.e., $\sigma[i] =_{\{sv\}\cup\Lambda} \ell$. Thus, we represent the system state σ as a tuple $(g_1, \dots, g_k, K[\sigma, \ell_1], \dots, K[\sigma, \ell_{|L|}])$, i.e., by the shared global state and by the counters for the local states. If a process moves from local state ℓ_i to local state ℓ_j, the counters of ℓ_i and ℓ_j will decrement and increment, respectively.

Stage 2: Abstraction of VASS We abstract the counters K of the VASS representation using the PIA domain to obtain a finite state Kripke structure $\mathsf{C}(\mathsf{Sk})$. To compute $\mathsf{C}(\mathsf{Sk}) = (S_\mathsf{C}, S^0_\mathsf{C}, R_\mathsf{C}, \mathrm{AP}, \lambda_\mathsf{C})$ we proceed as follows:

A state $w \in S_\mathsf{C}$ is given by values of shared variables from the set Γ, ranging over $\bar{D}^{|\Gamma|}$, and by a vector $(\kappa[\ell_1], \dots, \kappa[\ell_{|L|}])$ over the abstract domain $\widehat{D}$ from Section 4.1. More concisely, $S_\mathsf{C} = \widehat{D}^{|L|} \times \bar{D}^{|\Gamma|}$.

Definition 21. *The parameterized abstraction mapping $\bar{h}^{cnt}_{\mathbf{p}}$ maps a global state σ of the system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ to a state w of the abstraction $\mathsf{C}(\mathsf{Sk})$ such that: For all $\ell \in L$ it holds that $w.\kappa[\ell] = \alpha_{\mathbf{p}}(K[\sigma, \ell])$, and $w =_\Gamma \sigma$.*

From the definition, we can see how to construct the initial states. Informally, we require the following to hold true:

1. the initial shared states of $\mathsf{C}(\mathsf{Sk})$ correspond to initial shared states of Sk,
2. there are actually $N(\mathbf{p})$ processes in the system, and
3. initially all processes are in an initial state.

The intuitive construction of the transition relation is as follows: Like in VASS, a step that brings a process from local state ℓ_i to ℓ_j can be modeled by decrementing the (non-zero) counter of ℓ_i and incrementing the counter of ℓ_j using the existential abstraction $\alpha_E(\{\kappa'[\ell_i] = \kappa[\ell_i] - 1\})$ and $\alpha_E(\{\kappa'[\ell_j] = \kappa[\ell_j] + 1\})$.

Like Pnueli, Xu, and Zuck [83] we use the idea of representing counters in an abstract domain, and performing increment and decrement using existential abstraction. They used a three-valued domain representing 0, 1, or more processes. As we are interested, e.g., in the fact whether at least $t + 1$ or $n - t$ processes are in a certain state, the domain from [83] is too coarse for us. Therefore, we use counters from $\widehat{D}$, and we increment and decrement counters using the formulas $\alpha_E(\{x' = x + 1\})$ and $\alpha_E(\{x' = x - 1\})$.

Initial states: Let L_0 be a set $\{\ell \mid \ell \in L \wedge \exists s_0 \in S_0.\ \ell =_{\{sv\}\cup\Lambda} s_0\}$. It captures initial local states. Then $w_0 \in S^0_{\mathsf{C}}$ if and only the following conditions are met:

$$\exists \mathbf{p} \in \mathbf{P}_{RC}\ \exists k_1 \cdots \exists k_{|L|}.\ \sum_{1 \le i \le |L|} k_i = N(\mathbf{p}) \wedge \forall i\colon 1 \le i \le |L|.\ \alpha_{\mathbf{p}}(k_i) = w_0.\kappa[\ell_i]$$

$$\forall i : 1 \le i \le |L|.\ (\ell_i \notin L_0) \to (w_0.\kappa[\ell_i] = I_0)$$

$$\exists s_0 \in S_0.\ w_0 =_\Gamma s_0$$

Less formally:

- Concrete counter values are mapped to $w_0.\kappa[\ell_i]$ using $\alpha_{\mathbf{p}}$.
- We consider only combinations of counters that give a system size $N(\mathbf{p})$.
- Every counter $\kappa[\ell_i]$ is initialized to zero, if the local state ℓ_i is met in no initial state $s_0 \in S_0$.
- A shared variable g of w_0 may be initialized to a value v only if there is some initial state $s_0 \in S_0$ with $s_0.g = v$.

Transition relation: We now formalize the transition relation R_{C} of $\mathsf{C}(\mathsf{Sk})$. The formal definition of when for two states w and w' of the counter abstraction it holds that $(w, w') \in R_{\mathsf{C}}$ is given below in (4.3) to (4.12). We will discuss each of these formulas separately. We start from the transition relation R of the process skeleton Sk from which we abstract. Recall that $(s, s') \in R$ means that a process can go from s to s'. From (4.4) and (4.7) we obtain, $\textsc{from}$ is the local state of s, and $\textsc{to}$ is the local state of s'.

In the abstraction, if $\textsc{from} \neq \textsc{to}$, a step from s to s' is represented by increasing the counter at index $\textsc{to}$ by 1 and decreasing the one at $\textsc{from}$ by 1. Otherwise, that is, if $\textsc{from} = \textsc{to}$, the counter of $\textsc{from}$ should not change. Here "increase" and "decrease" is performed using the corresponding functions over the abstract domain $\widehat{D}$, and the mentioned updates of the counters are enforced in (4.10), (4.11), and (4.9). Further, the counters of all local states different from $\textsc{from}$ and $\textsc{to}$ should not change, which we achieved by (4.12). Performing such a transition should only be possible if there is actually a process in state s, which means in the abstraction that the corresponding counter is greater than I_0. We enforce this restriction by (4.6).

Thus, we abstract the transition with respect to local states. However, s and s' also contain the shared variables. We have to make sure that the shared variables are updated in the abstraction in the same way they are updated in the concrete system, which is achieved in (4.5) and (4.8).

We thus arrive at the formal definition of the abstract transition relation. R_{C} consists of all pairs (w, w') for which there exist s and s' in S, and $\textsc{from}$ and $\textsc{to}$ in L such that equations (4.3)–(4.12) hold:

$$(s, s') \in R \qquad (4.3) \qquad\qquad w.\kappa[\textsc{from}] \neq I_0 \qquad (4.6)$$

$$\textsc{from} =_{\{sv\}\cup\Lambda} s \qquad (4.4) \qquad\qquad \textsc{to} =_{\{sv\}\cup\Lambda} s' \qquad (4.7)$$

$$w =_{\Gamma} s \qquad (4.5) \qquad\qquad w' =_{\Gamma} s' \qquad (4.8)$$

$$(\textsc{to} = \textsc{from}) \;\rightarrow\; w'.\kappa[\textsc{from}] = w.\kappa[\textsc{from}] \qquad (4.9)$$

$$(\textsc{to} \neq \textsc{from}) \;\rightarrow\; (x = w.\kappa[\textsc{to}], x' = w'.\kappa[\textsc{to}] \models \alpha_E(\{x' = x + 1\})) \qquad (4.10)$$

$$(\textsc{to} \neq \textsc{from}) \;\rightarrow\; (x = w.\kappa[\textsc{from}], x' = w'.\kappa[\textsc{from}] \models \alpha_E(\{x' = x - 1\})) \qquad (4.11)$$

$$\forall i : 1 \leq i \leq |L|.(\ell_i \neq \textsc{from} \wedge \ell_i \neq \textsc{to}) \;\rightarrow\; w'.\kappa[\ell_i] = w.\kappa[\ell_i] \qquad (4.12)$$

4.3.1 Simulation proof for counter abstraction

Soundness: The soundness is based on two properties. First, between every concrete system and the abstract system, there is a simulation relation. The central argument to prove this comes from Proposition 13, from which it follows that if a threshold is satisfied in the concrete system, the abstraction of the threshold is satisfied in the abstract systems.

Intuitively, this means that if a transition is enabled in the concrete system, then it is enabled in the abstract system, which is required to prove simulation.

Second, the abstraction of a fair path (with respect to our justice properties) in the concrete system is a fair path in the abstract system. This follows from construction. We label an abstract state with a proposition if the abstract state satisfies the existential abstraction of the proposition, in other words, if there is a concretization of the abstract state that satisfies the proposition. We show that for all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all finite state process skeletons Sk, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is simulated by $\mathsf{C}(\mathsf{Sk})$, w.r.t. AP_{SV}. Further, the abstraction of a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is a J-fair path of $\mathsf{C}(\mathsf{Sk})$.

Theorem 22. *For all $\mathbf{p} \in \mathbf{P}_{RC}$, and all finite state process skeletons Sk, let system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) = (S_I, S_I^0, R_I, AP, \lambda_I)$, and $\mathsf{C}(\mathsf{Sk}) = (S_\mathsf{C}, S_\mathsf{C}^0, R_\mathsf{C}, AP, \lambda_\mathsf{C})$. Then: if $(\sigma, \sigma') \in R_I$, then $\left(\bar{h}_\mathbf{p}^{cnt}(\sigma), \bar{h}_\mathbf{p}^{cnt}(\sigma')\right) \in R_\mathsf{C}$.*

Proof. We have to show that if $(\sigma, \sigma') \in R_I$, then $w = \bar{h}_\mathbf{p}^{cnt}(\sigma)$ and $w' = \bar{h}_\mathbf{p}^{cnt}(\sigma')$ satisfy (4.3) to (4.12). We first note that as $(\sigma, \sigma') \in R_I$, it follows from the (MOVE) property of transition relations that there is a process index i such that $(\sigma[i], \sigma'[i]) \in R_I$; we will use the existence of i in the following:

(4.3). Abbreviating $s = \sigma[i]$ and $s' = \sigma'[i]$, (4.3) follows.

(4.4) and (4.7). Follows immediately from the definition of L.

(4.6). From the definition of $\bar{h}_\mathbf{p}^{cnt}$ it follows that $w.\kappa[\text{ FROM}] = \alpha_\mathbf{p}(K(\sigma, \text{ FROM}))$.

From the existence of the index i it follows that $K(\sigma, \text{ FROM}) \geq 1$. Hence, we have $K(\sigma, \text{ FROM}) \neq 0$ and from the definition of $\alpha_\mathbf{p}$ it follows that $\alpha_\mathbf{p}(K(\sigma, \text{ FROM})) \neq 0$. From $\alpha_\mathbf{p}(1) = I_1$ and Definition 12 of total order we conclude (4.6), i.e. $\alpha_\mathbf{p}(K(\sigma, \text{ FROM})) \neq I_0$.

(4.5) and (4.8). Follows immediately from the definition of $\bar{h}_\mathbf{p}^{cnt}$.

(4.9). Since TO $=$ FROM, it follows from (4.4) and (4.7) that $s =_{\{sv\} \cup \Lambda} s'$. Thus the process with index i does not change its local state. Moreover from the property (FRAME) of transition relations, all processes other than i maintain their local state. It follows that for all ℓ in L, $K(\sigma, \ell) = K(\sigma', \ell)$, and further that $\alpha_\mathbf{p}(K(\sigma, \ell)) = \alpha_\mathbf{p}(K(\sigma', \ell))$, and in particular $\alpha_\mathbf{p}(K(\sigma, \text{ FROM})) = \alpha_\mathbf{p}(K(\sigma', \text{ FROM}))$. Then (4.9) follows from the definition of $\bar{h}_\mathbf{p}^{cnt}$.

(4.10) and (4.11). From the property (FRAME) of transition relations, all processes other than i maintain their local state. Since TO $\neq$ FROM it follows that i changes it local state. It follows that

$$K(\sigma', \text{ TO}) = K(\sigma, \text{ TO}) + 1, \tag{4.13}$$

$$K(\sigma', \text{ FROM}) = K(\sigma, \text{ FROM}) - 1. \tag{4.14}$$

From the definition of $\bar{h}_\mathbf{p}^{cnt}$ we have

$$w'.\kappa[\text{ TO}] = \alpha_\mathbf{p}(K(\sigma', \text{ TO})) \text{ and } w.\kappa[\text{ TO}] = \alpha_\mathbf{p}(K(\sigma, \text{ TO}))$$
$$w'.\kappa[\text{ FROM}] = \alpha_\mathbf{p}(K(\sigma', \text{ FROM})) \text{ and } w.\kappa[\text{ FROM}] = \alpha_\mathbf{p}(K(\sigma, \text{ FROM}))$$

From Proposition 11 follows that

$$K(\sigma', \textsc{to}) \in \gamma_{\mathbf{p}}(w'.\kappa[\textsc{to}]) \text{ and } K(\sigma, \textsc{to}) \in \gamma_{\mathbf{p}}(w.\kappa[\textsc{to}]) \tag{4.15}$$
$$K(\sigma', \textsc{from}) \in \gamma_{\mathbf{p}}(w'.\kappa[\textsc{from}]) \text{ and } K(\sigma, \textsc{from}) \in \gamma_{\mathbf{p}}(w.\kappa[\textsc{from}]). \tag{4.16}$$

(4.10) follows from (4.13), (4.15), and the definition of existential abstraction α_E, while (4.11) follows from (4.14), (4.16), and the definition of existential abstraction α_E.

(4.12). From property (FRAME) processes other than i do not move. The move of process i does not change the number of processes in states other than FROM and TO. Consequently, for all local states ℓ different from FROM and TO it holds that $K(\sigma', \ell) = K(\sigma, \ell)$. It follows that $\alpha_{\mathbf{p}}(K(\sigma', \ell)) = \alpha_{\mathbf{p}}(K(\sigma, \ell))$, and (4.12) follows from the definition of $\bar{h}^{cnt}_{\mathbf{p}}$. □

To prove simulation, we now define the labeling function λ_{C}. Here we consider propositions from $\mathrm{AP}_D \cup \mathrm{AP}_{SV}$ in the form of $[\exists i.\ \Phi(i)]$ and $[\forall i.\ \Phi(i)]$. Formula $\Phi(i)$ is defined over variables from the $|\Pi|$-dimensional vector $\mathbf{x}^p$ of parameters, a k-dimensional vector $\mathbf{x}^\ell$ of local variables and sv, an m-dimensional vector of global variables $\mathbf{x}^g$. Then, the labeling function is defined by

$$[\exists i.\ \Phi(i)] \in \lambda_{\mathsf{C}}(w) \text{ iff } \bigvee_{\ell \in L} \left(\mathbf{x}^\ell =_\Lambda \ell, \mathbf{x}^g =_\Gamma w \models \mathit{abst}(\Phi(i)) \wedge w.k[\ell] \neq I_0\right)$$
$$[\forall i.\ \Phi(i)] \in \lambda_{\mathsf{C}}(w) \text{ iff } \bigwedge_{\ell \in L} \left(\mathbf{x}^\ell =_\Lambda \ell, \mathbf{x}^g =_\Gamma w \models \mathit{abst}(\Phi(i)) \vee w.k[\ell] = I_0\right)$$

Theorem 23. *For all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all finite state process skeletons Sk, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \preceq \mathsf{C}(\mathsf{Sk})$, w.r.t. AP_{SV}.*

Proof. Due to Theorem 22, it is sufficient to show that if a proposition $p \in \mathrm{AP}_{SV}$ holds in state σ of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ then it also holds in state $\bar{h}^{cnt}_{\mathbf{p}}(\sigma)$ and vice versa. We distinguish two types of propositions.

If $p = [\forall i.\ sv_i = Z]$ and $p \in \lambda_I(\sigma)$, then by the definition of λ_I we have $\bigwedge_{1 \le i \le N(\mathbf{p})} (\sigma[i].sv = Z)$. Thus, in global state σ all processes are in a local state with $sv = Z$. In other words, no process is in a local state with $sv \neq Z$. It follows that each local state ℓ satisfies in σ that $\ell.sv = Z$ or $K(\sigma, \ell) = 0$. From the definition of $\bar{h}^{cnt}_{\mathbf{p}}$ and the definition of λ_{C} this case follows. The same argument works in the opposite direction.

If $p = [\exists i.\ sv_i = Z]$ and $p \in \lambda_I(\sigma)$, then by the definition of λ_I we have $\bigvee_{1 \le i \le N(\mathbf{p})} (\sigma[i].sv = Z)$. Thus, in global state σ there is a process in a local state ℓ with $sv = Z$. It follows that $K(\sigma, \ell) > 0$. From the definition of $\bar{h}^{cnt}_{\mathbf{p}}$ and the definition of λ_{C} the case follows. The same argument works in the opposite direction.

These two cases conclude the proof. □

Theorem 24. *If $\pi = \{\sigma_i\}_{i \ge 1}$ is a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$, then the path $\hat{\pi} = \{\bar{h}^{cnt}_{\mathbf{p}}(\sigma_i)\}_{i \ge 1}$ is a J-fair path of C.*

Proof. By inductively applying Theorem 22 to π we conclude that $\hat{\pi}$ is indeed a path of C.

Fix an arbitrary justice constraint $q \in J \subseteq \mathrm{AP}_D$; infinitely many states on π are labelled with q. Fix an arbitrary state σ on π such that $q \in \lambda_I$. We show that $q \in \lambda_{\mathsf{C}}(\bar{h}^{cnt}_{\mathbf{p}}(\sigma))$.

Propositions from AP_D have the form of $[\exists i.\ \Phi(i)]$ and $[\forall i.\ \Phi(i)]$, where each $\Phi(i)$ has free variables of two types: a vector of parameters $\mathbf{x}^p = x^p_1, \ldots, x^p_{|\Pi|}$ from Π, a vector of local variables $x^\ell = x^\ell_1, \ldots, x^\ell_k$ from Λ, and a vector of global variables $x^g = x^g_1, \ldots, x^g_m$ from Γ.

$$[\exists i.\ \Phi(i)] \in \lambda_{\mathsf{C}}(w) \text{ iff } \bigvee_{\ell \in L} \Big(\mathbf{x}^\ell = \ell, \mathbf{x}^g =_\Gamma w \models \alpha_E(\Phi(i)) \wedge w.k[\ell] \neq I_0\Big) \tag{4.17}$$

$$[\forall i.\ \Phi(i)] \in \lambda_{\mathsf{C}}(w) \text{ iff } \bigwedge_{\ell \in L} \Big(\mathbf{x}^\ell = \ell, \mathbf{x}^g =_\Gamma w \models \alpha_E(\Phi(i)) \vee w.k[\ell] = I_0\Big) \tag{4.18}$$

Consider two cases: *Existential case (4.17).* There is a process index $i : 1 \leq i \leq N(\mathbf{p})$ such that $\hat{\sigma}[i] \models \alpha_E(\Phi(i))$.

Consider a local state $\ell \in L$ with $\ell =_L \hat{\sigma}[i]$. As $\hat{\sigma}[i] \models \alpha_E(\Phi(i))$ it follows that $x^\ell_1 = \ell.x^\ell_1, \ldots, x^\ell_k = \ell.x^\ell_k, x^g_1 = w.x^g_1, \ldots, x^g_m = w.x^g_m \models \alpha_E(\Phi(i))$. As i is the index of a process with $\ell =_L \hat{\sigma}[i]$, it immediately follows that $K(w, \ell) \neq 0$. From the definition of α it follows that for every $\mathbf{p} \in \mathbf{P}_{RC}$ it holds $\alpha_{\mathbf{p}}(K(w, \ell)) \neq I_0$. Thus, by the definition of $\bar{h}^{cnt}_{\mathbf{p}}$ we have $w.\kappa[\ell] \neq I_0$.

Hence, both requirements of Equation (4.17) are met for ℓ and from the property of disjunction we have $q \in \lambda_{\mathsf{C}}(w)$.

Universal case (4.18). Then for every process index $i : 1 \leq i \leq N(\mathbf{p})$ it holds $\hat{\sigma}[i] \models \alpha_E(\Phi(i))$.

By fixing an arbitrary $i : 1 \leq i \leq N(\mathbf{p})$, choosing $\ell \in L$ with $\ell =_L w$ and by repeating exactly the same argument as in the existential case, we show that $x^\ell_1 = \ell.x^\ell_1, \ldots, x^\ell_k = \ell.x^\ell_k, x^g_1 = w.x^g_1, \ldots, x^g_m = w.x^g_m \models \alpha_E(\Phi(i))$. Thus, for every $\ell \in L$ such that there exists $i : 1 \leq i \leq N(\mathbf{p})$ with $\ell =_L w$ the disjunct for ℓ in (4.18) holds true.

Consider $\ell' \in L$ such that for every $i : 1 \leq i \leq N(\mathbf{p})$ it holds $\ell' \neq_L w$. It immediately follows that $K(w, \ell') = 0$; from the definition of $\alpha_{\mathbf{p}}$ we have that $\alpha_{\mathbf{p}}(K(w, \ell')) = I_0$ and thus $\kappa[\ell'] = I_0$. Then for ℓ' the disjunct in (4.18) holds true as well.

Thus, we conclude that the conjunction in the right-hand side of the equation (4.18) holds, which immediately results in $q \in \lambda_{\mathsf{C}}(w)$.

From the Universal Case and the Existential Case we conclude that $q \in \lambda_{\mathsf{C}}(w)$. As we chose $\hat{\sigma}$ to be an arbitrary state on π labelled with q and we know that there are infinitely many such states on π, we have shown that there are infinitely many states $\bar{h}^{cnt}_{\mathbf{p}}(\hat{\sigma})$ on $\hat{\pi}$ labelled with q. Finally, as q was chosen to be an arbitrary justice constraint from J, we conclude that every justice constraint $q \in J$ appears infinitely often on $\hat{\pi}$.

This proves that $\hat{\pi}$ is a fair path. □

From Theorems 19, 20, 23, 24, and [38, Thm. 16] we obtain the following central corollary in the form necessary for our parameterized model checking problem.

Corollary 25 (Soundness of data & counter abstraction)**.** *For all CFA $\mathcal{A}$, and for all formulas φ from $\mathsf{LTL}_{\text{-}\mathsf{X}}$ over AP_{SV} and justice constraints $J \subseteq AP_D$: if $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A})) \models_J \varphi$, then for all $\mathbf{p} \in \mathbf{P}_{RC}$ it holds $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \models_J \varphi$.*

Theorem 26 (Soundness of data & counter abstraction)**.** *For all CFA $\mathcal{A}$, and for all formulas φ from $\mathsf{LTL}_{\text{-}\mathsf{X}}$ over AP_{SV} and justice constraints $J \subseteq AP_D$: if $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A})) \models_J \varphi$, then for all $\mathbf{p} \in \mathbf{P}_{RC}$ it holds $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \models_J \varphi$.*

4.4 Abstraction Refinement

In Section 4.2 and Section 4.3, we constructed approximations of the transition systems. First, we transformed parameterized code of a process into finite-state non-parameterized code. Second, we constructed a finite-state process that approximates the behavior of $n - f$ processes.

The states of the abstract system are determined by variables over $\widehat{D}$. Proposition 13 shows that we precisely abstract the relevant properties of our variables, i.e., comparisons to thresholds. Hence, the classic CEGAR approach [25], which consists of refining the state space, does not appear suitable. However, the non-determinism due to our existential abstraction leads to *spurious transitions* that one can eliminate.

We encountered two sources of spurious transitions:

1. Transitions can "lose processes," i.e., any concretization of the abstract number of processes is less than the number of processes we started with. This is not within the assumption of FTDAs, and is thus spurious.

2. In our use case (Figure 2.1) processes increase the global variable *nsnt* by one, when they transfer to a state where the value of the status variable is in $\{\mathrm{SE}, \mathrm{AC}\}$. Hence, in concrete system instances, *nsnt* should always be equal to the number of processes whose status variable value is in $\{\mathrm{SE}, \mathrm{AC}\}$, while due to phenomena similar to those discussed above, we can "lose messages" in the abstract system.

The experiments show that in our case studies neither losing processes nor losing messages has influence on the verification of safety specifications. However, these behaviors pose challenges for liveness as they lead to spurious counterexamples. Message passing FTDAs typically require a process to receive messages from (nearly) all correct processes, which is problematic if processes (i.e., potential senders) or messages are lost.

Besides, in Figure 2.1 we model message receptions by an update of the variable *rcvd*, more precisely, $rcvd \le rcvd' \wedge\ rcvd' \le nsnt + f$. We can observe that this does not require the value of *rcvd* to actually increase. Hence, we add justice requirements, e.g., $J = \{[\forall i.\ rcvd_i \ge nsnt]\}$ in our case study. As observed by [83], counter abstraction may lead to justice suppression. Given a counter-example in the form of a lasso, we detect whether its loop contains only unjust states. If this is the case, similar to an idea from [83], we refine $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ by adding a justice requirement, which is consistent with existing requirements in all concrete instances.

Below, we give a general framework for a sound refinement of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$. We provide a more detailed discussion on the practical refinement techniques that we use in our experiments in Section 4.5. To simplify presentation, we define a *monster system* as a (possibly infinite) Kripke structure $\mathsf{Sys}_\omega = (S_\omega, S_\omega^0, R_\omega, \mathrm{AP}, \lambda_\omega)$, whose state space and transition relation are disjoint unions of state spaces and transition relations of system instances $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) = (S_\mathbf{p}, S_\mathbf{p}^0, R_\mathbf{p}, \mathrm{AP}, \lambda_\mathbf{p})$ over all admissible parameters:

$$S_\omega = \bigcup_{\mathbf{p}\in\mathbf{P}_{RC}} S_\mathbf{p}, \qquad S_\omega^0 = \bigcup_{\mathbf{p}\in\mathbf{P}_{RC}} S_\mathbf{p}^0, \qquad R_\omega = \bigcup_{\mathbf{p}\in\mathbf{P}_{RC}} R_\mathbf{p}$$
$$\lambda_\omega : S_\omega \to 2^{\mathrm{AP}} \text{ and } \forall \mathbf{p} \in \mathbf{P}_{RC}, \forall s \in S_\mathbf{p}.\ \lambda_\omega(s) = \lambda_\mathbf{p}(s)$$

Let $h\colon S_\omega \to S_\mathsf{C}$ be an abstraction mapping, e.g., a combination of the abstraction mappings from Section 4.2 and Section 4.3.

Definition 27. *A sequence $T = \{\sigma_i\}_{i\geq 1}$ is a* concretization *of path $\hat{T} = \{w_i\}_{i\geq 1}$ from $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ if and only if $\sigma_1 \in S_\omega^0$ and for all $i \geq 1$ it holds $h(\sigma_i) = w_i$.*

Definition 28. *A path $\hat{T}$ of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ is a* spurious path *iff every concretization T of $\hat{T}$ is not a path in Sys_ω.*

A prerequisite to abstraction refinement is to check whether a counter-example provided by the model checker is spurious. While for finite state systems there are methods to detect whether a path is spurious [25], we are not aware of a method to detect whether a path $\hat{T}$ in $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ corresponds to a path in the (concrete) infinite monster system Sys_ω. Therefore, we limit ourselves to detecting and refining uniformly spurious transitions and unjust states. We first consider spurious transitions.

Definition 29. *An abstract transition $(w, w') \in R_\mathsf{C}$ is* uniformly spurious *iff there is no transition $(\sigma, \sigma') \in R_\omega$ with $w = h(\sigma)$ and $w' = h(\sigma')$.*

The following theorem provides us with a general criterion that ensures that removing uniformly spurious transitions does not affect the property of transition preservation.

Theorem 30. *Let $T \subseteq R_\mathsf{C}$ be a set of spurious transitions. Then for every transition $(\sigma, \sigma') \in R_\omega$ there is a transition $(h(\sigma), h(\sigma'))$ in $R_\mathsf{C} \setminus T$.*

Proof. Assume that there is transition $(\sigma, \sigma') \in R_\omega$ with $w = h(\sigma)$, $w' = h(\sigma')$, and $(w, w') \in R_\mathsf{C} \cap T$. As T is a set of uniformly spurious transitions, we have that the transition (w, w') is uniformly spurious. Consider a pair of states $\rho, \rho' \in S_\omega$ with the property $h(\rho) = w$ and $h(\rho') = w'$. From Definition 29 it follows that $(\rho, \rho') \notin R_\omega$. This contradicts the assumption $(\sigma, \sigma') \in R_\omega$ as we can take $\rho = \sigma$ and $\rho' = \sigma'$. □

It follows that the system $(S_\mathsf{C}, S_\mathsf{C}^0, R_\mathsf{C} \setminus T, \mathrm{AP}, \lambda_\mathsf{C})$ still simulates Sys_ω. After considering spurious transitions, we now consider justice suppression.

Definition 31. *An abstract state $w \in S_{\mathsf{C}}$ is* unjust under *$q \in AP_D$ iff there is no concrete state $\sigma \in S_\omega$ with $w = h(\sigma)$ and $q \in \lambda_\omega(\sigma)$.*

Consider infinite counterexamples of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$, which have a form of lassos $w_1 \dots w_k(w_{k+1} \dots w_m)^\omega$. For such a counterexample $\hat{T}$ we denote the set of states in the lasso's loop by U. We then check, whether all states of U are unjust under some justice constraint $q \in J$. If this is the case, then $\hat{T}$ is a spurious counterexample, because the justice constraint q is violated. Note that it is sound to only consider infinite paths, where states outside of U appear infinitely often; in fact, this is a justice requirement. To refine C's unjust behavior we add a corresponding justice requirement. Formally, we augment J (and AP_D) with a propositional symbol $[\textit{off } U]$. Further, we augment the labelling function λ_{C} such that every $w \in S_{\mathsf{C}}$ is labelled with $[\textit{off } U]$ if and only if $w \notin U$.

Theorem 32. *Let $J \subseteq AP_D$ be a set of justice requirements, $q \in J$, and $U \subseteq S_{\mathsf{C}}$ be a set of unjust states under q. Let $\pi = \{\sigma_i\}_{i \geq 1}$ be an arbitrary fair path of Sys_ω under J. The path $\hat{\pi} = \{h(\sigma_i)\}_{i \geq 1}$ is fair in $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ under $J \cup \{[\textit{off } U]\}$.*

Proof. Consider an arbitrary fair path $\pi = \{\sigma_i\}_{i \geq 1}$ of Sys_ω under J. Assume that $\hat{\pi} = \{h(\sigma_i)\}_{i \geq 1}$ is fair under J, but it becomes unfair under $J \cup \{[\textit{off } U]\}$.

If $\hat{\pi}$ is unfair under $\{[\textit{off } U]\}$, then $\hat{\pi}$ does not have infinitely many states labelled with $[\textit{off } U]$. Thus, $\hat{\pi}$ must have an infinite suffix $\textit{suf}(\hat{\pi})$, where each $w \in \textit{suf}(\hat{\pi})$ has the property $[\textit{off } U] \notin \lambda_{\mathsf{C}}$. From the definition of $[\textit{off } U]$ we immediately conclude that every state $w \in \textit{suf}(\hat{\pi})$ belongs to U, i.e., w is unjust under $q \in J$.

Using the suffix $\textit{suf}(\hat{\pi})$ we reconstruct a corresponding suffix $\textit{suf}(\pi)$ of π (by skipping the prefix of the same length as in $\hat{\pi}$). From the fact that every state of $\textit{suf}(\hat{\pi})$ is unjust under q we know that every state $\sigma \in \textit{suf}(\pi)$ violates the constraint q as well, namely, $q \notin \lambda_\omega(\sigma)$. Thus, π has at most finitely many states labelled with $q \in J$. It immediately follows from the definition of fairness that π is not fair under J. This contradicts the assumption of the theorem. □

From this it follows that loops containing only unjust states can be eliminated, and thus $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ be refined.

We encountered cases where several non-uniform spurious transitions resulted in a uniformly spurious path (i.e., a counterexample). We refine such spurious behavior by invariants. These invariants are provided by the user as invariant candidates, and are then automatically checked to actually be invariants using an SMT solver. In our example the invariant is simply "the number of processes that sent a message equals the number of sent messages."

4.5 Practical Refinement Techniques

Given a run of the counter abstraction, we have to check that the run is spurious for all combinations of parameters from $\mathbf{P}_{RC}$. This problem is again parameterized, and we are not aware of techniques to deal with it in the general case. Thus, we limit ourselves to detecting the runs that have a *uniformly spurious transition*, that is, a transition that does not have a concretization for all the parameters from $\mathbf{P}_{RC}$.

We check for spurious transitions using SMT solvers. To do so, we have to encode the transition relation of all concrete systems (which are defined by different parameter values) in SMT. We explain our approach in three steps: first we encode a single PROMELA statement. Based on this we encode a process step that consists of several statements. Finally, we use the encoding of a step to define the transition relation of the system.

4.5.1 Encoding the transition relation

Encoding a single statement: As we want to detect spurious behavior, the SMT encoding must capture a system on a less abstract level than the counter abstraction. One first idea would be to encode the transition relation of the concrete systems. However, as we do parameterized model checking, we actually have infinitely many concrete systems, and the state space and the number of processes in these systems is not bounded. Hence, we require a representation whose "degree of abstraction" lies between the concrete systems and the counter abstraction. In principle, the counter representation from Section 4.3 seems to be a good candidate. Its state is given by finitely many integer counters, and finitely many shared variables that range over the abstract domain. Although there are infinitely many states (the counters are not bounded), the state space and transition relation can be encoded as an SMT problem. Moreover, threshold guards and the operations on the process counters can be expressed in linear integer arithmetic, which is supported by many SMT solvers.

However, experiments showed that we need a representation closer to the concrete systems. Hence, we use a system whose only difference to the counter representation from Section 4.3 is that the shared variables are not abstracted. The main difficulty in this is to encode transitions that involve abstract local as well as concrete global variables. For that, we represent the parameters in SMT. Then, instead of comparing global variables against abstract values, we check whether the global variables are within parametric intervals. Here we do not go into the formal details of this abstraction. Rather, we explain it by example. The most complicated case is the one where an expression involves the parameters, local variables, and shared variables. For instance, consider the code in Listing 8, where a is a local variable and b is a shared one. In this new abstraction a is abstract and b is concrete. Thus, we have to encode constraints on b as inequalities expressing which interval b belongs to. Specifically, we replace $b = I_k$ with either $\vartheta_k \leq b < \vartheta_{k+1}$ (when k is not the largest threshold ϑ_μ), or $\vartheta_\mu \leq b$ (otherwise):

```
   $a == I_{n-t}$ && $n-t \leq b$ || $a == I_{t+1}$ && $n-t \leq b$
|| $a == I_1$ && $n-t \leq b$ || $a == I_0$ && $n-t \leq b$
|| $a == I_{n-t}$ && $t+1 \leq b < n-t$ || $a == I_{t+1}$ && $t+1 \leq b < n-t$
|| $a == I_1$ && $t+1 \leq b < n-t$ || $a == I_0$ && $t+1 \leq b < n-t$
|| $a == I_{t+1}$ && $1 \leq b < t+1$ || $a == I_1$ && $1 \leq b < t+1$
|| $a == I_0$ && $1 \leq b < t+1$ || $a == I_1$ && $0 \leq b < 1$
|| $a == I_0$ && $0 \leq b < 1$
```

Apart from this, statements that depend solely on shared variables are not changed. Finally, statements that consist of local variables and parameters are abstracted as in Section 4.2. This level of abstraction allows us to detect spurious transitions of both types (a) and (b).

Encoding a single process step: Let us recall that our Promela code is an implementation of the CFA representation of the pseudo-code. That is, the Promela code defines a transition system. A single iteration of the loop expresses one step (or transition) which consists of several expressions executed indivisibly. The code before the loop defines the constraints on the initial states of the transition system.

As introduced in Section 2.3, formally, a *guarded control flow automaton* (CFA) is an edge-labeled directed acyclic graph $\mathcal{A} = (Q, q_I, q_F, E)$ with a finite set Q of nodes called locations, an initial location $q_I \in Q$, and a final location $q_F \in Q$. Edges are labeled with simple Promela statements (assignments and comparisons). Each transition is defined by a path from q_I to q_F in a CFA. Our goal is to construct a formula that encodes the transition relation. We do this by translating a statement on every edge from E into an SMT formula in a way similar to [13, Chapter 16]. Let us recall that the CFA for the algorithm is given in the SSA form. That is we take care of multiple assignments to the same variable, such that they do not overwrite previously assigned values. We assume the following notation for the multiple copies of a variable x in SSA: x denotes the *input* variable, that is, the copy of x at location q_I; x' denotes the *output* variable, that is, the copy of x at location q_F; x^i denotes a *temporary* variable, that is, a copy of x that is overwritten by another copy before reaching q_F. This requires us to capture all paths of the CFA. Our goal is to construct a single formula T which represents the process transition relation over the following vectors of free variables:

- $\mathbf{p}$ is the vector of integer parameters from Π, which is not changed by a transition;
- $\mathbf{x}$ is the vector of integer input variables from $\Lambda \cup \{sv\}$;
- $\mathbf{x}'$ is the vector of integer output variables of $\mathbf{x}$;
- $\mathbf{g}$ is the vector of integer input variables from Γ;
- $\mathbf{g}'$ is the vector of integer output variables of $\mathbf{g}$;
- $\mathbf{t}$ is the vector of integer temporary variables of $\mathbf{x}$ and $\mathbf{g}$;
- $\mathbf{en}$ is the vector of boolean variables, one variable en_e per an edge $e \in E$, which means that edge e lies on the path from q_I to q_F.

Let *form*(s) be a straightforward translation of a Promela statement s into a formula as discussed above. Assignments are replaced with equalities and relations (e.g., $\leq$, $>$) are kept as they are. Then, for an edge $e \in E$ labeled with a statement s, we construct a formula $T_e(\mathbf{p}, \mathbf{x}, \mathbf{x}', \mathbf{g}, \mathbf{g}', \mathbf{t}, \mathbf{en})$ as follows:

$$T_e \equiv en_e \rightarrow \mathit{form}(s),$$

Now, formula T is constructed as the following conjunction whose subformulas are discussed in detail below:

$$T \equiv \quad \mathit{start} \wedge \mathit{follow} \wedge \mathit{mux} \wedge \bigwedge_{e \in E} T_e$$

Intuitively, *start* says that at least one edge outgoing from q_I is activated, *follow* says that whenever a location has an incoming activated edge, it also has at least one outgoing activated edge and *mux* expresses the fact that at most one outgoing edge can be picked. These formulas are defined formally as follows:

$$\begin{aligned}
\mathit{start} \equiv &\quad \bigvee_{(q,q') \in E:\ q = q_I} en_{(q,q')} \\
\mathit{follow} \equiv &\quad \bigwedge_{(q,q') \in E} \left(en_{(q,q')} \rightarrow \bigvee_{(q',q'') \in E} en_{(q',q'')} \right) \\
\mathit{mux} \equiv &\quad \bigvee_{(q,q'),(q,q'') \in E} \neg en_{(q,q')} \vee \neg en_{(q,q'')}
\end{aligned}$$

We have to introduce formula *mux*, because the branching operators in PROMELA allow one to pick a branch non-deterministically whenever the guard of the branch evaluates to true. To pick exactly one branch, we have to introduce the mutual exclusion constraints in the form of *mux*. In contrast, programming languages like `C` do not need this constraint, as the conditions of the if-branch and the else-branch cannot both evaluate to true simultaneously.

Having constructed formula T, we say that a process can take a transition from state $\mathbf{x}$ to state $\mathbf{x}'$ under some combination of parameters if and only if the formula $T(\mathbf{p}, \mathbf{x}, \mathbf{x}', \mathbf{g}, \mathbf{g}', \mathbf{t}, \mathbf{en}) \wedge RC(\mathbf{p})$ is satisfiable.

Transition relation of the counter representation: Now we show how to encode the transition relation R of the counter representation using the process transition relation T. The transition relation connects counters $\mathbf{k}$ and global variables $\mathbf{g}$ before a step with their primed versions $\mathbf{k}'$ and $\mathbf{g}'$ after the step. Recall that in Section 4.3, we introduced the bijection h that maps states to numbers. In the following, by abusing the notation, we denote an SMT expression that encodes the bijection h, by $h(\mathbf{x})$. We use the formulas *dec* and *inc*. Informally, *dec* ensures that the counter that corresponds to $h(\mathbf{x})$ is not equal to zero and decrements the counter, while *inc* increments the counter $\mathbf{k}[h(\mathbf{x}')]$. Let R be the following conjunction:

$$R \equiv \mathit{dec} \wedge T \wedge \mathit{inc} \wedge \mathit{keep},$$

We define *dec*, *inc*, and *keep* as follows:

$$\begin{aligned}
\mathit{dec} \equiv &\quad \bigwedge_{0 \le \ell < |L|} (h(\mathbf{x}) = \ell) \rightarrow \mathbf{k}[\ell] > 0 \wedge \mathbf{k}'[\ell] = \mathbf{k}[\ell] - 1 \\
\mathit{inc} \equiv &\quad \bigwedge_{0 \le \ell < |L|} (h(\mathbf{x}') = \ell) \rightarrow \mathbf{k}'[\ell] = \mathbf{k}[\ell] + 1 \\
\mathit{keep} \equiv &\quad \bigwedge_{0 \le \ell < |L|} (h(\mathbf{x}) \neq \ell \wedge h(\mathbf{x}') \neq \ell) \rightarrow \mathbf{k}'[\ell] = \mathbf{k}[\ell]
\end{aligned}$$

Now we can say that a counter representation of a system takes a step from $(\mathbf{k}, \mathbf{g})$ to $(\mathbf{k}', \mathbf{g}')$ if and only if $R(\mathbf{p}, \mathbf{x}, \mathbf{x}', \mathbf{k}, \mathbf{k}', \mathbf{g}, \mathbf{g}', \mathbf{t}, \mathbf{en}) \wedge RC(\mathbf{p})$ is satisfiable. In what follows, we denote the latter formula by *Step*.

In order to encode operations on $\mathbf{k}$, we use arrays. In our case, however, each array may be replaced with $|L|$ integer variables. Thus, we do not actually use important properties of array theory.

4.5.2 Spurious Behavior

Losing and introducing processes: We start with the first type of spurious behavior, where a transition "loses" or "introduces" processes. Consider the following sequence of abstract states, which introduces new processes due to non-determinism of the counter updates:

```
1 k = {0, 0, 0, 0,  3 , 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =0
2 k = {0, 0, 0, 0,  3 , 0, 0, 0,  1 , 0, 0, 0, 0, 0, 0, 0}, nsnt =1
3 k = {0, 0, 0, 0,  3 , 0, 0, 0,  2 , 0, 0, 0, 0, 0, 0, 0}, nsnt =2
```

Here we represent the abstract states in the format similar to the one used in our tool. The assignment "$k = \{\dots\}$" shows the contents of the array k in C format, that is, the position $i = h(\ell)$ gives the abstract number of processes in local state ℓ. The assignment "$nsnt = \dots$" shows the value of $nsnt$.

As one can see, counter $\mathtt{k}[8]$ changes its value from I_0 to I_1 and then to I_{t+1}. The combination of $\mathtt{k}[8] = I_{t+1}$ and $\mathtt{k}[4] = I_{n-t}$ indicates that the transition from state 2 to state 3 is spurious. In fact, we can detect this kind of spurious behavior with YICES:

```
1  (set-evidence! true)
2  (set-verbosity! 3)
3  (define n::int)
4  (define t::int)
5  (define f::int)
6  (assert (and (> n (* 3 t)) (> t f) (>= f 0)))
7  (define k :: (-> (subrange 0 15) nat))
8  (assert+ (and (<= (- n t) (k 4))))
9  (assert+ (and (<= (+ t 1) (k 8)) (< (k 8) (- n t))))
10 ;; -> copy the assertion below for the indices 1-3, 5-7, 9-15
11 (assert+ (and (<= 0 (k 0)) (< (k 0) 1)))
12 (assert (= (- n f) (+
13   (k 0) (k 1) (k 2) (k 3) (k 4) (k 5) (k 6) (k 7)
14   (k 8) (k 9) (k 10) (k 11) (k 12) (k 13) (k 14) (k 15))))
15 (check)
```

Listing 14: Constraints on state 3 encoded in YICES

In lines (8) – (11), we constrain the values of process counters to reside within the parametric intervals as defined by the abstract values of state 3. In lines (12) – (14), we assert that the total number of processes equals $n - f$. YICES reports that the constraints are unsatisfiable, which means that state 3 cannot be an abstraction of a system state with $n - f$ processes. We conclude that the transition from state 2 to state 3 is uniformly spurious, and we eliminate it.

In fact, Yices also reports that it did not use all the assertions to come up with unsatisfiability. An unsatisfiable core — a minimal set of assertions that leads to unsatisfiability — consists of the assertions in lines (8) – (9). Thus, we can remove every transition leading to a state with $k[4] = I_{n-t}$ and $k[8] = I_{t+1}$.

Now consider a sequence of abstract states, which is losing processes:

```
1 k = {0, 0, 0, 0,  3 , 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =0
2 k = {0, 0, 0, 0,  2 , 0, 0, 0,  1 , 0, 0, 0, 0, 0, 0, 0}, nsnt =1
3 k = {0, 0, 0, 0,  1 , 0, 0, 0,  1 , 0, 0, 0, 0, 0, 0, 0}, nsnt =1
```

As with the case of introducing processes, we can detect with Yices that the transition from state 2 to state 3 is uniformly spurious, and eliminate all the transitions captured with an unsatisfiable core.

Losing messages: In our case study (Figure 2.1) processes increase the global variable *nsnt* by one, when they transfer to a state where the value of the status variable is in $\{\text{SE}, \text{AC}\}$. Hence, in concrete system instances, *nsnt* should always be equal to the number of processes whose status is in $\{\text{SE}, \text{AC}\}$, while due to phenomena similar to those discussed above, we can "lose messages" in the abstract system. When checking safety properties, this kind of spurious behavior does not produce counterexamples. However, it generates spurious counterexamples for liveness. Consider the following example:

```
1 k = {0, 0, 0, 0,  3 , 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =0
2 k = {0, 0, 0, 0,  3 , 0, 0, 0,  1 , 0, 0, 0, 0, 0, 0, 0}, nsnt = 1
3 k = {0, 0, 0, 0,  2 , 0, 0, 0,  2 , 0, 0, 0, 0, 0, 0, 0}, nsnt = 1
```

Consider state 3. Here, the number of processes with $sv = \text{SE}$ is at least $t + 1$ (as $k[8] = 2$ corresponding to I_{t+1}), while the number of messages is always strictly less than $t + 1$ (as $nsnt = 1$ corresponding to I_1). We can try to check, whether the transition from state 2 to state 3 is spurious. Again, we add the constraints by *Step*:

```
(set-evidence! true)
(set-verbosity! 3)
(define n::int)
(define t::int)
(define f::int)
(assert (and (> n (* 3 t)) (> t f) (>= f 0)))
(define k :: (-> (subrange 0 15) nat))
(define k' :: (-> (subrange 0 15) nat))
(assert+ (and (<= (+ t 1) (k 4)) (<= (k 4) (- n t) )))
(assert+ (and (<= 1 (k 8)) (< (k 8) (+ t 1))))
(assert+ (and (<= 1 (k' 4)) (<= (k' 4) (+ t 1) )))
(assert+ (and (<= 1 (k' 8)) (< (k' 8) (+ t 1))))
;; copy the assertions below for the indices 1-3, 5-7, 9-15
(assert+ (and (<= 0 (k 0)) (< (k 0) 1)))
(assert+ (and (<= 0 (k' 0)) (< (k' 0) 1)))
;; -> copy Step here <-
(check)
```

Listing 15: Concretization of transition from state 2 to state 3 in Yices

However, this time YICES reports that the constraints are satisfiable. Indeed, it is possible to pick the number of processes that satisfy the constraints in lines (9) – (12) in Listing 15 and still do not increase `nsnt` so that it reaches $t + 1$. As we know that this example represents spurious behavior, the user can introduce an invariant candidate in PROMELA:

```
atomic tx_inv =
    ((card(Proc:pc == SE) + card(Proc:pc == AC)) == nsnt);
```

Then we can automatically test, whether the invariant candidate *tx_inv* is an invariant by checking that the following formula is unsatisfiable (*tx_inv′* is a copy of *tx_inv* with $\mathbf{x}$ replaced by $\mathbf{x}'$, and *Init* is a formula encoding the initial states):

$$\neg((\mathit{Init} \rightarrow \mathit{tx\_inv}) \land ((\mathit{tx\_inv} \land \mathit{Step}) \rightarrow \mathit{tx\_inv}'))$$

As soon as we know that `tx_inv` is an invariant, we can add the following assertion to the previous query in YICES:

```
(assert (= nsnt (+
  (k 8) (k 9) (k 10) (k 11) (k 12) (k 13) (k 14) (k 15))))
```

Listing 16: Constraint expressed by the invariant *tx_inv*

With this assertion in place, we discover that the transition from state 2 to state 3 is uniformly spurious.

4.5.3 Removing transitions in Promela

So far, we were concerned with detecting uniformly spurious transitions. Now we discuss how can we remove spurious transitions from the counter abstraction that was introduced in Section 4.3 (Listing 11).

Whenever we detect a uniformly spurious transition, we extract two sets of constraints from the SMT solver:

1. The constraints on the abstract state before the transition (precondition) and

2. The constraints on the abstract state after the transition (postcondition).

Consider the following uniformly spurious transition:

```
k = {1,  0 , 0, 0,  3 , 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =0
k = {1,  1 , 0, 0,  3 , 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt = 1
```

Here we extract the following constraints from an unsatisfiability core given to us by the SMT solver (written in PROMELA notation):

```
pre = (nsnt == 0);
post = (k[0] == 1) && (k[1] == 2)
  && (k[4] == 3) && (k[15] == 0) && (nsnt == 0);
```

In order to remove the spurious transition, we have to enforce Spin to prune the executions that include the transition. To this end, we introduce a boolean variable `is_spur` that turns true, whenever the current execution has at least one spurious transition. Then for each refinement iteration $K \geq 1$ we introduce a boolean variable `pK_pre` that turns true, whenever the current state satisfies the precondition of the spurious transition detected in iteration K. We modify Promela code as follows:

```
bool is_spur = 0; /* is the current execution spurious */
bool p1_pre = 0;  /* detected at refinement iteration 1 */
...
bool pK_pre = 0;  /* detected at refinement iteration K */
...
active[1] proctype CtrAbs() {
  ...
  /* init */
loop:
  ...
  pK_pre = (nsnt == 0);
  /* select */
  /* receive */
  /* compute */
  /* send */
  /* update counters */
  ...
  /* is the current transition spurious? */
  spur = spur || pK_pre && k[0] == 1 && k[1] == 2
       && k[4] == 3 && k[15] == 0 && nsnt == 0;
  goto loop;
}
```

Listing 17: Counter abstraction with detection of spurious transitions

Finally, we prune the spurious executions by modifying each $\mathsf{LTL}_{\text{-X}}$ formula φ in Promela specifications as follows:

```
  []!is_spur -> φ
```

4.5.4 Detecting unfair loops

There is a third kind of spurious behavior that is not present in our case study, but it occurs in the experiments with omission faults (Section 2.4.4). Modeling omission faults introduces 12 local states instead of 16. Here is a counterexample showing the violation of the liveness property R (Section 2.4.1):

```
k = { 2 , 0, 0,  2 , 0, 0, 0, 0, 0, 0, 0, 0}, nsnt = 0
k = { 2 , 0, 0,  2 , 0, 0,  1 , 0, 0, 0, 0, 0}, nsnt =  1
k = { 2 , 0, 0,  2 , 0, 0,  1 ,  1 , 0, 0, 0, 0}, nsnt =  2
k = { 2 , 0, 0,  2 , 0, 0,  1 ,  2 , 0, 0, 0, 0}, nsnt =  2
k = { 1 , 0, 0,  2 , 0, 0,  1 ,  2 , 0, 0, 0, 0}, nsnt =  2
k = {0, 0, 0,  2 , 0, 0,  1 ,  2 , 0, 0, 0, 0}, nsnt =  2
```

```
 9 k = {0, 0, 0,  1 , 0, 0,  1 ,  2 , 0, 0, 0, 0}, nsnt =  2
10 k = {0, 0, 0, 0, 0, 0,  1 ,  2 , 0, 0, 0, 0}, nsnt =  2
11 k = {0, 0, 0, 0, 0, 0, 0,  2 , 0, 0, 0, 0}, nsnt =  2
12 k = {0, 0, 0, 0, 0, 0, 0,  2 , 0, 0, 0,  1 }, nsnt =  2
13 k = {0, 0, 0, 0, 0, 0, 0,  2 , 0, 0, 0,  2 }, nsnt =  2
14 k = {0, 0, 0, 0, 0, 0, 0,  1 , 0, 0, 0,  2 }, nsnt =  2
15  «««START OF CYCLE»»>
16 k = {0, 0, 0, 0, 0, 0, 0,  1 , 0, 0, 0,  2 }, nsnt =  2
```

Listing 18: A counterexample with a spurious (unfair) loop

Here state 16 is repeated in a loop, but it violates the following fairness constraint which says that up to $nsnt - F$ messages must be eventually delivered:

```
  atomic in_transit = some(Proc:nrcvd < nsnt - F);
  ltl fairness { []<>(!in_transit) && (...) }
```

Again, using the SMT solver we can check whether the loop is unfair, that is, no state within the loop satisfies the fairness constraint, e.g., `!in_transit`.

```
(set-evidence! true)
(set-verbosity! 3)
(define n::int)
(define t::int)
(define f::int)
(assert (and (> n (* 2 t)) (> t f) (>= f 0)))
(define k :: (-> (subrange 0 11) nat))
;; the constraints by the state 14:
(assert+ (and (<= 1 (k 7)) (< (k 4) (+ t 1) )))
(assert+ (and (<= (+ t 1) (k 11))))
;; -> repeat the assertion below for the indices 0-6, 7-10
(assert+ (and (<= 0 (k 0)) (< (k 0) 1)))
(assert+ (>= nsnt (+ t 1)))
;; constraints by !in_transit
(assert+ (not (or
 (and (>= (- nsnt f) (+ t 1))
   (or (/= (k 1) 0) (/= (k 4) 0) (/= (k 7) 0) (/= (k 10) 0)))
 (and (>= (- nsnt f) (+ t 1))
   (or (/= (k 0) 0) (/= (k 3) 0) (/= (k 6) 0) (/= (k 9) 0)))
 (and (>= (- nsnt f) 1) (< (- nsnt f) (+ t 1))
    (or (/= (k 0) 0) (/= (k 3) 0) (/= (k 6) 0) (/= (k 9) 0)))
)))
(check)
```

Listing 19: Does state 16 have a concretization that meets justice constraints?

This query is unsatisfiable and Yices gives us an unsatisfiable core that we track in Promela as we did with the spurious transitions:

```
/* update counters */
...
r0 = k[0] == 0 && k[1] == 0
  && k[2] == 0 && k[3] == 0 && k[4] == 0
  && k[5] == 0 && k[7] == 1 && k[10] == 0;
```

and modify each specification φ to avoid infinite occurrences of `r0`:

```
(<>[]r0) || φ
```

4.6 Experiments

To show the feasibility of our abstractions, we implemented the PIA abstractions and the refinement loop in OCaml as a prototype tool ByMC. We evaluated it on different broadcasting algorithms. They deal with different fault models and resilience conditions. The algorithms are: (Byz), which is our use case algorithm (Algorithm 2.1 whose CFA given in Figure 2.1), for t Byzantine faults if $n > 3t$, (symm) for t symmetric (identical Byzantine [8]) faults if $n > 2t$, (omit) for t send omission faults if $n > 2t$, and (clean) for t clean crash faults [96] if $n > t$. (symm), (omit) and (clean) are basically Algorithm 2.1 with different fault assumptions as explained in Section 2.4.4. In addition, we verified the folklore broadcasting algorithm (FBC), which was also formalized in [47]. The formalization of FBC is discussed in Sections 5.1.

Table 4.1 summarizes our experiments run on 3.3GHz Intel® Core$^{\text{TM}}$ 4GB. In the cases (A) we used resilience conditions as provided by the literature, and verified the specification. The model FBC is the folklore reliable broadcast algorithm under the resilience condition $n \geq t \geq f$. In the bottom part of Table 4.1 we used different resilience conditions under which we expected the algorithms to fail. The cases (B) capture the case where more faults occur than expected by the algorithm designer ($f \leq t + 1$ instead of $f \leq t$), while the cases (C) and (D) capture the cases where the algorithms were designed by assuming wrong resilience conditions (e.g., $n \geq 3t$ instead of $n > 3t$ in the Byzantine case). We omit (clean) as the only sensible case $n = t = f$ (all processes are faulty) results into a trivial abstract domain of one interval $[0, \infty)$. The column "#R" gives the number of refinement steps. In the cases where #R is greater than zero, refinement was necessary, and "Spin Time" refers to the Spin running time after the last refinement step. Finally, column $|\widehat{D}|$ indicates the size of the abstract domain.

4.7 Related Work

Most of the previous research on parameterized model checking focused on concurrent systems with $n + c$ processes where n is the parameter and c is a *constant*. n of the processes are *identical* copies and$c$ processes represent the non-replicated part of the system, e.g., cache directories, shared memory, dispatcher processes etc. [48, 56, 74, 29]. Algorithms in the heard-of model were verified by (bounded) model checking [93].

Partial order reductions for a class of fault-tolerant distributed algorithms (with "quorum transitions") for fixed-size systems were introduced in [14].

None of the above methods consider parameterized model checking of FTDAs. To the best of our knowledge there are two papers on parameterized model checking of FTDAs [47, 7]. The authors of [47] use regular model checking to make interesting theoretical progress, but did not do any implementation. Their models are limited to processes

$M \models \varphi$?	RC	Spin Time	Spin Memory	Spin States	Spin Depth	$\lvert\widehat{D}\rvert$	#R	Total Time
Byz $\models U$	(a)	2.3 s	82 MB	483k	9154	4	0	4 s
Byz $\models C$	(a)	3.5 s	104 MB	970k	20626	4	10	32 s
Byz $\models R$	(a)	6.3 s	107 MB	1327k	20844	4	10	24 s
Symm $\models U$	(a)	0.1 s	67 MB	19k	897	3	0	1 s
Symm $\models C$	(a)	0.1 s	67 MB	19k	1113	3	2	3 s
Symm $\models R$	(a)	0.3 s	69 MB	87k	2047	3	12	16 s
Omit $\models U$	(a)	0.1 s	66 MB	4k	487	3	0	1 s
Omit $\models C$	(a)	0.1 s	66 MB	7k	747	3	5	6 s
Omit $\models R$	(a)	0.1 s	66 MB	8k	704	3	5	10 s
Clean $\models U$	(a)	0.3 s	67 MB	30k	1371	3	0	2 s
Clean $\models C$	(a)	0.4 s	67 MB	35k	1707	3	4	8 s
Clean $\models R$	(a)	1.1 s	67 MB	51k	2162	3	13	31 s
FBC $\models U$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
FBC $\models F$	—	0.1 s	66 MB	1.7k	333	2	0	1 s
FBC $\models R$	—	0.1 s	66 MB	1.2k	259	2	0	1 s
FBC $\not\models C$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
Byz $\not\models U$	(b)	5.2 s	101 MB	1093k	17685	4	9	56 s
Byz $\not\models C$	(b)	3.7 s	102 MB	980k	19772	4	11	52 s
Byz $\not\models R$	(b)	0.4 s	67 MB	59k	6194	4	10	17 s
Byz $\models U$	(c)	3.4 s	87 MB	655k	10385	4	0	5 s
Byz $\models C$	(c)	3.9 s	101 MB	963k	20651	4	9	32 s
Byz $\not\models R$	(c)	2.1 s	91 MB	797k	14172	4	30	78 s
Symm $\not\models U$	(b)	0.1 s	67 MB	19k	947	3	0	2 s
Symm $\not\models C$	(b)	0.1 s	67 MB	18k	1175	3	2	4 s
Symm $\models R$	(b)	0.2 s	67 MB	42k	1681	3	8	12 s
Omit $\models U$	(d)	0.1 s	66 MB	5k	487	3	0	1 s
Omit $\not\models C$	(d)	0.1 s	66 MB	5k	487	3	0	2 s
Omit $\not\models R$	(d)	0.1 s	66 MB	0.1k	401	3	0	2 s

Table 4.1: Summary of experiments in the parameterized case.

whose local state space and transition relation are *finite and independent of parameters*. This was sufficient to formalize a reliable broadcast algorithm that tolerates crash faults, and where every process stores whether it has received at least one message. Such models are *not sufficient* to capture FTDAs that contain threshold guards as in our case. Moreover, the presence of a resilience condition such as $n > 3t$ would require them to intersect the regular languages, which describe sets of states, with context-free languages that enforce the resilience condition. In [7], the safety of synchronous broadcasting algorithms that tolerate crash or send omission faults has been verified. These FTDAs have similar restrictions as the ones considered in [47]: Alberti et al. [7] mention that they did not consider FTDAs that feature "substantial arithmetic reasoning", i.e., threshold guards and resilience conditions, as they would require novel suitable techniques. Our abstractions address this arithmetic reasoning.

Interval abstraction [30] is a natural solution to the problem of unboundedness of local variables. However, if we fixed the interval bounds to numeric values, then they would not be aligned to the thresholds, and the abstraction would not be sufficiently precise to do parametric verification. At the same time, we do not have to deal with symbolic ranges over *variables* in the sense of [86], because for FTDAs the interval bounds are *constant* in each run.

Further, we want to produce a single process skeleton that is independent of parameters and captures the behavior of *all* process instances. This can be done by using ideas from existential abstraction [26, 32, 63] and sound abstraction of fairness constraints [63]. We combined these two ideas to arrive at PIA data abstraction.

The PIA counter abstraction is similar to [83] in a way that counters range over an abstract domain, and increment and decrement is done using existential abstraction. The domain in [83] consists of three values representing 0, 1, or *more*. This domain is sufficient for mutual-exclusion-like problems. It allows to distinguish good from bad states, while it is not possible and also not necessary to distinguish two bad states. A bad state is one where at least two processes are in the critical section, which is precisely abstracted in the three-valued domain. However, two bad states where, e.g., 2 and 3 processes are in the critical section cannot be distinguished. Verification of threshold-based FTDAs requires more involved counting: e.g., we have to capture whether at least $n - t$ processes or at most t processes incremented $nsnt$. Therefore, we use counters from the PIA domain.

Our abstraction differs from interval abstraction [30] since in our case the interval bounds are not numeric. However, for every instance, the boundaries are *constant* because the parameters are fixed. We hence do not have to deal with symbolic ranges over *variables* in the sense of [86].

CHAPTER 5

Selected FTDAs in PROMELA

In Chapter 2 we presented our formalization method and validated it using experimental results. We used Algorithm 2.1 as a use case for this purpose. In this chapter we consider three other threshold-based FTDAs each with a different fault model: the Folklore Reliable Broadcast Algorithm from [19] which tolerates non-clean crash faults, the Asynchronous Byzantine Agreement Algorithm from [16] and the Asynchronous Condition-Based Consensus Algorithm from [77], which tolerates clean crash faults. We formalize these algorithms using extended CFA and Promela and also present experimental results by verifying these algorithms for various combinations of parameter values.

5.1 Folklore Reliable Broadcast Algorithm

Algorithm 5.1 Core logic Folklore Broadcast Algorithm from [19].

```
code for a correct process $i$


$v_i$ in {FALSE, TRUE} <- FALSE
$accept_i$ in {FALSE, TRUE} <- FALSE

CODE

if $v_i$ or received (echo) from some other process and
not sent ⟨echo⟩ before
then
  send ⟨echo⟩ to all;
  $accept_i$ <- TRUE;
```

Algorithm 5.1 shows the core logic of the Folklore Reliable Broadcast Algorithm from [19]. This algorithm is designed to tolerate non-clean crash faults (Section 2.4.4). The algorithm works as follows. A process broadcasts a message by sending it to all processes. Upon the reception of a message for the first time by a process, it sends the message to all the processes in the system and accepts it.

As with Algorithm 2.1, we limit the number of broadcasting processes to at most one at a time, in order to limit the number of messages types. We are able to use this restriction since the messages sent by individual broadcasters do not interfere with each other. For this purpose we use the variable v_i. $v_i = \textsc{true}$ indicates that the process i has received the message from the broadcasting process and $v_i = \textsc{false}$ indicates otherwise. Thus, if a process starts with $v_i = \textsc{true}$, and it has not sent an $\langle$echo$\rangle$ yet, it sends an $\langle$echo$\rangle$ to every process and accepts. Also, if a process has received $\langle$echo$\rangle$ and has not sent $\langle$echo$\rangle$ yet then it sends an $\langle$echo$\rangle$ to every process and accepts.

As we can see from the pseudo code, there is only one message type in the system: $\langle$echo$\rangle$. The send operation always sends to all processes in the system. A process sends message only if it is correct. Lines 9 - 13 represent one single step of a process.

Details about reliable broadcast and the related specifications are explained in Section 2.4.1. Folklore Reliable Broadcast Algorithm is also considered in [47], where the authors use an agreement property, which can be defined as follows:

(F) Agreement From a certain point of time, no two correct processes decide differently.

This property can be formalized as follows:

$$(\mathbf{G}\,[\forall i.\ (sv_i = \mathrm{V0} \vee sv_i = \mathrm{V1})] \rightarrow \mathbf{F}\,\mathbf{G}\ (\neg[\exists i.\ sv_i = \mathrm{AC}] \vee [\forall i.\ sv_i = \mathrm{AC}])) \tag{F}$$

Thus, in addition to the specifications of reliable broadcast (unforgeability U, relay R and correctness C), we also consider the property F while verifying this algorithm.

Control flow: Now we will explain status values required for modeling the Folklore Reliable Broadcast Algorithm. As already mentioned, the variable v_i may take the values $\textsc{true}$ or $\textsc{false}$. We set the status variable of a process to V0 if it starts with $v_i = \textsc{false}$ and to V1 if it starts with $v_i = \textsc{true}$ by V_1 . Thus, the set of initial status values $SV_0 = \{\mathrm{V0}, \mathrm{V1}\}$. We can observe from the pseudo code that there are two more important distinctions that we should make in the status of a process, that is, whether a process has sent an $\langle$echo$\rangle$ and whether it has accepted. We use SE and AC to distinguish these two control states, where SE denotes that the process has sent an $\langle$echo$\rangle$ already, but has not accepted yet and AC denotes that the process has accepted.

Modeling non-clean crash faults: Modeling of non-clean crash faults is explained in detail in Section 2.4.4. The faulty processes are modeled explicitly, by using a special status value CR to denote that a process has crashed. We let a process crash by assigning the value CR to the status variable of the process. Thus, the status variable of a process can take one of the five values from the set of status values $SV = \{\mathrm{V0}, \mathrm{V1}, \mathrm{SE}, \mathrm{AC}, \mathrm{CR}\}$.

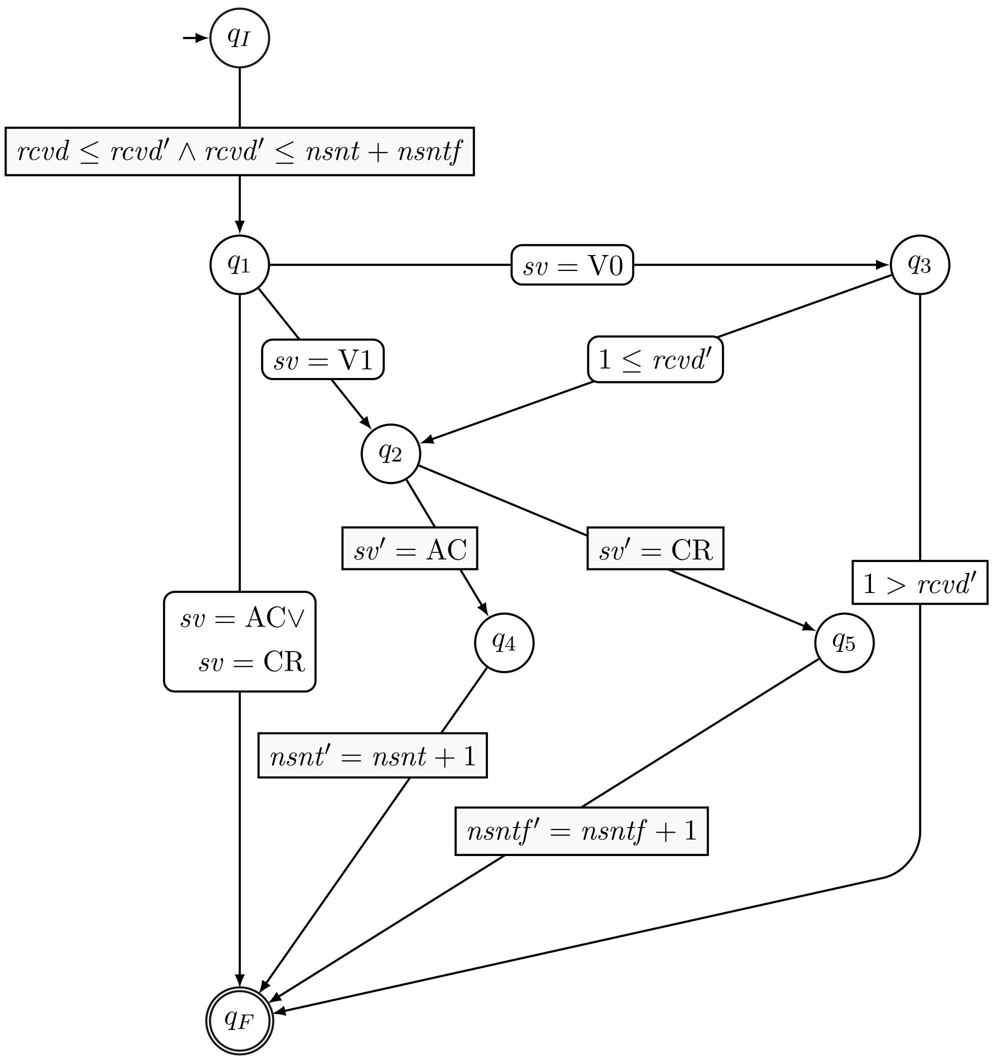


Figure 5.1: CFA of Algorithm 5.1 (if x' is not assigned, then $x' = x$).

Upon crashing, the process increments the global variable *nsntf* (if it is required to send a message), which denotes the number of messages sent by the crashed processes. This algorithm does not limit the number of crashes. Hence we do not count the number of crashes.

We use the local variable *rcvd* to model the reception of the messages and the global variable *nsnt* to model the number of messages sent in the system at any point of time, as explained in the modeling of Algorithm 2.1 in Section 2.4.2. Thus, we obtain the CFA for Algorithm 5.1 as given in Figure 5.1 and the symbolic PROMELA code as given in Listing 20.

```
symbolic int N;
int nsnt, nsntF;
atomic in_transit
    = some(Proc:nrcvd < nsnt);
...
active[N] proctype STFolklore() {
 ...
 if /* receive */
  :: next_nrcvd < nsnt + nsntF ->
   next_nrcvd = nrcvd + 1;
  :: next_nrcvd = nrcvd;
 fi;
 if /* compute */
  :: sv == V1 ->
    next_sv = AC;
  :: sv == V1 ->
    next_sv = CR; /* crash */
  :: sv != AC && sv != CR && (next_nrcvd >= 1) ->
    next_sv = AC;
  :: sv != AC && sv != CR && next_nrcvd >= 1 ->
    next_sv = CR;
  :: else -> next_sv = sv;
 fi;
 if /* send */
  :: (pc == V0 || pc == V1)
   && next_pc == AC -> nsnt++;
  :: (pc == V0 || pc == V1)
   && (next_pc == CR) -> nsntF++;
  :: else
 fi;
 ...
ltl fkl { [](prec_init -> <>[](!ex_acc || all_acc)) }
```

Listing 20: Fragment of Algorithm 5.1 in Symbolic PROMELA

5.2 Asynchronous Byzantine Agreement Algorithm

The second algorithm we consider is the Asynchronous Byzantine Agreement Algorithm from [16]. It works as follows: Similar to the previous algorithm, the broadcasting process sends the message to all processes. A process which receives a message from a broadcaster

Algorithm 5.2 Core logic of Asynchronous Byzantine Agreement Algorithm from [16].

```
1  code for a correct process $i$
2
3  $v_i$ in {FALSE, TRUE}<- FALSE
4  $\text{accept}_i$ in {FALSE, TRUE} <- FALSE
5
6  CODE
7
8  if $v_i$ and not sent $\langle\text{echo}\rangle$ before
9  then
10   send$\langle\text{echo}\rangle$ to all;
11 if received $\langle\text{echo}\rangle$ from at least $\lfloor(n+t)/2\rfloor$ distinct processes and not sent
12   $\langle\text{ready}\rangle$ before
13 then
14    send $\langle\text{ready}\rangle$ to all;
15 if received $\langle\text{ready}\rangle$ from at least 2t+1 distinct processe
16 then
17    $\text{accept}_i$ <- TRUE;
```

for the first time relays the message to all processes via $\langle\text{echo}\rangle$ messages. If a process receives at least $\lfloor(n+t)/2\rfloor$ of such $\langle\text{echo}\rangle$ messages, it sends $\langle\text{ready}\rangle$ to all processes. When a process has received $(2k+1)$ $\langle\text{ready}\rangle$ messages, it decides on that value by setting $accept_i$ to TRUE.

The pseudo code for the core logic of Asynchronous Byzantine Agreement Algorithm is given in Algorithm 5.2. As is obvious from the name of the algorithm, it tolerates Byzantine faults. Using the same reasoning as for Algorithm 5.1, we limit the number of broadcasters to at most one, by using the variable v_i. $v_i =$ TRUE indicates that the process i has received a message from the broadcaster. Otherwise, it has not received any messages from the broadcaster yet. Lines 8 - 17 denote one step of a process. A send operation always sends to all. There are two message types in this algorithm: $\langle\text{echo}\rangle$ and $\langle\text{ready}\rangle$. Thus, the Byzantine faulty processes affect the correct processes only when the messages they send belong to one of these two message types: others are ignored.

The specifications for this algorithm are those for reliable broadcast: unforgeability U, relay R and correctness C as given in Section 2.4.1.

Control flow: As in the previous example, the variable v_i can take two values: TRUE or FALSE. The different states to be distinguished here are (i) whether a process starts from $v_i =$ TRUE or $v_i =$ FALSE, (ii) whether a process has sent an $\langle\text{echo}\rangle$, (iii) whether it has sent a $\langle\text{ready}\rangle$ and (iv) whether it has accepted by setting $\text{accept}_i =$ TRUE. The set of initial status values, $SV_0 = \{\text{V0}, \text{V1}\}$, where V0 corresponds to a process starting with $v_i =$ FALSE and V1 corresponds to a process starting with $v_i =$ TRUE. SE denotes that the process has sent the $\langle\text{echo}\rangle$ and SR denotes that the process has sent a $\langle\text{ready}\rangle$ already, but has not accepted yet. AC denotes that the process has already accepted. Thus the set of status values $SV = \{\text{V0}, \text{V1}, \text{SE}, \text{SR}, \text{AC}\}$.

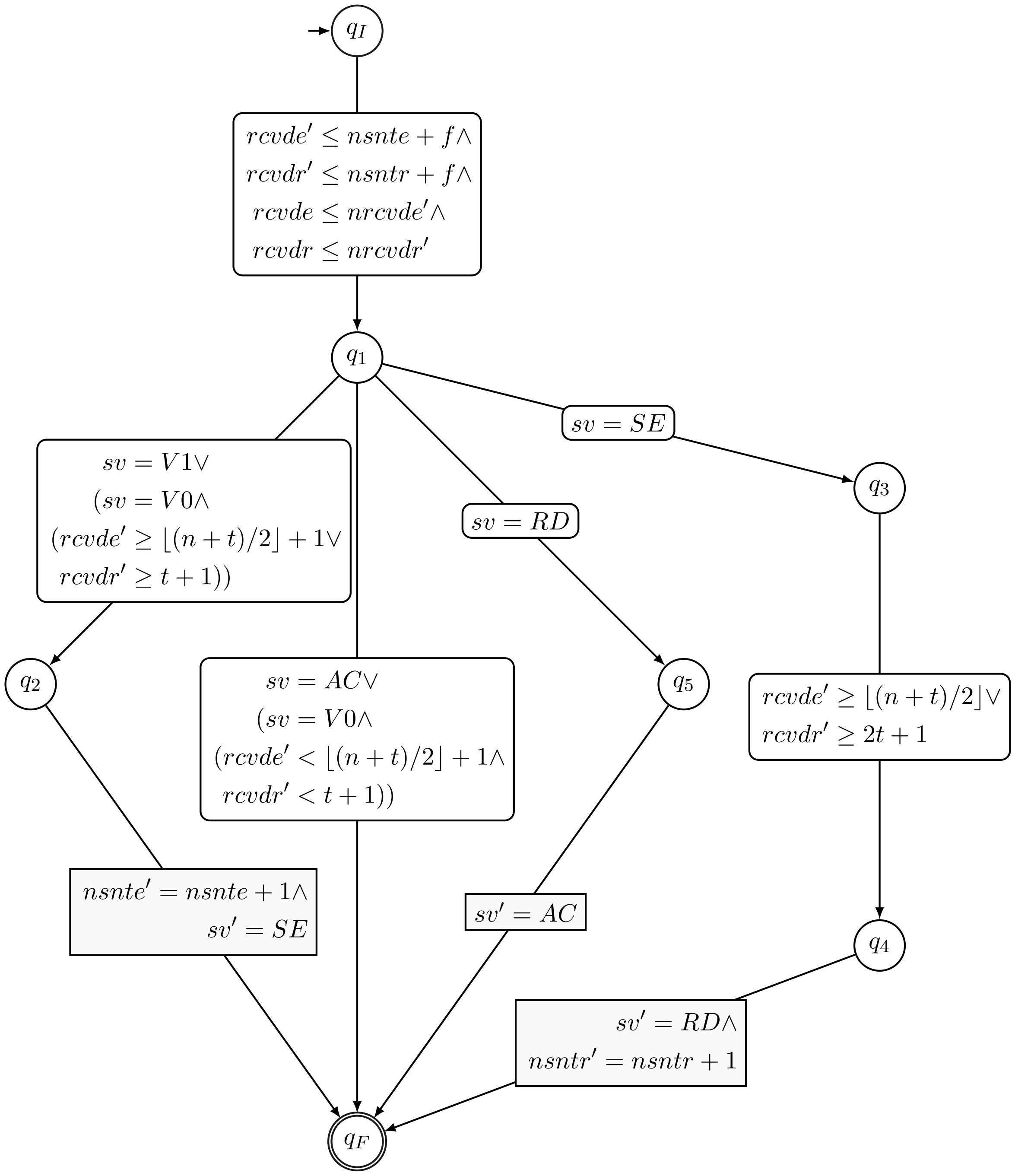


Figure 5.2: CFA of Algorithm 5.2 (if x' is not assigned, then $x' = x$).

Modeling byzantine faults: Byzantine faults are modeled as explained in Section 2.4. Since we have two types of messages in this algorithm, we need to use two variables per message type to model the send and receive operations. The local variables *rcvde* and *rcvdr* are used to keep track of the reception of ⟨echo⟩ and ⟨ready⟩, respectively. Similarly, we use two global variables to model the number of messages ⟨echo⟩ and ⟨ready⟩ sent in the system: *nsnte* and *nsntr*, respectively.

Thus, we get the formalization of the algorithm as given in the CFA in Figure 5.2 and the symbolic Promela code as given in Listing 21.

```
symbolic int N, T, F;
int nsnte, nsntr;
assume(N > 3); assume(F >= 0);
assume(T >= 1); assume(N > 3 * T);
assume(F <= T);
assume(((N + T) / 2 + 1) > (2 * T + 1));/*To impose order*/
atomic prec_unforg = all(Proc:pc == V0);
atomic prec_corr = all(Proc:pc == V1);
atomic prec_inV0 = all(Proc@end);
atomic prec_no0 = all(Proc:pc != V1);
atomic ex_acc = some(Proc:pc == AC);
atomic all_acc = all(Proc:pc == AC);
atomic in_transV0e = some(Proc:nrcvde < nsnte);
atomic in_transV0r = some(Proc:nrcvdr < nsntr);
active[N-F] proctype Proc() {
    byte pc = 0, next_pc = 0;
    int nrcvde = 0, next_nrcvde = 0;
    int nrcvdr = 0, next_nrcvdr = 0;
 if /* INIT */
    :: pc = V0;
    :: pc = V1;
 fi;    end:
 do
 :: atomic {
    if
      :: (nrcvde < nsnte + F) ->  next_nrcvde = nrcvde + 1;
      :: next_nrcvde = nrcvde;
    fi;
      assume(nrcvde <= nsnte + F);
      assert(next_nrcvde <= N + F);
    if
      :: (nrcvdr < nsntr + F) -> next_nrcvdr = nrcvdr + 1;
      :: next_nrcvdr = nrcvdr;
    fi;
      assume(nrcvdr <= nsntr + F);
      assert(next_nrcvdr <= N + F);
/* a step by FSM: find the next value of the program counter*/
    if
      :: pc == V1 ->  next_pc = EC;
      :: (pc == V0) && ((next_nrcvde >= (N + T)/2 + 1)
          || (next_nrcvdr >= (T + 1))) ->  next_pc = EC;
      :: (pc == EC) && ((next_nrcvde >= (N + T)/2 + 1)
           || (next_nrcvdr >= (T + 1))) -> next_pc = RD;
      :: (pc == RD) && (next_nrcvdr >= (2*T + 1)) -> next_pc = AC;
      :: else -> next_pc = pc;
    fi;/* send the echo and ready messages */
    if
      :: ((pc == V0) || (pc == V1)) && (next_pc == EC) -> nsnte++;
      ::(pc == EC) && next_pc == RD  -> nsntr++;
```

```
        :: else;
      fi;
      pc = next_pc;   nrcvde = next_nrcvde;
      nrcvdr = next_nrcvdr;  next_pc = 0;
      next_nrcvde = 0; next_nrcvdr = 0; }
  od; }
ltl fairness { []<>(!in_transV0e && !in_transV0r) }
ltl agreement { [](ex_acc -> <>(all_acc))}
ltl corr { []((prec_inV0 && prec_corr) -> <>(ex_acc)) }
ltl unforg { []((prec_inV0 && prec_unforg) -> []!ex_acc) }
```

Listing 21: Fragment of Algorithm 5.2 in Symbolic PROMELA

5.3 Asynchronous Condition-Based Consensus Algorithm

Algorithm 5.3 Core logic of Condition-Based Consensus Algorithm from [77].

```
code for a correct process $i$

$v_i$ in $\{0, 1\}$
$estimate_i$ in $\{0, 1\}$
$phase_i$ in $\{0, 1\}$ <- 0
$accept_i$ in $\{ 0, 1, \perp \}$ <- $\perp$

CODE

send ($phase_i$, $v_i$) to all
if $phase_i$ is 0 and not sent estimate yet
and received at least (n-t) distinct phase0 messages
then
   if number of $v_i$=0 messages received > $v_i$=1 messages
    then
      $estimate_i$ <- 0;
      $phase_i$ <- 1;
      send ($phase_i$, $estimate_i$) to all;
if $phase_i$ is 0 and not sent estimate yet
and received at least (n-t) distinct phase0 messages
then
   if number of $v_i$=1 messages received > $v_i$=0 messages
    then
      $estimate_i$ <- 1;
      $phase_i$ <- 1;
      send ($phase_i$, $estimate_i$) to all;
if $phase_i$ is 1 and not accepted yet
and received at least (n-1)/2 distinct ($phase_i$=1, $estimate_i$=0) messages
   then  $accept_i$ <- 0;

if $phase_i$ is 1 and not accepted yet
and received at least (n-1)/2 distinct ($phase_i$=1, $estimate_i$=1) messages
   then $accept_i$ <- 1;
```

The Asynchronous Condition-Based Consensus Algorithm as presented in [77] works as follows: The algorithm works in two phases. In phase 0, all the processes send its value to all. Each process then computes an estimate value when it has received $n - t$ distinct messages from phase 0. The estimate value can be chosen based on different conditions: for instance, the most frequently occurring value or the maximum value. We calculate the estimate as the most frequently occurring value in the input vector. On calculating the estimate, a process moves to phase 1 and sends this estimate value to all processes if it has not already done so. In phase 1, each process accepts the estimate value which it has received from a majority of processes. This algorithm tolerates clean crash faults.

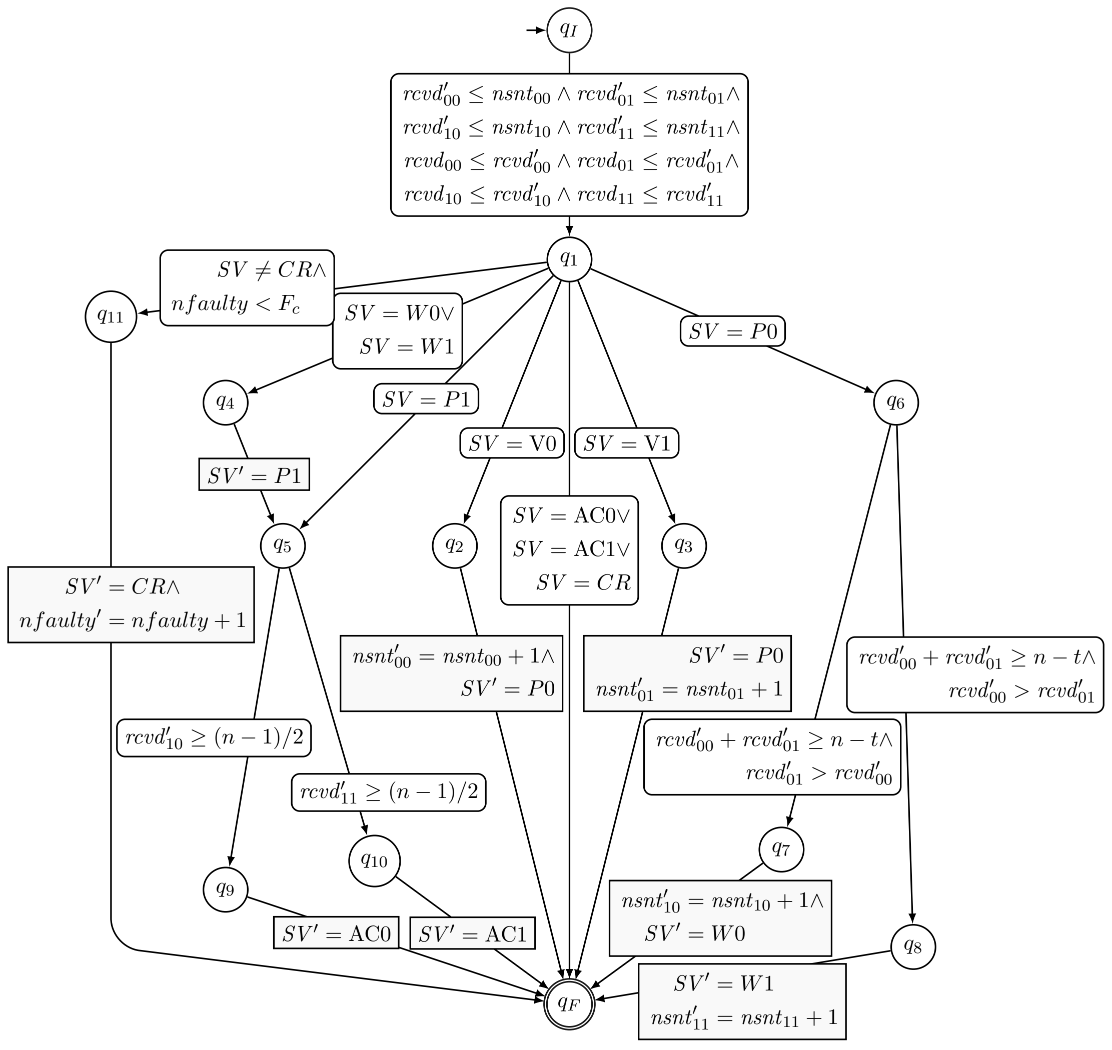

q_I
$rcvd'_{00} \leq nsnt_{00} \wedge rcvd'_{01} \leq nsnt_{01} \wedge$
$rcvd'_{10} \leq nsnt_{10} \wedge rcvd'_{11} \leq nsnt_{11} \wedge$
$rcvd_{00} \leq rcvd'_{00} \wedge rcvd_{01} \leq rcvd'_{01} \wedge$
$rcvd_{10} \leq rcvd'_{10} \wedge rcvd_{11} \leq rcvd'_{11}$
q_1
$SV \neq CR \wedge$
$nfaulty < F_c$
q_{11}
$SV = W0 \vee$
$SV = W1$
$SV = P0$
q_4
$SV = P1$
q_6
$SV = \mathrm{V0}$
$SV = \mathrm{V1}$
$SV' = P1$
$SV = \mathrm{AC0} \vee$
$SV = \mathrm{AC1} \vee$
$SV = CR$
q_5
q_2
q_3
$SV' = CR \wedge$
$nfaulty' = nfaulty + 1$
$nsnt'_{00} = nsnt_{00} + 1 \wedge$
$SV' = P0$
$SV' = P0$
$nsnt'_{01} = nsnt_{01} + 1$
$rcvd'_{00} + rcvd'_{01} \geq n - t \wedge$
$rcvd'_{00} > rcvd'_{01}$
$rcvd'_{10} \geq (n-1)/2$
$rcvd'_{00} + rcvd'_{01} \geq n - t \wedge$
$rcvd'_{01} > rcvd'_{00}$
$rcvd'_{11} \geq (n-1)/2$
q_{10}
q_7
q_9
$nsnt'_{10} = nsnt_{10} + 1 \wedge$
$SV' = W0$
$SV' = \mathrm{AC0}$
$SV' = \mathrm{AC1}$
q_8
$SV' = W1$
$nsnt'_{11} = nsnt_{11} + 1$
q_F

Figure 5.3: CFA of Algorithm 5.3 (if x' is not assigned, then $x' = x$).

Algorithm 5.3 shows the core logic of Asynchronous Condition-Based Consensus

Algorithm. We reduce the problem to be solved to binary consensus by limiting the set of input values to {0, 1}. Lines 10 - 33 represent one single step.

Control flow: As mentioned before, there are two input values possible (0 and 1) and each correct process has to accept one of these values. A process can start with either $v_i = 0$ or $v_i = 1$. We distinguish these two cases with the status values V0 and V1, respectively: that is the set of initial states, $SV_0 = \{\mathrm{V0}, \mathrm{V1}\}$. The two phases in which the algorithm operates are distinguished by the status values $P0$ and $P1$. In $P0$, the processes send their estimate values and in $P1$ they accept a value, both based on the number of each values received. The other states to be distinguished are (i) whether a process has sent the estimate value and (ii) whether it has accepted a value. We use the status values $W0$ and $W1$to denote that the process has sent the estimate value 0 and 1, respectively. Similarly, the status values $AC0$ and $AC1$ are used to denote that the process has accepted value 0 and 1, respectively. Thus, $SV = \{V0, V1, P0, P1, W0, W1, AC0, AC1\}$ is the set of status values.

Specifications: The specifications for the consensus problem are as follows [77]:

(V0, V1) Validity. If a process decides a value v, then v was proposed by some process.

(A) Agreement At any point of time, no two processes decide differently.

(T) Termination Every process eventually decides on some value.

Note that (V0) and (A) are safety properties and (T) is a liveness property.

```
1  symbolic int N, T, F;
2  int nsnt00, nsnt01, nsnt10, nsnt11, nrcvd00,  nrcvd01,  nrcvd10,  nrcvd11;
3  int init0, init1, nfaulty;
4  assume(N > 2); assume(F >= 0); assume(T >= 1); assume(N > 2 * T); assume(F <= T);
5  /* two orders between the thresholds are possible: */
6  assume (F >= 1); assume (F == 0);
7  atomic prec_no0 = all(Proc:pc != V0); atomic prec_no1 = all(Proc:pc != V1);
8  atomic ex_acc0 = some(Proc:pc == AC0); atomic ex_acc1 = some(Proc:pc == AC1);
9  atomic prec_init = ((init0 + init1) == N);
10 atomic cond_init = ((init0 > (init1 + F)) || (init1 > (init0 + F)));
11 atomic all_acc = all(Proc:pc == CR ||Proc:pc == AC0 || Proc:pc == AC1);
12 atomic in_transit00 = some(Proc:nrcvd00 < nsnt00);
13 atomic in_transit01 = some(Proc:nrcvd01 < nsnt01);
14 atomic in_transit10 = some(Proc:nrcvd10 < nsnt10);
15 atomic in_transit11 = some(Proc:nrcvd11 < nsnt11);
16 active[N] proctype Proc() {
17    byte pc = 0, next_pc = 0;
18    int nrcvd00 = 0, next_nrcvd00 = 0, nrcvd01 = 0, next_nrcvd01 = 0;
19    int nrcvd10 = 0, next_nrcvd10 = 0,  nrcvd11 = 0, next_nrcvd11 = 0;
20    if /* INIT */
21      :: pc = V0 -> init0++;
22      :: pc = V1 -> init1++;
23    fi;
24 end:
25 do
26   :: atomic {
27      if
```

```
      :: (pc == V0 || pc == V1 || pc == P0) && (nrcvd00 < nsnt00) ->
          next_nrcvd00 = nrcvd00 + 1;
      :: next_nrcvd00 = nrcvd00; fi;
     if
      :: (pc == V0 || pc == V1 || pc == P0) && (nrcvd01 < nsnt01) ->
          next_nrcvd01 = nrcvd01 + 1;
      :: next_nrcvd01 = nrcvd01; fi;
     if
      :: (pc == W0 || pc == W1 || pc == P1) && (nrcvd10 < nsnt10) ->
          next_nrcvd10 = nrcvd10 + 1;
      :: next_nrcvd10 = nrcvd10; fi;
     if
      :: (pc == W0 || pc == W1 || pc == P1) && (nrcvd11 < nsnt11) ->
          next_nrcvd11 = nrcvd11 + 1;
      :: next_nrcvd11 = nrcvd11; fi;
     if  /* a step by FSM: find the next value of the program counter */
      :: pc == V0 || pc == V1 ->  next_pc = P0;
      :: (pc == P0) && ((next_nrcvd00 + next_nrcvd01) >= N - T )
          && (next_nrcvd00 > next_nrcvd01) ->   next_pc = W0;
      :: (pc == P0) && ((next_nrcvd00 + next_nrcvd01) >= N - T )
           && (next_nrcvd01 > next_nrcvd00) ->  next_pc = W1;
      :: pc == W0 || pc == W1 ->  next_pc = P1;
      :: (pc == P1) && (next_nrcvd10 >= ((N-1) / 2)+1) ->  next_pc = AC0;
      :: (pc == P1) && (next_nrcvd11 >= ((N-1) / 2)+1) -> next_pc = AC1;
      :: nfaulty < F && pc != CR ->  nfaulty++; next_pc = CR;
      ::else -> next_pc = pc;   fi;
     if /* send the echo message */
      :: (pc == V0) && (next_pc == P0) ->  nsnt00++;
      :: (pc == V1) && (next_pc == P0) ->  nsnt01++;
      :: (pc == P0) && (next_pc == W0) -> nsnt10++; next_nrcvd00 = 0; next_nrcvd01
          = 0;
      :: (pc == P1) && next_pc == W1  -> nsnt11++; next_nrcvd00 = 0; next_nrcvd01 =
           0;
      :: else; fi;
      pc = next_pc;
      nrcvd00 = next_nrcvd00;  nrcvd01 = next_nrcvd01;
      nrcvd10 = next_nrcvd10;  nrcvd11 = next_nrcvd11;
      next_pc = 0;
      next_nrcvd00 = 0;  next_nrcvd01 = 0;
      next_nrcvd10 = 0;  next_nrcvd11 = 0;    }
 od; }
ltl fairness { []<>(!in_transit00 && !in_transit01 && !in_transit10 && !
    in_transit11) }
ltl validity0 { (([](prec_no0) && <>(prec_init)) -> []!ex_acc0) }
ltl validity1 { (([](prec_no1) && <>(prec_init)) -> []!ex_acc1) }
ltl agreement { [](!ex_acc0 || !ex_acc1)}
ltl termination { []((!prec_init || !cond_init) || (<>(all_acc))) }
```

Listing 22: Fragment of Algorithm 5.3 in Symbolic Promela

Modeling clean crash faults: In Section 2.4.4, we discussed two methods to model clean crash faults. Here, we use the second model. That is, we have a global variable *nfaulty*, which keeps track of the number of processes that have crashed. When a process crashes, its status variable is assigned the value CR and the variable *nfaulty* is incremented. Note that a process is allowed to crash only if *nfaulty* is less than the actual number of faulty process F_c in the system.

The local variables used to model the message reception are, $rcvd_{00}$, $rcvd_{01}$, $rcvd_{10}$, $rcvd_{11}$. $rcvd_{00}$ stands for the reception of message 0 in phase 0, $rcvd_{01}$ stands for reception of message1 in phase 0, $rcvd_{10}$ stands for reception of message 0 in phase 1 and $rcvd_{11}$ stands for reception of message 1 in phase 1. Similarly, we use $nsnt_{00}$ to model sending of message 0 in phase 0, $nsnt_{01}$ to model sending of message1 in phase 0, $nsnt_{10}$ to model sending of message 0 in phase 1 and $nsnt_{11}$ to model sending of message 1 in phase 1.

The number of messages received is allowed to take any value up to the number of messages sent for each phase and message value. This includes the message sent by the crashed processes, if they send any. Thus, the fairness condition is represented by the following condition:

$$\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_{00} < nsnt_{00}] \land \tag{5.1}$$
$$\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_{01} < nsnt_{01}] \land \tag{5.2}$$
$$\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_{10} < nsnt_{10}] \land \tag{5.3}$$
$$\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_{11} < nsnt_{11}] \tag{5.4}$$

Now let us formalize the specifications for this Algorithm 5.3 as follows:

$$\mathbf{G}\ ([\forall i.\ sv_i \neq \mathrm{V0}] \rightarrow \mathbf{G}\ [\forall j.\ sv_j \neq \mathrm{AC0}]) \tag{V0}$$
$$\mathbf{G}\ ([\forall i.\ sv_i \neq \mathrm{V1}] \rightarrow \mathbf{G}\ [\forall j.\ sv_j \neq \mathrm{AC1}]) \tag{V1}$$
$$\mathbf{G}\ (\neg\,[\exists i.\ sv_i = \mathrm{AC1}] \lor \neg\,[\exists i.\ sv_i = \mathrm{AC0}]) \tag{A}$$
$$\mathbf{F}\ [(\forall i.\ sv_i = \mathrm{AC0}) \lor (\forall j.\ sv_i = \mathrm{AC1})] \tag{T}$$

The CFA in Figure 5.3 shows the formalization of Algorithm 5.3 and the symbolic Promela code is given in Listing 22.

5.4 Experiments

In this section we list our experimental results for the three algorithms modeled in this chapter for different combinations of parameter values, using Spin.

Table 5.1 summarizes our experiments for Algorithm 5.1, Algorithm 5.2 and Algorithm 5.3. The Property (F) is the agreement property from [47]. Properties (V0) and (V1) are *non-triviality*, that is, if all processes propose 0 (1), then 0 (1) is the only possible decision value. Property (A) is agreement and similar to (R), while Property (T) is termination, and requires that every correct process eventually decides. In all experiments the validity of the specifications was as expected from the distributed algorithms literature.

#	parameter values	spec	valid	Time	Mem.	Stored	Transitions	Depth
				FOLKLORE BROADCAST [19]				
F1	N=2	(U)	✓	0.01 sec.	98 MB	121	$7 \cdot 10^3$	77
F2	N=2	(R)	✓	0.01 sec.	98 MB	143	$8 \cdot 10^3$	48
F3	N=2	((F)	✓	0.01 sec.	98 MB	257	$2 \cdot 10^3$	76
F4	N=6	(U)	✓	386 sec.	670 MB	$15 \cdot 10^6$	$20 \cdot 10^6$	272
F5	N=6	(R)	✓	691 sec.	996 MB	$24 \cdot 10^6$	$370 \cdot 10^6$	272
F6	N=6	((F)	✓	1690 sec.	1819 MB	$39 \cdot 10^6$	$875 \cdot 10^6$	328
				ASYNCHRONOUS BYZANTINE AGREEMENT [16]				
T1	N=5,T=1,F=1	(R)	✓	131 sec.	239 MB	$4 \cdot 10^6$	$74 \cdot 10^6$	211
T2	N=5,T=1,F=2	(R)	✗	0.68 sec.	99 MB	$11 \cdot 10^3$	$465 \cdot 10^3$	187
T3	N=5,T=2,F=2	(R)	✗	0.02 sec.	99 MB	726	$9 \cdot 10^3$	264
				CONDITION-BASED CONSENSUS [77]				
S1	N=3,T=1,F=1	(V0)	✓	0.01 sec.	98 MB	$1.4 \cdot 10^3$	$7 \cdot 10^3$	115
S2	N=3,T=1,F=1	(V1)	✓	0.04 sec.	98 MB	$3 \cdot 10^3$	$18 \cdot 10^3$	128
S3	N=3,T=1,F=1	(A)	✓	0.09 sec.	98 MB	$8 \cdot 10^3$	$42 \cdot 10^3$	127
S4	N=3,T=1,F=1	(T)	✓	0.16 sec.	66 MB	$9 \cdot 10^3$	$83 \cdot 10^3$	133
S5	N=3,T=1,F=2	(V0)	✓	0.02 sec.	68 MB	1724	9835	123
S6	N=3,T=1,F=2	(V1)	✓	0.05 sec.	68 MB	3647	$23 \cdot 10^3$	136
S7	N=3,T=1,F=2	(A)	✓	0.12 sec.	68 MB	$10 \cdot 10^3$	$55 \cdot 10^3$	135
S8	N=3,T=1,F=2	(T)	✗	0.05 sec.	68 MB	$3 \cdot 10^3$	$17 \cdot 10^3$	135

Table 5.1: Summary of experiments with algorithms from [19, 16, 77]

CHAPTER 6

Conclusions

The work done in thesis has been motivated by the importance of ensuring the correctness of FTDAs and the fact that existing work in this area is extremely scarce due to the various challenges faced in verifying FTDAs. We have seen that FTDAs are widely used in many safety critical applications and the traditional way of proving their correctness such algorithms using handwritten proofs requires expert knowledge of the algorithm and is very time consuming. Due to the inherent non-determinism and concurrency of distributed algorithms and uncertainty introduced by the presence of faults, such manual proofs can be error prone. Thus it is highly desirable to automate such proofs as much as possible.

Our aim was to automatically verify an important and widely used class of FTDAs, called threshold-based FTDAs. Thus, in this thesis, we extended the standard setting of parameterized model checking to processes that use threshold guards, and are parameterized with a resilience condition.

6.1 Contributions

In Chapter 2 we presented a method to formalize Threshold-based FTDAs using an extension version of Control Flow Automata. We presented a way to efficiently encode fault-tolerant threshold-guarded distributed algorithms using shared variables. We showed that our encoding scales significantly better than a straightforward approach. With the help of a use case, we showed how to transfer an algorithm given in pseudocode to CFA, and also how to translate the CFA obtained to our symbolic Promela language. In more detail, we first added mild extensions to the syntax of Promela to be able to express the kind of parameterized systems we are interested in. We also showed by experimental evaluation that the standard language constructs for interprocess communication do not scale well, and do not naturally match the required semantics for fault-tolerant distributed algorithms. We thus introduced an efficient encoding of a fault-tolerant distributed algorithm in the extended Promela. We also discussed the modeling approaches for different fault models and presented the the CFA and Promela representations of our use case algorithm in presence of different kind of faults. This representation builds the input for our tool chain, and we discussed in detail how it can be automatically translated into abstract models.

In Chapter 3 we proved the undecidability of parameterized model checking problem (PMCP) for CFAs, by reducing a 2-counter machine to PMCP.

We presented our method of abstraction, the Parameterized Interval Abstraction (PIA) in Chapter 4, to verify threshold-based FTDAs. We showed how the abstraction is applied on two levels (data and counter) to eliminate the unbounded variables in each process and to abstract away the parameters in the system which consists of a parallel combination of all the finite state processes obtained after the first abstraction. We also presented the simulation proofs for both the levels of abstraction. As our abstractions are over-approximations, the model checker returned spurious counterexamples, which we eliminated using counterexample guided abstraction refinement (CEGAR) [25]. In contrast to the classic CEGAR setting, in the parameterized case we have an infinite number of concrete systems which posed new challenges. We discussed several of them and presented the details of the abstraction refinement approach that was sufficient to verify our case studies.

The only way to evaluate the practical use of an abstraction is to conduct experiments on several case studies, and thus demonstrate that the abstraction is sufficiently precise to verify correct distributed algorithms, and find counterexamples in buggy ones. Hence, understanding implementations is crucial to evaluate the theoretical work and they are thus of highest importance. With our encoding we were able to verify small system instances and the parameterized versions of a number of broadcasting algorithms [90, 16, 19] for diverse failure models, as demonstrated in Chapter 5. The experiments with fixed parameters help us to validate the adequacy of our formalization method. For instance, we tried variations in the parameter combinations which violates the resilience condition to see if the specifications are violated. With system sizes as big as $n = 11$, we run out of memory, which points at the need of parameterized verification. We found counterexamples in cases where the actual number of faults exceeded the threshold. We also verified a condition-based consensus algorithm [77].

6.2 Future Work

The case studies presented in this thesis are fundamental algorithms with limited message types. For instance, the state-of-the-art FTDAs for consensus or replicated state machines such as Paxos [66] and other similar algorithms in [37, 19] have characteristics that currently no parameterized model checking tool can deal with. The above mentioned algorithms have executions separated into rounds and the messages carry the round number. Thus, there are multiple message types. Also these algorithms often work based on a (rotating) coordinator, which breaks symmetry. Our current verification method is limited to one round. Also, the symmetry of the processes involved is an essential factor for our abstraction to work. Currently there are no tools which make parameterized model checking of such algorithms possible. Developing new techniques of parameterized model checking that address these issues is future work.

We need to address two issues to be able to handle the parameterized verification of the state-of-the-art algorithms mentioned above. They are,

- New abstraction techniques have to be designed that can handle limited forms of asymmetric systems.
- To handle multiple rounds we must develop compositional methods that allow us to combine verification results of several individual rounds.

Yet another direction of future work is to automatize the simulation proofs. We have presented the simulation proofs for our abstraction in Chapter 4. Automated theorem provers like Isabelle can be used to automatize these proofs instead of manually proving the simulation relation.

Bibliography

[1] ByMC 0.4.0: Byzantine model checker, 2013. Accessed: March, 2014.

[2] Tempo toolset. Web page. `http://www.veromodo.com/`.

[3] TLA – the temporal logic of actions. Web page. `http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html`.

[4] P. Abdulla. Regular model checking. *International Journal on Software Tools for Technology Transfer*, 14:109–118, 2012.

[5] P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996.

[6] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Consensus with Byzantine failures and little system synchrony. In *DSN*, pages 147–155, 2006.

[7] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, 8(1/2):29–61, 2012.

[8] H. Attiya and J. Welch. *Distributed Computing*. John Wiley & Sons, 2nd edition, 2004.

[9] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 64–78. Springer-Verlag, 2009.

[10] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, volume 5643 of *LNCS*, pages 64–78. Springer, 2009.

[11] M. Biely, B. Charron-Bost, A. Gaillard, M. Hutle, A. Schipe, and J. Widder. Tolerating corrupted communication. In *PODC*, pages 244–253, August 2007.

[12] M. Biely, U. Schmid, and B. Weiss. Synchronous consensus under hybrid process and link failures. *Theoretical Computer Science*, 412(40):5602–5630, 2011.

[13] A. Biere. *Handbook of satisfiability*, volume 185. IOS Press, 2009.

[14] P. Bokor, J. Kinder, M. Serafini, and N. Suri. Efficient model checking of fault-tolerant distributed protocols. In *DSN*, pages 73–84, 2011.

[15] R.. Boyer and J. Moore. Proof-checking, theorem-proving and program verification. Technical report, 1983.

[16] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[17] M. Browne, E. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Inf. Comput.*, 81:13–31, April 1989.

[18] P. Chambart and P. Schnoebelen. Mixing lossy and perfect fifo channels. In *CONCUR*, volume 5201 of *LNCS*, pages 340–355, 2008.

[19] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267.

[20] B. Charron-Bost, H. Debrat, and S. Merz. Formal verification of consensus algorithms tolerating malicious faults. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 120–134, 2011.

[21] B. Charron-Bost and S. Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2–3):273–303, 2009.

[22] C. Chou, P. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *in Formal Methods in Computer Aided Design*, pages 382–398. Springer, 2004.

[23] E. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71, 1981.

[24] E. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR*, pages 395–407, 1995.

[25] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[26] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994.

[27] E. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *CONCUR 2004*, volume 3170, pages 276–291, 2004.

[28] E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *In 7 th VMCAI, LNCS 3855*, pages 126–141. Springer, 2006.

[29] E. Clarke, M. Talupur, and H. Veith. Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In *TACAS'08/ETAPS'08*, pages 33–47. Springer, 2008.

[30] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[31] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[32] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, March 1997.

[33] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34:77–97, January 1987.

[34] A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *CAV*, volume 6806 of *LNCS*, pages 356–371. Springer, 2011.

[35] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(7):1165–1178, July 2008.

[36] B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL (T). In *Computer Aided Verification*, pages 81–94. Springer, 2006.

[37] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), April 1988.

[38] E.Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[39] A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, volume 1831 of *LNCS*, pages 236–254. Springer, 2000.

[40] A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME*, volume 2860 of *LNCS*, pages 247–262. Springer, 2003.

[41] A. Emerson and V. Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL*, volume 3210 of *LNCS*, pages 325–339. Springer, 2004.

[42] E. Emerson and K. Namjoshi. Reasoning about rings. In *POPL*, pages 85–94, 1995.

[43] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[44] A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, pages 297–âĂŞ308, 2012.

[45] J. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '83, pages 1–7. ACM, 1983.

[46] J. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[47] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *TACAS*, volume 4963 of *LNCS*, pages 315–331. Springer, 2008.

[48] S. German and P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[49] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder. Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In *Formal Methods for Executable Software Models*, LNCS, pages 122–171. Springer, 2014.

[50] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, CAV '90, pages 176–185, London, UK, UK, 1991. Springer-Verlag.

[51] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. pages 72–83. Springer-Verlag, 1997.

[52] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 2nd edition, 1993.

[53] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, 1994.

[54] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.

[55] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[56] C. Ip and D. Dill. Verifying systems with replicated components in murϕ. In *CAV*, volume 1102 of *LNCS*, pages 147–158. Springer, 1996.

[57] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Brief announcement: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *ACM PODC*, 2013. (to appear; long version at arXiv CoRR abs/1210.3846).

[58] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proceedings Formal Methods in Computer-Aided Design (FMCAD'13)*, pages 201–209. IEEE, 2013.

[59] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Towards modeling and model checking fault-tolerant distributed algorithms. In *Proceedings 20th International Symposium on Model Checking Software (SPIN'13), Springer LNCS 7976*, pages 209–226. Springer, 2013.

[60] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with TLA$^+$. *Formal Methods in System Design*, 22(2):125–131, 2003.

[61] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, volume 6174 of *LNCS*, pages 654–659. Springer, 2010.

[62] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata (Synthesis Lectures in Computer Science).* Morgan & Claypool Publishers, 2006.

[63] Y. Kesten and A. Pnueli. Control and data abstraction: the cornerstones of practical formal verification. *STTT*, 2:328–342, 2000.

[64] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1st edition, 1997.

[65] L. Lamport. Email message sent to a dec src bulletin board at 12:23:29 pdt on 28 may 87.

[66] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[67] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[68] L. Lamport. The pluscal algorithm language. In *ICTAC*, volume 5684 of *LNCS*, pages 36–60, 2009.

[69] L. Lamport. Byzantizing paxos by refinement. In *DISC*, volume 6950 of *LNCS*, pages 211–224, 2011.

[70] P. Lincoln and J. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *FTCS-23*, pages 402–411, jun 1993.

[71] N. Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

[72] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[73] K. Mcmillan. Verification of an implementation of tomasuloâĂŹs algorithm by compositional model checking. pages 110–121. Springer-Verlag, 1998.

[74] K. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In *CHARME*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.

[75] M. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[76] S. Mitra and N. Lynch. Proving approximate implementations for probabilistic I/O automata. *Electr. Notes Theor. Comput. Sci.*, 174(8):71–93, 2007.

[77] A. Mostéfaoui, E. Mourgaya, P. Parvédy, and M. Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*. IEEE Computer Society, 2003.

[78] L. De Mourao and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[79] T. Nipkow and L. C. Paulson. Isabelle-91. In *In Proceedings of the 11th International Conference on Automated Deduction, D. Kapur, Ed. Springer-Verlag LNAI 607*, pages 673–676, 1992.

[80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J.ACM*, 27(2):228–234, April 1980.

[81] D. Peled. All from one, one for all: On model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 409–423, London, UK, UK, 1993. Springer-Verlag.

[82] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[83] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0,1,\infty)$- counter abstraction. In *CAV*, volume 2404 of *LNCS*, pages 93–111. Springer, 2002.

[84] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, August 2002.

[85] J. M. Rushby S. Owre and N. Shankar. Pvs: A prototype verification system. In Deepak Kapur, editor, *Automated DeductionâĂŤCADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin Heidelberg, 1992.

[86] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis using symbolic ranges. In *SAS*, volume 4634 of *LNCS*, pages 366–383, 2007.

[87] N. Santoro and P. Widmayer. Time is not a healer. In *STACS*, volume 349 of *LNCS*, pages 304–313. Springer, 1989.

[88] U. Schmid, B. Weiss, and J. Rushby. Formally verified Byzantine agreement in presence of link faults. In *ICDCS*, pages 608–616, 2002.

[89] T. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[90] T. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

[91] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *DSN*, pages 189–198, 2004.

[92] M. Talupur and M. Tuttle. Going with the flow: Parameterized verification using message flows. In *FMCAD*, pages 1–8. IEEE, 2008.

[93] T. Tsuchiya and A. Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5–6):341–358, 2011.

[94] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, pages 1–22, 1989.

[95] J. Widder, M. Biely, G. Gridling, B. Weiss, and J. Blanquart. Consensus in the presence of mortal Byzantine faulty processes. *Distributed Computing*, 24(6):299–321, 2012.

[96] J. Widder and U. Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Distributed Computing*, 20(2):115–140, August 2007.

[97] S. Wöhrle and W. Thomas. Model checking synchronized products of infinite transition systems. *LMCS*, 3(4), 2007.

APPENDIX A

Running the Tool ByMC

In what follows, we use the tool on the running example `bcast-byz.pml` available in the set of benchmarks `benchmarks-sfm14` at [1]. We also assume that the tool resides in the directory `${bymc}`.

The tool chain supports two modes of operation:

- **Concrete model checking.** In this mode, the user fixes the values of the parameters **p**. The tool instantiates code in standard Promela and performs finite-state model checking with Spin. This step is very useful to make sure that the user code operates as expected without abstraction involved.
- **Parameterized model checking**. In this mode, the tool applies data and counter abstractions and performs finite-state model checking of the abstract model with Spin.

For concrete-state model checking of the `relay` property, one issues the command `verifyco-spin` as follows:

```
$ ${bymc}/verifyco-spin "N=4,T=1,F=1" bcast-byz.pml relay
```

The tool instantiates the model checking problem in the directory "`./x/spin-bcast-byz-relay-N=4,T=1,F=1`". The directory contains the file `concrete.prm` that differs from the source code as follows: The parameters N, T, and F in the Promela code are replaced with the values 4, 1, 1 respectively. The process prototype is replaced with $N - F = 3$ active processes.

In order to run parameterized model checking, one issues `verifypa-spin` as follows:

```
$ ${bymc}/verifypa-spin bcast-omit.pml relay
```

The tool instantiates the model checking problem in a directory, whose name follows the pattern "`./x/bcast-byz-relay-yymmdd-HHMM.*`". The directory contains the following files of interest: `abs-interval.prm` is the result of the data abstraction;

`abs-counter.prm` is the result of the counter abstraction; `abs-vass.prm` is the auxiliary abstraction for the abstraction refinement; `mc.out` contains the last output by Spin; `cex.trace` contains the counterexample (if there is one); `yices.log` contains communication log with Yices.